M2C: A Massive Performance and Energy Throttling Framework for High-Performance Computing Systems

Muhammad Usman Ashraf[1]
Department of Computer Science
University of Management and Technology, Sialkot, Pakistan

Kamal M. Jambi[2]
Department of Computer Science
King Abdulaziz University
Jeddah, Saudi Arabia

Amna Arshad[3], Rabia Aslam[4]
Iqra Ilyas[5]
Department of Computer Science
GCWUS, Sialkot, Pakistan

***Abstract*—At the Petascale level of performance, High-Performance Computing (HPC) systems require significant use of supercomputers with the extensive parallel programming approaches to solve the complicated computational tasks. The Exascale level of performance having 10^{18} calculations per second is another remarkable achievement in computing with a fathomless influence on everyday life. The current technologies are facing various challenges while achieving ExaFlop performance through energy-efficient systems. Massive parallelism and power consumption are vital challenges for achieving ExaFlop performance. In this paper, we have introduced a novel parallel programming model that provides massive performance under power consumption limitations by parallelizing data on the heterogeneous system to provide coarse grain and fine-grain parallelism. The proposed dual-hierarchical architecture is a hybrid of MVAPICH2 and CUDA, called the M2C model, for heterogeneous systems that utilize both CPU and GPU devices for providing massive parallelism. To validate the objectives of the current study, the proposed model has been implemented using bench-marking applications including linear Dense Matrix Multiplication. Furthermore, we conducted a comparative analysis of the proposed model by existing state-of-the-art models and libraries such as MOC, kBLAS, and cuBLAS. The suggested model outperforms existing models while achieving massive performance in HPC clusters and can be considered for emerging Exascale computing systems.**

***Keywords*—High performance computing; Exascale computing; compute unified device architecture**

I. Introduction

The High-Performance Computing (HPC) can practice computing power and parallel processing techniques to resolve extensive enigmas in various disciplines, e.g., medicine, engineering, commercial enterprise, smart cities [51] and so on, by providing much higher performance than that of a traditional desktop computer [1, 2]. A traditional computer system has a single CPU in general. However, an HPC system usually consists of a community of CPUs where each processor contains multi-cores along with its local memory to execute a variety of complicated tasks [38]. HPC systems utilize supercomputers and parallel processing techniques to execute extensive jobs. Therein, thousands of processors operate in parallel to solve extensive problems by supercomputing. On the other hand, large problems are broken down into smaller ones that could be resolved at the same time to enhance the overall performance of HPC systems by parallel computing. If a traditional desktop computer takes 200 hours to complete a specific task while it could be completed in 1 hour by utilizing 200 computers at once by the HPC system. Therefore, a single desktop computer might not be as useful as it could be while utilizing all the resources collectively as a community.

There exist various advancements in the performance of the HPC system from GigaScale to Terascale, to Petascale, to Exascale, each of which constitutes an extraordinary improvement in computing performance. HPC system provides high-end designing and simulation environments, helps applications to deal with marketing delivery challenges by providing the facility to accelerate or even get rid of prototyping and testing phases. The HPC system not only enhances the quality but also predict the failure rate to enhance the overall performance of the product [4, 13]. Moreover, the HPC system supports existing technologies in the research and development process to deliver products to the marketplace more quickly. Similarly, organizations and industries use supercomputers before the actual implementation to build and examine their strategies [5]. The fastest supercomputer can currently solve complex problems using Petascale systems that can perform 10^{15} (quadrillion) calculations each second. Though these Petascale systems are going well in this era, the next milestone in computing advancement is to pace relatively towards high-performance Exascale systems offering outstanding computing power. These advance and powerful HPC systems will reveal many Scientific mysteries and will have a fathomless impact on everyday life [1, 2, 3].

The architecture of such a system might vary from conventional order. The following two options could be possible in this regard. First, all the accelerated processing General Purpose Graphics Processing Unit (GPGPU), and conventional CPU devices will reside exclusively within the node. In the second option, accelerated devices will be separated at the cabinets/rack level. We have shown the broad level conceptual depiction of future Exascale supercomputer (see Fig. 1), in which the heterogeneous system is comprised of a number of racks that can communicate with each other over the network. These racks are further consisting of

multiple nodes and additional components resided in it where each node contained Heterogamous Processing Chip (HPC) having several heterogeneous devices including CPUs GPGPUs as well.

Current supercomputers cannot deliver such a high level of computation under the power consumption limitation. Although developers can extend the cores in the current Petascale systems to devise a way towards Exascale computing system, the challenge of power consumption still exists [43]. The United States Department of Energy has pointed out some primary constraints for the roadmap of Exascale computing systems by considering the financial and power consumption limitations. These constraints include the power consumption not more than 25 to 30 Mega Watts, system development cost around 200 million USD by 2020 and integrated multi-cores no more than 100 million [6, 8, 45].

Current technologies are facing various challenges to meet the above-defined constraints (see Fig. 2) [7].

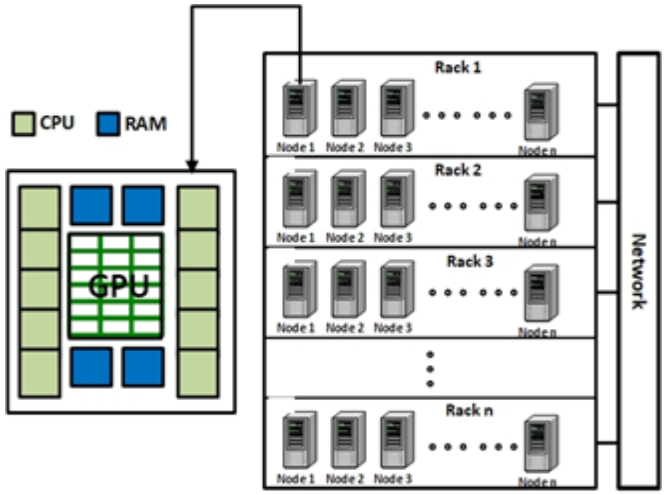

Fig. 1. Predictive Structure of the Exascale Supercomputing System.

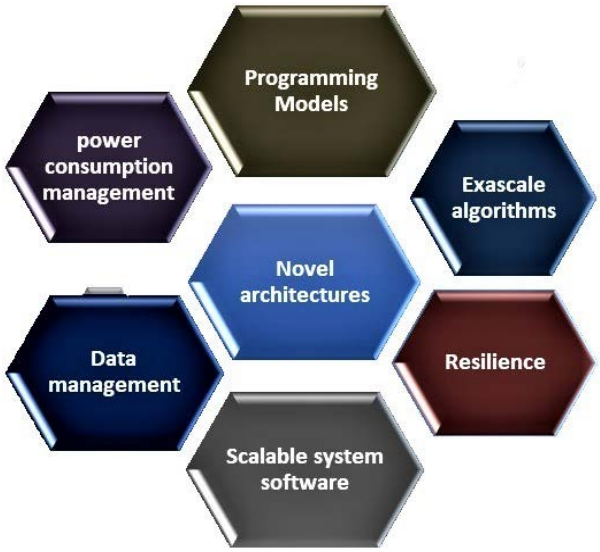

Fig. 2. Challenges of Exascale Supercomputing System.

Exascale systems generally include heaps of nodes drawing excessive Megawatt power, system-wide strategies for power monitoring, controlling, and scheduling. The power consumption of both individual nodes and the overall system is, therefore, an essential issue to cope with [8, 48]. However, new energy-efficient algorithms with novel architecture and devices are needed that can implement the advanced programming models supporting both homogeneous and heterogeneous systems [9, 10, 11].

A considerable number of practical components (computing cores, memory chips, network interfaces) can extensively increase the possibility of partial disasters, load balancing, and reliability issues [3]. Developers cannot be intended to continually cope with load balancing, data management, and reliability issues. The operating system must discover a scalable mechanism that can offer an effective way for load management, memory management, and check-pointing while allowing software developers to complete control over the performance of the system [12, 38].

The objective of the current study is to deal with the two fundamental issues including performance and power consumption HPC Systems. These challenges have a direct relation to the number of resources used. Increase in resources consequently enhance the performance of the system and increase the power consumption as well. The existing programming models and approaches failed to attain such a high level of performance under the constraints defined by the United States Department of Energy [6, 8, 45, 48].

The purpose of this study was to focus on how to enhance the performance of HPC systems with minimum power consumption. Going towards the solution of the problem; we have proposed a Massive Performance and Energy Throttling Framework M2C, for HPC Systems that consumes less power while achieving massive performance efficiently.

The key contributions of this paper can be summarized as follows:

- We have proposed a novel hybrid parallel programming for parallel computing, which is called as MVAPICH2+ CUDA (M2C) model. The proposed M2C model helps to achieve coarse-grained and fine-grained parallelism while parallelizing the data in heterogeneous systems.
- The MVAPICH2 module helps to reduce communication overhead in the heterogeneous system by utilizing the Message Passing Interface (MPI) local handle in the proposed M2C framework and so reduces power consumption.
- The accelerated GPU Computation through CUDA in the M2C model that enables a larger task to run on multiple processors at the same time to enhance the overall performance of the proposed HPC system.

The rest of this paper is structured as follows. In Section 2, we provided an overview of related systems. In Section 3, we described the system model including architecture, flowchart, and algorithm of the proposed M2C framework. In Section 4, we discussed the fundamental HPC metrics which have measured in our experiments. Further Section 5 presents the experimental results, detailed discussion, and comprehensive comparative analysis with existing state-of-the-art methods. Finally, our conclusion follows in Section 6.

II. Related Work

The concept of HPC has been a hot topic in the field of parallel computing for the last two decades. As demonstrated by Moore's law, there has been a rapid evolution of novel hardware and computer architectures to develop parallel computing machines. However, the growth of parallel computing software is not as fast as it should be. One of the reasons for this gap could be the unavailability of desired parallel programming models that support such novel architectures [3]. The traditional models are not capable of delivering such a massive performance as expected in the modern era. Therefore, the demand for novel parallel processing models is increasing day by day which should support homogeneous and heterogeneous systems. Homogeneous cluster systems made up of similar types of cores typically CPUs guarantees the same storage representation. On the other hand, heterogeneous cluster systems usually use more than one kind of processors (or cores) therein CPUs and GPUs are integrated on a single cluster system. If the CPU itself is a multi-core architecture but the cores are different in their capabilities i.e. at least two cores are different in their architecture, such a multi-core architecture is termed heterogeneous multi-core and any embedded system based on it would be termed heterogeneous embedded system. Similarly, an embedded system that has a CPU and a Digital Signal Processor (DSP) is termed heterogeneous embedded system because CPU and DSP are fundamentally different in their architecture. We present an overview of various systems of such types in the following.

A. Message Passing Interface

MPI is the basic library for distributing and communicating the messages among host CPU processors of all the connected nodes [14]. It is used for distributed computing applications and provides an efficient and portable way to address parallel programs. Moreover, to distribute and parallelize data at the inter-node level MPI provides coarse-grained parallelism and maintains synchronization via blocking methods [15]. In the early stages, the basic version MPI-1.0 was originally introduced for distributed memory structures. Later, while evaluating the basic version many modifications were made to enhance the usability of MPI-1.0.

Recently MPI came up with the advanced version MPI-3.1 that include many additional features and functionalities including process groups, process creation & management, environmental management and inquiry, the Info object and point-to-point communication [29, 30].

Throughout the development of HPC, it has been considered as the basic standard for distributing data to multiple nodes and processors. Though MPI designers did not consider the future Exascale systems, it still requires novel MPI configurations and runtime. MPI could be considered a promising model for communicating and passing messages among heterogeneous systems. We have shown the basic structure of how to use the MPI model (see Listing 1).

```
program main
Start
// Starts MPI
 MPI_Init();
// Get no. of processes
 MPI_Comm_size(MPI_COMM_WORLD, size);
// Get current process id
 MPI_Comm_rank(MPI_COMM_WORLD, rank);
// Print Message
 Print ("I am", rank ,"of", size);
// Finalize MPI
 MPI_Finalize ();
 End
```

Listing. 1. Basic Structure of MPI.

B. MPI+OpenMP

The hybrid of MPI and Open Multi-Processing (OMP) is commonly being used for multi-cores distributed homogeneous systems (see Fig. 3) [38]. Data is distributed over multiple nodes and communicated with each other through the MPI scheme. Then it further transferred to the OMP region. OMP determines the available number of threads for that particular node and data written in the OMP region is computed over these threads in shared memory access [16, 17, 18]. MPI is used for inter-node communication to attain coarse grain parallelism whereas OMP is used to achieve fine-grain parallelism over intra-node. In the future, this hybrid approach is one of the promising strategies for dealing with future HPC applications. The latest OMP version is also capable to program GPGPUs for accelerated computing. OMP shared memory multi-processing the programming model is available in C, C++, and FORTRAN for windows and UNIX platforms. This hybrid model is useful for a homogeneous (CPU processors) cluster system [25, 26]. However, it is not preferred for heterogeneous (CPU + GPU) system.

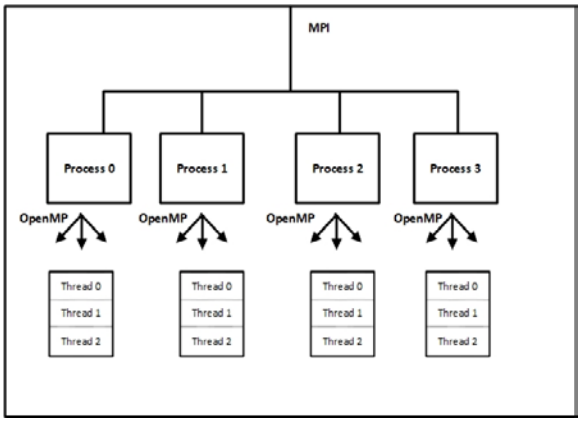

Fig. 3. Processing Mechanism of Hybrid MPI and OMP.

C. Open Computing Language

OpenCL is an efficient parallel programming model for heterogeneous frameworks. OpenCL supports run-time compilation that excludes dependencies on instruction sets, allowing hardware providers to make remarkable changes to instruction sets, drivers, and supporting libraries. It grants portability and compatibility of kernels across multiple hardware and platforms [22, 23]. However, OpenCL requires a complicated setup which includes preparation of settings, command queues, alternatively in addition to a compilation of kernel codes [23].

D. KAUST basic Linear Algebra Subprograms

KBLAS is a diminutive open-source CUDA library that effectively utilizes significant numerical kernels on CUDA-enabled GPUs. It performs a subset of Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) libraries on NVIDIA GPUs [12, 33]. KBLAS presents a subset of approved BLAS functions. It offers remarkable function along with BLAS-like interface that addresses single as well as multi-GPU systems. Using recursive and batch algorithms, KBLAS maximizes the GPU bandwidth, reuses locally cached data and increases device occupancy. KBLAS's ultimate intent is performance. The collection of KBLAS's tuning parameters have a great impact on its performance regarding Compute Unified Device Architecture (CUDA) runtime version and GPU architecture. While the best performance using the default tuning parameters cannot be promised. Such parameters could be easily edited by users on their local systems. It supports compute capabilities 2.0 (Fermi) or higher. KBLAS is written in CUDA C and requires CUDA Toolkit for installation. In the coming future, KBLAS might be exported with auto-tuners.

E. Compute Unified basic Linear Algebra Subprograms

The CUBLAS library has introduced by Nvidia for its CUDA-enabled GPUs. This library is an implementation of BLAS on the CUDA environment that does not provide ease of parallelizing data automatically across multiple devices. However, this library helps the user to gain access to all the resources provided by Nvidia GPU devices [31, 32, 44]. This API provides the facility to speed up the applications either by scaling up and distributing data across multiple GPUs or by processing expensive tasks on a single GPU configuration. For the best utilization of the CUBLAS library, the application needs to reserve the memory in GPU memory space against each dataset, provides the data to memory, calls the CUBLAS functions in a sequence and then needs to move the data from GPU device back to the host memory [37]. It provides some helper function that assists the user in retrieving and writing data on the GPU memory space. CUBLAS can be used for GPU-accelerated algorithms in various sectors such as image analysis, machine learning and high-performance computing.

F. MOC (MPI + OpenMP + CUDA)

To achieve massive parallelism in parallel computing, a Tri-Hierarchy hybrid MOC (MPI + OpenMP + CUDA) model was proposed in 2018 [20]. This model helped in achieving massive performance through monolithic parallelism when we compute any HPC application over a large-scale cluster system having multiple nodes and a number of GPUs > 2. This model was incorporated of MPI, OpenMP and CUDA where MPI is responsible for broadcasting data over distributed nodes, OpenMP is to run received data in parallel on CPU threads, and CUDA is responsible to execute data over accelerated GPU cores, which is the third level of parallelism [24, 27]. MOC provides massive parallelism by achieving tri-level granularity such as coarse, fine, and finer grain parallelism. Although, it is applicable for heterogeneous single and multiple nodes but preferred to use for a large-scale system.

G. Open Accelerators

OpenACC appeared as a high-level programming model that makes use of high-end and supportive directives programs to achieve parallel computing. The main aim of such directories was to minimize the overhead of modifying the source code and hence enabling the portability to a broad field of computing architecture. It allows single source code to run both on CPU and accelerated GPU devices. So, it supports both homogenous and heterogeneous environments. However, OpenACC does not provide flexibility, thread management, thread synchronization, and optimization for the programs being encountered while achieving massive parallelism [21]. A basic structure of OpenACC has been presented (see Listing 2).

```
Start
#pragma acc data
{
#pragma acc parallel loop …
#pragma acc parallel loop
…
}
End
```

Listing. 2. Basic Structure of OpenACC.

H. MPI + CUDA

A dual hierarchical model, the hybrid of MPI and CUDA is considered to be a promising model for node-level and thread-level optimization. However, the use of more resources decreases system efficiency. In this model, an equal number of CPU processors are used as the number of GPU devices connected to the system. The master MPI processor scatters data to all slave processes. These slave processes help for transferring data from CPU cores to GPU devices. Whenever data is transferred to GPU devices, a kernel is invoked where grid and thread block configurations are mentioned to optimize the resources [19]. After GPU processing completion, data is copied back on host CPU cores and utilized [36]. A general data distributing and processing mechanism through a hybrid of MPI+CUDA have been presented (see Fig. 4). Using the MPI-CUDA hybrid approach, we can call multi kernels to compute different accelerated available devices to achieve finer-grain parallelism. This model could be a leading model for future HPC Exascale Computing System (ECS) applications [38].

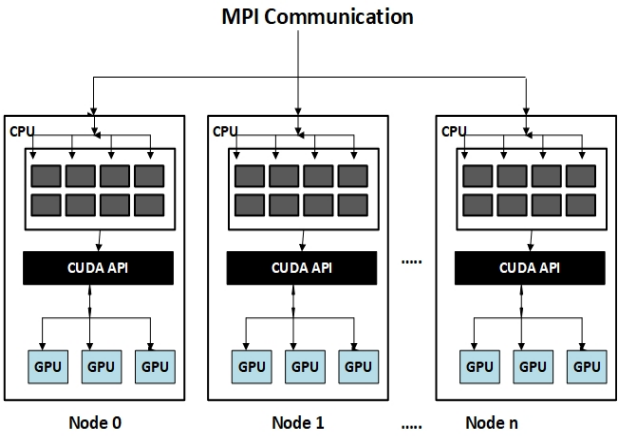

Fig. 4. Processing Mechanism of Hybrid MPI and CUDA.

III. Proposed M2C Model

We have proposed a dual-hybrid model that includes the MVAPICH2+CUDA module (M2C). This massive parallel programming model achieves massive performance by utilizing resources in an efficient manner to reduce power consumption. Following is the elaboration of the proposed M2C model.

A. MVAPICH2

MVAPICH2 is an implementation of the MPI-3.1, which is a thread-safe library developed for high scalability, best performance, and fault tolerance of high-performance computing systems. MPI_Session is a local handle to MPI-3.1 that consumes fewer resources while utilizing the concept of MPI_Group and enabling the scalable communication between different environments [29, 30]. MPI Sessions began as an effort to make aggressive additions/changes to MPI to ensure its success at the Exascale level [38]. It enables better scalability, increases abstraction for better resource isolation, and supports less tightly coupled applications.

It is a new way to initialize (and re-initialize) MPI that uses very few resources by keeping only the state for active communications. It dynamically gathers required data when needed and stores state for the future. It establishes communication relationships/peers before communicating and affect MPI initialization error behavior. The general scheme of MPI_Session has been described (see Fig. 5). MPI Standard 3.1 describes the generic scheme of MPI Session in the following steps [29]:

1) Session initialization and finalization: The compiler creates the session when the constructor for MPI_Session is called and destroys the session when its matching destructor is called. It has elaborated on how these functions initialize and finalize the session (see Listing 3).

```
int MPI_Session_init (
      MPI_Info process_info,
      MPI_Errhandler err,
      MPI_Session *sessionName);

int MPI_Session_Finalize (
MPI_Session *sessionName);
```

Listing. 3. Session Constructor and Destructor.

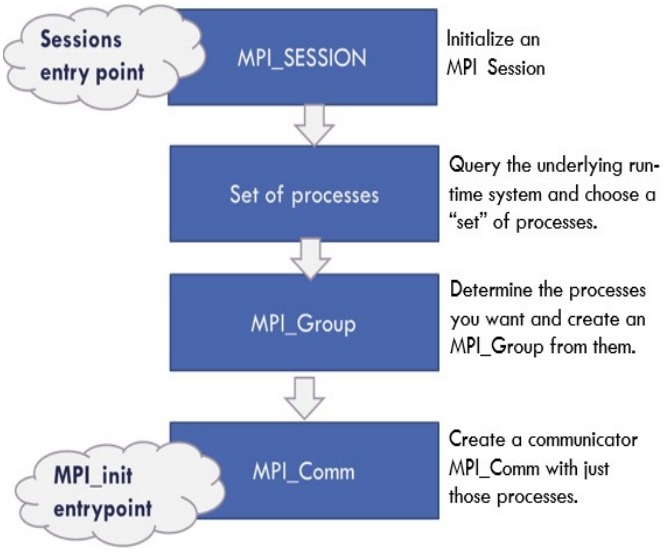

Fig. 5. Overview of the MPI_SESSION.

The working of a new session is initialized whenever MPI_Session_init() function is called by passing certain parameters to it, which returns a valid handle i.e. sessionName of the recently created session. The parameter 'err' is passed to this function which controls the MPI's error handling response during the creation of the session. One more parameter 'process_info' is passed which provides the information to the user that can be helpful for MPI to manage the session's creation. For destroying a session, the function MPI_Session_finalize() is used, which ensures that session no more exists by setting the session handle i.e. sessionName to MPI_SESSION_NULL.

2) Querying runtime system for named sets: MPI processes use the concept of 'named set' for querying the runtime system to retrieve the named sets of processes to utilize them for creating the corresponding MPI_Group against each named set. The function MPI_Session get_names() is used to retrieve all set's names from the runtime has been listed (see Listing 4).

```
int MPI_Session_get_names (
 MPI_Session sessionName,
 char **setName);
```

Listing. 4. Querying Runtime System for Named Sets.

MPI allocates space for the array of strings in the memory (which we termed as 'setName' in Listing 4) when the session is created and freed the space on the session destruction.

3) Getting the size of set from runtime: Information regarding a specific named set is exposed by the runtime using function MPI_Session_get_info() has been prototyped (see Listing 5). It provides an object called 'MPI_Info' which returns the 'size'. The value of this size actually provides the information about each set i.e. the total number of processes in each set. By using this information against each set provided by the runtime, a user can easily decide which groups have to be created for gaining access to the exact required resources.

```
int MPI_Session_get_info (
 MPI_Session sessionName,
 char *setName,
 MPI_Info *size);
```

Listing. 5. Getting Size of Named Set from the Runtime via a Session.

4) Converting set to group: The function MPI_Group_create_session() is used to convert each named set to the group by using the information of each set as elaborated in Step 2 and Step 3 (see Listing 6). This group can then be used for making the communicator for a group of processes. MPI_Group helps to maintain the meta-data of all the available resources stored by the runtime in a scalable manner. If the information of sets stored by runtime internally is scalable, then the internal representation of the corresponding group can also be scalable [28]. Operations that create a group 'non-scalable' must be avoided in order to achieve good scalability.

```
int MPI_Group_create_session (
 MPI_Session session,
 char *name,
 MPI_Group *group);
```

Listing. 6. Converting Named set to Group via a Session.

5) Assigning Communicator to each group: For creating a communicator against each group, the MPI library introduces MPI_Comm_create_group_X() function (see Listing 7). In MPI, to initiate the communication, a parent communicator (MPI_COMM_WORLD) is used generally containing the information of all the processes. This parent communicator consumes countless resources by storing the information of processes even not to be used by the communicator in the communication process. This new concept of creating a communicator via a session provides the facility of assigning a single communicator to each group by providing the same functionality without even using a parent communicator [29].

```
int MPI_Comm_create_group_X (
 MPI_Group myGroup,
 char *tagID,
 MPI_Info size,
 MPI_Errhandler err,
 MPI_Comm *mycomm);
```

Listing. 7. Assigning a Communicator to a Group.

The parameter "myGroup" is passed to the function to provide the specific group of processes to the communicator. The parameter "mycomm" is used to return the new communicator for the targeted group. Other parameters i.e. "size" and "err" are passed to the function that will provide the additional information, which is what is usually not provided by MPI for this communicator creation function. The 'size' parameter is used to query the exact number of processes for which the group has been made and the corresponding communicator is going to be created. The "err" parameter is added to the communicator function that will catch error in case of any failure throughout the session creation.

B. Accelerated Graphics Processing Unit Computation

CUDA is an efficient parallel programming model utilizing accelerated GPUs and threads for massive parallelism [23]. CUDA refers to as the most competent model for thread-level optimization that allows application flexibility by supporting heterogeneous computation where the application uses both the CPU and GPU devices. CUDA parallel programming utilizes the 'CUDA Kernel' to parallelize the data on GPU devices. According to the novel architecture of CUDA GPUs, a block dispatcher has introduced that assign one self-synchronized thread per computational core to schedule the grid. Shared memory is assigned to each block and all the cores within the block can have access to this shared memory [19, 24, 27]. Threads use this shared memory to process the data of a certain block and return the processed data to the GPU block scheduler. The scheduler stores the processed data to GPU global memory space accessible to all the CPU cores of the host. The CPU cores in response read the data from GPU memory and transfer it to CPU cores and then to the main memory. Using this mechanism, CUDA helps to achieve massive parallelism by utilizing both the CPU and GPU devices. The basic structure of using CUDA in C++ has been demonstrated (see Listing 8).

```
Begin
float *a_cpu,*a_gpu;
cudaMalloc ((void **) &a_d, size);
cudaMemcpy (a_gpu, a_cpu, cudaMemcpyHostToDevice);
kernel_function <<< n_blocks, block_size >>> (params);
cudaMemcpy (a_gpu, a_cpu, cudaMemcpyHostToDevice);
cudaFree(a_gpu);
End
        // Cuda kernel to run on GPU
__global__ void kernel_function (params)
        {
        Statements;
        }
```

Listing. 8. Basic Structure of CUDA.

C. M2C Framework

We have shown the architecture of our proposed model M2C (see Fig. 6). Therein, the dual-hierarchy model of MVAPICH2 and CUDA is incorporated. MVAPICH2 is responsible for broadcasting data over distributed nodes while CUDA is responsible to run the code on GPU. The HPC cluster system is made up of multiple racks having thousands of nodes that perform extensive calculations and computations simultaneously.

A detailed workflow of the dual-level hybrid model has been shown in which we have shown the steps of how the proposed model works for a single node in the HPC cluster environment (see Fig. 7). We have considered the DMM application developed in the C++ language to interact with MVAPICH2. The basic functionality of the proposed model is embedded in the concerned user application. This user application receives input data from MPI to solve the extensive computational problem. In the MVAPICH2 communication world, the problem is divided into subproblems by providing coarse-grained parallelism. MPI master process further scatters these subproblems to multiple slave processes and data is transformed from CPU to GPU

environment. GPU environment performs CUDA computation on the received data by invoking the CUDA kernels.

The subproblems are divided into multiple chunks/statements and are parallelized to multiple processors and thousands of cores by providing fine-grained parallelism. The thousands of accelerated cores work simultaneously on a single problem and solve the extensive problem by utilizing both coarse-grained and fine-grained parallelism. Once the data processing is completed, the resultant data is returned in a similar way where processed data is transferred from GPU to CPU cores and so CPU cores return data to the MPI master process. Then the reserved memory for CPUs and GPUs is released.

A quick overview of how MVAPICH2 and CUDA in the M2C model take part to provide massive parallelism. A detailed algorithm for the basic Dense Matrix Multiplication (DMM) application has been presented (see Listing 9). In our proposed model, some significant specifications i.e. the total number of nodes, number of CPUs per node, number of GPU devices per node, number of cores per CPU, and memory levels of the system are obtained in an initial phase. These specifications help the programmer to acknowledge the best resources for the proposed model. However, the rest of the workflow of this proposed algorithm is elaborated as follows:

- Initialize session and getting set names and size of processes (line 2-4): MPI Session in the proposed model is a local handle to MPI-3.1, used to provide the coarse-grained parallelism by passing the data to slave processes. MPI Session query the runtime to obtain the active processes. By using this information provided by the runtime, a user can easily decide which groups of processes have to be created for gaining access to the exact required resources.
- Converting sets to groups, assigning communicators, and getting rank of processes (line 5-7): A communicator is assigned to each group that helps to communicate between the processes. Some additional necessary functions are called that return the information of the sets and groups, size, and rank of the active processes.
- Send and Receive data according to the ranks of processes (line 8-15): Following the ranks, ’0’ ranks are defined as a master rank process and rest of all are considered as slave processes. In the case of master rank, data is broadcasted over all other ranked processes through appropriate methods. Generally, MPI_Send() and MPI_Recv() are used for synchronous (blocking) whereas MPI_Isend and MPI_Irecv() are used for asynchronous (non-blocking) processing. The current study deals with different HPC applications instead of any specific application, in such case, normally programmer is not sure about data dependency then asynchronous processing may not perform better.

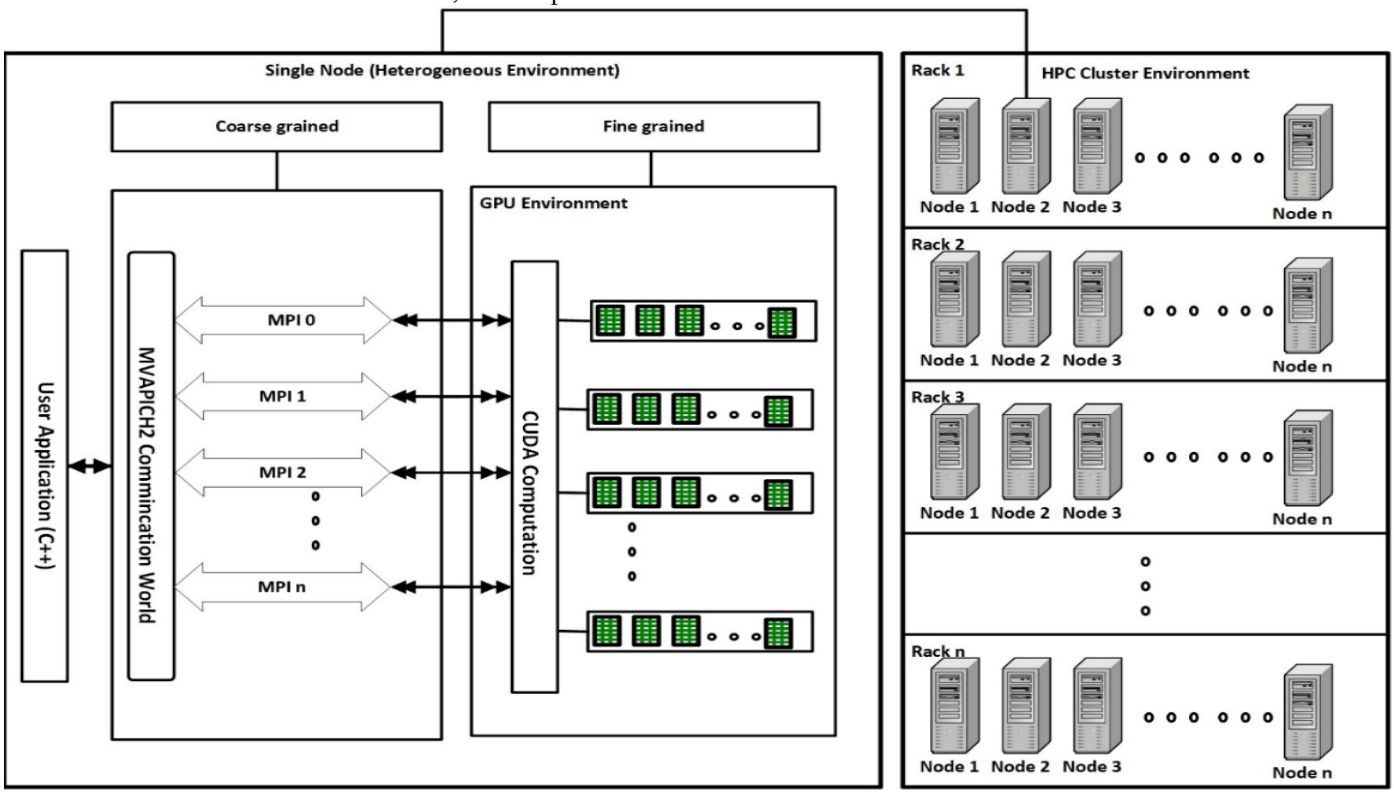

Fig. 6. Architecture of Proposed M2C Model.

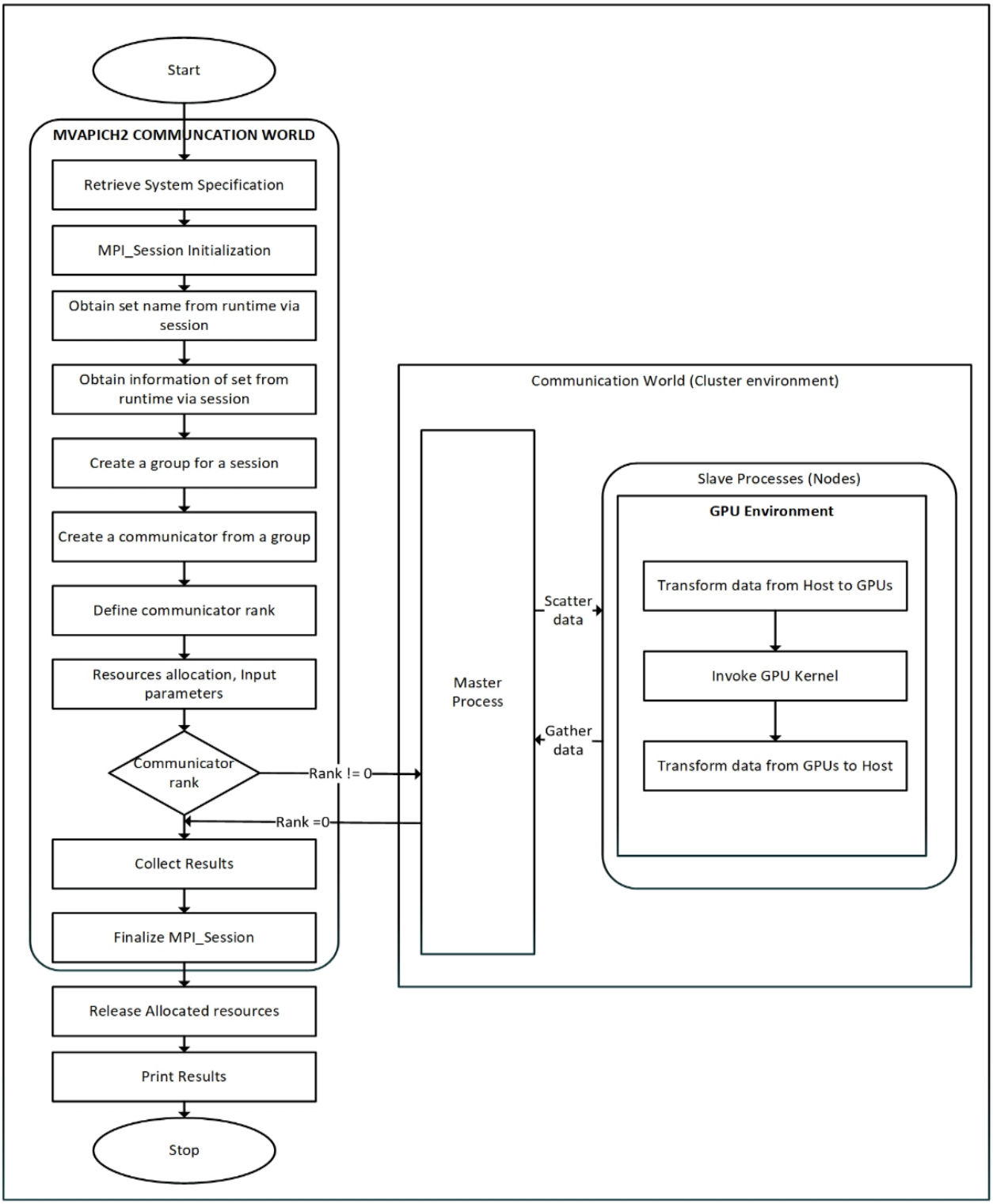

Fig. 7. Workflow of Proposed M2C Model.

```
Input:          mA Input Matrix ←
 mB Input Matrix ←
Output: mC Resulting Matrix ←
Declarations: SessionName, SetName, Size,
GroupName, Comm, Rank, i, numDev, A_D, C_D.
1.      Start MVAPICH2 Parallel Region
2.      SessionName MPI_Session_init()// Initialize MPI session
3.      SetName MPI_Session_get_names()// Group of processes = Set, obtaining set name from runtime via session
4.      Size MPI_Session_get_info()// Get Communication size
5.      GroupName MPI_Group_create_session()// Convert SetName into GroupName.
6.      Comm MPI_Comm_create_group_X()// Group Communicator
7.      Rank MPI_Comm_rank()// Get MPI Ranks
8.      If Rank == Master // define master rank if rank (0)
9.      Make local initialization.
10.     Else
11.     MPI_Isend() // Send data to all processes Rank>0
12.     End If – Else
13.     If Rank > 0 // If rank is slave but not master process
14.     MPI_Irecv() // Receive data from all slave processes
15.     End If
16.     Start CUDA Parallel Region
17.     While (i<numDev) Do // Until Number of GPU devices
    // Assign a device id to specific GPU device
18.      cudaSetdevice(devID)
    // Copy data from host to NIVIDIA GPU Devices
19.      cudaMemcpy() A_D mA ←
20.      InvokeCUDAKernel << grdSize,blkSiz, >>> (A_D, A_B,A_C)
    // Copy data from GPU Devices to host.
21.      cudaMemcpy() mC C_D ←
22.     Free all device variables
23.     End CUDA Parallel Region
24.     // There is MPI master process
25.     If Rank == 0
            // Receive processed data from all rank
26.     MPI_Irecv()
27.     End If
            // Finalize MVAPICH2 processing
28.     MPI_Session_finalize()
29.     End MVAPICH2 Processing/Parallel Region
            // Return the results
30.     Return mC
```

Listing. 9. The Dual-Hybrid M2C Model.

However, we implemented a synchronous (blocking) strategy for all MPI based models in the current study. This data distribution mechanism requires fewer resources (i.e. get the information of only the active processes) and hence consume less power while implementation and provides coarse grain parallelism in the system [48].

Start CUDA parallel region and receive processed data (line 16-20): Once data is distributed over all connected nodes in the targeted system, the data is computed over accelerated GPU devices. For this purpose, CUDA statements are defined and kernels are invoked for GPU computation.

Finalize MPI Session, return results and End MVAPICH2 parallel region (line 26-30): Once the whole data processing is completed, the resultant data is returned in the similar way where processed data is transferred from GPU to CPU cores then CPU cores return data to MPI master process and reserved memory for CPUs and GPUs is released.

IV. Performance Analysis

In this section, we investigate the performance of the proposed M2C system. In this regard, our experiments related suggested dual-level hybrid model M2C were carried out on Aziz-supercomputer manufactured by Fujitsu of HPC Centre of King Abdul-Aziz University, Jeddah, Saudi Arabia [40, 41, 42].

A. Platform of Experiment

The platform of our experiment, Aziz-Fujitsu Primergy CX400 Intel Xeon True-scale QDR supercomputer, was ranked at 360th position in the list of top- 500 supercomputers in 2015 [39]. This Aziz-supercomputer contains 380 thin (regular) and 112 fat (large) computing nodes with a total of 492 computing nodes that interlinked through the InfiniBand within the racks. Recently Aziz supercomputer was upgraded from homogeneous to the heterogeneous system according to the growing need of massively parallel computing by adding two NVIDIA Tesla K20 GPUs based on SMID architecture of 2496 CUDA-cores per device. Two Intel Xeon Phi Coprocessor (MIC) containing 60 integrated cores per processor were also introduced into Aziz-supercomputer to upgrade the system. The heterogeneous system of Aziz supercomputer has total 11904 number of cores [39, 40, 41, 42]. Therein, the memory of a regular computing node and a large node is 96GB and 256GB, respectively. Each node consisting of Intel E5- 2695v2 processor with 12 physical cores with 2.4GHz processing power, which is what is operated by Cent Operating System-v6.4. CUDA toolkit version-9.1 is installed along with other compilers essential for accelerated programming in HPC libraries. MVAPICH2-GDR 2.3.1 is used to enable MPI with the support of accelerated GPU devices. All the accelerated devices and computing nodes of the supercomputer are connected by InfiniBand, User, and Management networks. InfiniBand is used to parallelize the file system, while the user network is used for login and job submission handling. Management and controlling operations are usually supported by the management network only. According to the LINPACK benchmark and theoretical peak performance, the performance of Aziz supercomputer was marked as 211.3 TFlops/sec and 228.5 TFlops/sec, respectively.

B. HPC Metrics

We have taken different metrics including total time for execution, total number of Flops measured, energy efficiency, and overall power consumption of the system. These metrics can be categorized into the following two fundamentals metrics.

1) Performance Measurement: The overall performance of HPC system is considered as the most fundamental and essential metric in massive parallel programming which is measured in a total number of achieved floating-point operations per second (Flops) [32]. The total number of achieved Flops (F_T) can be calculated by dividing the achieved Flops calculated at the peak performance of the system (F_{pp}) by the total execution time (T_{Exec}) [20], which can be written as follows:

$$F_T = \frac{F_{pp}}{T_{Exec}} \quad (1)$$

We have calculated the number of Flops (F_T) based on execution time by implementing the M2C model on DMM application against varying datasets by considering the peak performance of Aziz-supercomputer i.e. 211.3 TFlops /s, [47].

2) Power Measurement: In emerging supercomputers, energy efficiency with less power consumption is of interest [6, 8, 46]. In this regard, we have used well-known software applications of Open Hardware Monitor [34] and GPU-Z.2.6.0 [35] for the measurement of CPU and GPU temperature and power consumption, respectively. We have shown the running states of Open Hardware Monitor and GPU-Z.2.6.0 (see Appendix 1 and Appendix 2, respectively). The total energy consumed by a system Esys can be calculated by integrating the energy consumption composed of memory contention, bandwidth, parallelism and behavior of the application [20] as follows:

$$E_{sys} = \int_0^t memC\ (dt) + bandW\ (dt) + Parll\ (dt) + Bhv\ (dt) \quad (2)$$

As discussed earlier, the future Exascale supercomputing system will be a heterogeneous architectural system. This is how necessary to emphasize primarily on power consumption for both homogeneous and heterogeneous architectural based systems. The power consumption in a heterogeneous system is considered into three major parts including power consumption by host CPU processors, power consumption by memory operations (inter-memory and intra-memory) and power consumption by accelerated GPU devices. Power consumption for the systems having GPU installed on it is calculated by [20] as follows,

$$P_{sys}(w) = \sum_{i=1}^{N} P_{GPU}^{i}\ (w^i) + P_{CPU}\ \sum_{j}^{M}(w^j) + P_{main}(w) \quad (3)$$

where, P_{cpu}, P_{gpu}, P_{sys}, P_{main} indicate power of the CPU, GPU, system and motherboard, respectively. N and M denote the number of GPUs and the total CPU threads in the system, respectively. w_i and w_j account for the workload of GPU i and CPU j, respectively.

The power consumption on a specific application (P_{app}) of the system can be calculated as follows [20],

$$P_{app}(w) = \sum_{i=1}^{N_{app}} P_{GPU}^{i}\ (w^i) + P_{CPU}\ \sum_{j}^{M}(w^j) + P_{main}\left(w_{app}\right) \quad (4)$$

Where P_{app} is proportional to the workload of the system.

V. Results and Discussion

We have investigated our M2C model by implementing it into the Dense Matrix Multiplication (DMM) application on Aziz-supercomputer for emerging Exascale systems that provide tremendous performance through massive parallelism and limiting the overall power consumption. Therein, two fundamental HPC metrics i.e. performance and power consumption were observed during the experiments. All results have been executed using four kernels for GPU processing of supercomputers. The quantified metrics were

observed for different matrix sizes. The performance in linear DMM application was observed against different datasets by determining the floating-point operations in GFlops/sec while the power consumption was calculated for different matrix sizes in GFlops/ Watt. We quantified different metrics including performance and power consumption. We measured the fundamental parameters including time execution, speedup, and number of flops whereas power consumption related metrics were including as power consumption and temperature.

We have implemented three well-known libraries named MOC, kBLAS, and cuBLAS [49, 50] in DMM application for the same HPC metrics taken on Aziz-supercomputer. For datasets 1000-10000, kBLAS and cuBLAS were able to achieve their peak performances as 830 and 690 GFlops/sec, respectively. We observe that MOC outperformed all the three implemented models by delivering peak performance up to 1TFlops. The results against each dataset for MOC, kBLAS, and cuBLAS were compared with that of the proposed M2C model.

Our proposed model outperforms all the other implemented models for datasets 1000-10000 by delivering peak performance even more than that of the MOC model with 1.27 TFlops (see Fig. 8).

Similarly, the energy efficiency by all of the implemented algorithms was observed, therein cuBLAS, kBLAS and MOC achieved 5.2, 5.6 and 6 number of GFlops/Watt for small matrix size, respectively. For the same small matrix size, the proposed M2C model attains 7.099 number of GFlops/Watt. It is noticed that cuBLAS, kBLAS and MOC models attain the maximum number of GFlops/Watt as 6.2, 6.5 and 8, respectively. On the other hand, M2C delivered 8 GFlops/Watt in initial matrix size and by increasing the matrix size the energy efficiency also increased and reached up to a maximum of 10.38 GFlops/Watt for the large matrix size (see Fig. 9).

We observe that our M2C model delivered better performance in 1Watt compared to that of the other implemented models. In M2C the framework, the idea of MPI_Session and MPI_Group has worked well for limiting power consumption. MPI_Session a local handle of MPI-3.1 i.e. MVAPICH2 uses very few resources by keeping only the state for active communications. So, it helps to reduce communication in the heterogeneous system. This parent communicator consumes countless resources by storing the information of processes even not to be used by the communicator in the communication process. The new concept of creating a communicator via a session provides the facility of assigning a single communicator to each group (created via MPI_Group) to provide the same functionality without even using a parent communicator.

MVAPICH2 by consuming fewer resources and reducing communication over-head has helped us to limit the power consumption and provided remarkable performance in 1Watt compared to that in kBLAS, cuBLAS, and MOC models. This could be helpful to notice that performance and power consumption are directly related to each other. However, there exists a trade-off between these two metrics which can be determined as follows:

$$\frac{Performance}{Power} = \frac{Execution\ within\ the\ time\ unit}{Energy\ during\ the\ execution\ time\ unit} \quad (5)$$

where the rate of change of performance under specific power consumption measured in Glops and GFlops/Watt, respectively.

The comparative analysis of tradeoff for cuBLAS, kBLAS, MOC and M2C models (see Fig. 10). The vertical and horizontal lines respectively represent performance and power consumption. We can notice that cuBLAS, kBLAS and MOC libraries performed well in the DMM algorithm. However, the proposed model M2C outperformed all the implemented models by accomplishing 1278 GFlops and by consuming 123W total power during larger matrix multiplication.

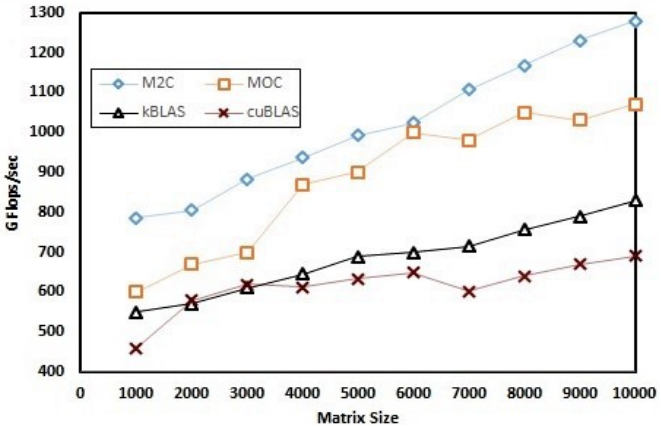

Fig. 8. M2C Performance Comparison with MOC, kBLAS and cuBLAS.

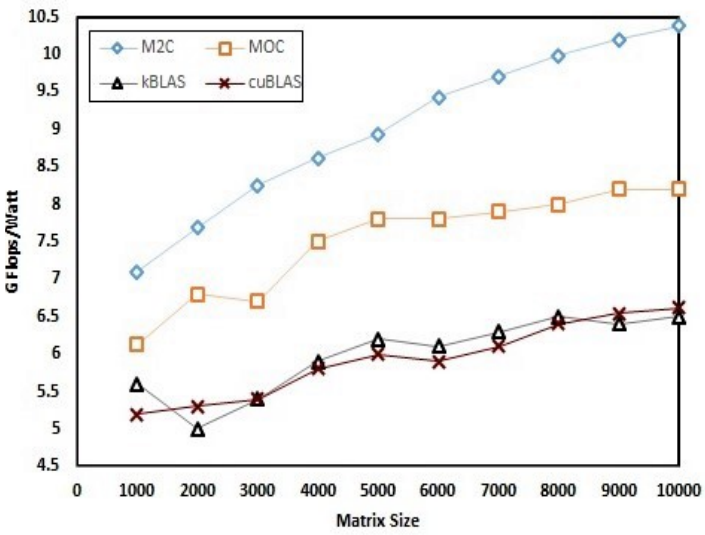

Fig. 9. Energy Efficiency in DMM for M2C vs (MOC, kBLAS & cuBLAS).

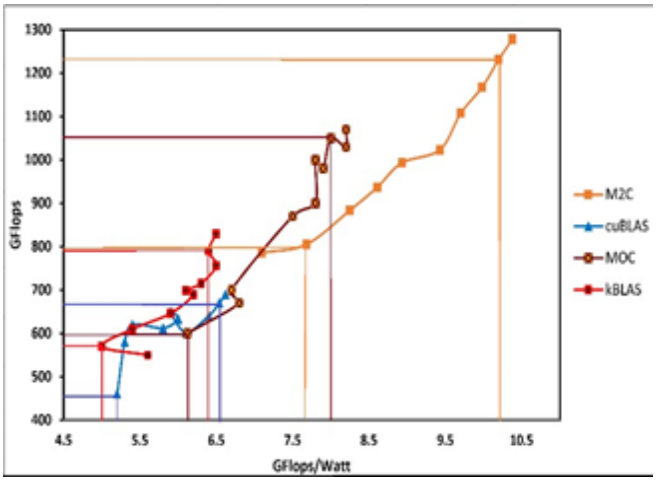

Fig. 10. Performance-Energy Efficiency Tradeoff.

Hence, our proposed system could be envisioned as a promising model for achieving massive performance in emerging Exascale systems. In this regard, the most challenging step towards Exascale computing systems is that it does not exist yet. However, the performance of HPC systems has been improved based on the current results and predictions to achieve Exascale performance. The fundamental challenges mentioned in this paper would require targeted investments to acquire Exascale computing. The primary goal of Exascale computing systems is to handle massive data HPC applications. Performance and power consumption are the two primary HPC metrics that have been taken into consideration the most challenging factors for Exascale computing systems. These challenges have a direct relationship with the number of resources. An increase in the number of resources consequently enhance the performance in the system and so increase the power consumption as well.

VI. Conclusion

Towards the race of achieving Exascale performance, power consumption has been the most significantly constrained resource among all the others. Therefore, achieving practical Exascale computing with optimum performance is of interest. In this regard, an advanced computing system can deliver a thousand-fold performance improvement compared to the current Petascale computing. However, new system-wide methodologies and methods for power monitoring and administration are somewhat necessary. Novel programming models and programming methodologies are undergone rapid growth, but the quest for enhanced programming models always exists. There are significant questions and research regarding the models that will be used at the Exascale level to achieve better performance than the current Petascale systems. Contributing to the quest for the optimum programming model for Exascale systems, a novel parallel programming model named M2C has been proposed. The proposed model has been evaluated by implementing linear DMM application on heterogeneous architecture and the results have been compared with well-known libraries such as MOC, kBLAS, and cuBLAS. M2C results using 4 kernels for GPU processing outperformed all the other implemented models while achieving 1278 GFlops by consuming 123W total power. Nevertheless, the proposed M2C model could be a promising and leading model for emerging Exascale systems.

Acknowledgments

This project was funded by the Deanship of Scientific Research (DSR) at King Abdul Aziz University, Jeddah, under grant no. (D-154-611-1440). The authors, therefore, acknowledge with thanks DSR technical and financial support.

References

[1] Perarnau, Swann, Rinku Gupta, Pete Beckman: ‘Argo: An Exascale Operating System and Runtime’, 2015.

[2] Shalf, Dosanj, Morrison: ‘Exascale computing technology challenges.’ International Conference on High Performance Computing for Computational Science 2010, pp. 1-25.

[3] Reed, Dongarra: ‘Exascale computing and big data.’ Communications of the ACM 2015, vol. 58, no. 7, pp. 56-68.

[4] Zhou, Min: ‘Petascale adaptive computational fluid dynamics’, Diss. RENS- Selaer Polytechnic Institute, 2009.

[5] Dongarra, Walker: ‘The quest for petascale computing.’ Computing in Science & Engineering 2001, vol. 3, no. 3, pp. 32-39.

[6] Reed, Berzins, Pennington, Sarkar, Taylor: ‘Report: Exascale Computing Initiative Review’. ASCAC 2015.

[7] Gropp, Snir: ‘Programming for Exascale Computers.’ Computing in Science & Engineering 2013, vol. 15, no. 6, pp. 27-35.

[8] Ashraf, Muhammad Usman, Fathy Alboraei Eassa, and Aiiad Ahmad Albeshri. "Massive Parallel Computational Model for Heterogeneous Exascale Computing System." 2017 9th IEEE-GCC Conference and Exhibition (GCCCE). IEEE, 2017.

[9] Ashraf, Muhammad Usman, et al. "Toward exascale computing systems: An energy efficient massive parallel computational model." International Journal of Advanced Computer Science and Applications 9.2 (2018).

[10] Rajovic, Vilanova: ‘The low power architecture approach towards Exascale computing.’ Journal of Computational Science 2013, vol. 4, no. 6, pp. 439-443.

[11] Kogge, Shalf: ‘Exascale Computing Trends: Adjusting to the “New Normal” for Computer Architecture.’ Computing in Science & Engineering 2013, vol. 15, no. 6, pp. 16-26, 2013. Available: 10.1109/mcse.2013.95.

[12] Workshop on programming abstractions for data locality, PADAL 2015, Available online: https://sites.google.com/a/lbl.gov/padal workshop/,2015.

[13] Ashraf, Muhammad Usman, et al. "Empirical investigation: performance and power-consumption based dual-level model for exascale computing systems." IET Software (2020).

[14] Message Passing Interface (MPI). Available Online: https:// computing.llnl.gov/tutorials/mpi/. [Accessed: 10- Nov- 2019].

[15] Dinan, Balaji: ‘An implementation and evaluation of the MPI 3.0 one-sided communication interface.’ Concurrency and Computation: Practice and Experience 2016, vol. 28, no. 17, pp. 4385-4404, 2016. Available: 10.1002/cpe.3758.

[16] Shuangshuang Jin, Chassin: ‘Thread Group Multithreading: Accelerating the Computation of an Agent-Based Power System Modeling and Simulation Tool -- C GridLAB-D’. Hawaii International Conference on System Sciences, 2014.

[17] Martineau, McIntosh-Smith: ‘Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model.’ IEEE (IPDPSW), 2016. Available: 10.1109/ipdpsw.2016.70.

[18] Artur Podobas, Sven Karlsson: ‘Towards Unifying OpenMP Under the Task-Parallel Paradigm, International Workshop on OpenMP 2016’. Springer International Publishing.

[19] CUDA Toolkit 10.1 Update 2 Download, NVIDIA Developer. Available Online: https://developer.nvidia.com/cuda-downloads. [Accessed: 11-Nov- 2019].

[20] Muhammad Usman, Alburaei, Ahmad: 'Toward Exascale Computing Systems- An Energy Efficient Massive Parallel Computational Model'. IJACSA 2018, vol. 9, no. 2. Available: 10.14569/ijacsa.2018.090217.

[21] Alsubhi, K., et al. "A Tool for Translating Sequential Source Code to Parallel Code Written in C++ and OpenACC." 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA). IEEE, 2019.

[22] Khronos OpenCL Working Group, The OpenCL Specification Version 1.2, November 2011. [Online]. Available: http://www.khronos.org/.

[23] NVIDIA Corporation, OpenCL Best Practices Guide, 2011.

[24] Ong, Weldon: 'Acceleration of large-scale FDTD simulations on high performance GPU clusters.' IEEE Antennas and Propagation Society International Symposium, 2009. Available: 10.1109/aps.2009.5171722 [Accessed 11 November 2019].

[25] Jin, Jespersen: 'High performance computing using MPI and OpenMP on multi-core parallel systems.' Parallel Computing 2011, no. 9, pp. 562-575.

[26] Mininni, Rosenberg: 'A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence.' Parallel Computing 2011, no. 6-7, pp. 316-326.

[27] Langer, Totoni: 'Energy-efficient computing for HPC workloads on heterogeneous many-core chips', Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '15, 2015. Available: 10.1145/2712386.2712396 [Accessed 11 November 2019].

[28] Chai, Hartono: 'Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters'. IEEE International Conference on Cluster Computing, 2006. Available: 10.1109/clustr.2006.311850 [Accessed 11 November 2019].

[29] MPI Standard 3.1. Available Online: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, pp. 585-597.

[30] MPI-2 Journal of Development. Available Online: http://www.mpi-forum.org/docs/mpi- jd/mpi-20-jod.ps.Z.

[31] CUBLAS. Available Online: https://developer.nvidia.com/cublas, Sep 2017 [Dec 11, 2017]

[32] NVIDIA Accelerated Computing "developer.nvidia.com/cuda-downloads", 02 Nov 2016

[33] Lilja, David: 'Measuring computer performance: A practitioner's guide.' Cambridge university press, 2005.

[34] Pawliczek, Dzwinel: 'Visual exploration of data by using multidimensional scaling on multicore CPU, GPU, and MPI cluster.' Concurrency and Computation: Practice and Experience 26, 2014, no. 3, pp. 662-682.

[35] Satish, Harris: 'Designing efficient sorting algorithms for manycore GPUs. IEEE International Symposium on Parallel & Distributed Processing'. 2009. Available: 10.1109/ipdps.2009.5161005 [Accessed 11 November 2019].

[36] Agostini, Rossetti: 'Offloading Communication Control Logic in GPU Accelerated Applications'. 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017. Available: 10.1109/ccgrid.2017.29 [Accessed 11 November 2019].

[37] CUDA Samples. Available Online: http://docs.nvidia.com/cuda/cuda-samples/index.htmlsimple-cublas.

[38] Muhammad Usman, Amna Arshad, Rabia Aslam: 'Improving Performance In Hpc System Under Power Consumptions Limitations', IJARS 2019, Volume. 10, No. 2, pp. 75-84, 2019.

[39] Chrysos, George: Intel Xeon Phi™ coprocessor-the architecture. Intel Whitepaper 176, 2014.

[40] Muhammad Ashraf, Alburaei Eassa: 'Performance and Power Efficient Massive Parallel Computational Model for HPC Heterogeneous Exascale Systems.' EEE Access 2018, vol. 6, pp. 23095-23107. Available: 10.1109/access.2018.2823299.

[41] King Abdul-Aziz University. Available Online: https://www.top500.org/site/50585.

[42] Fujitsu to Provide High-Performance Computing and Services Solution to King Abdul-Aziz University. Available Online: http://www.fujitsu.com/global/about/resources/news/press-releases/2014/0922-01.html.

[43] Amarasinghe. ASCR programming challenges for Exascale computing, Rep. Workshop Exascale Program. Challenges, 2011.

[44] CUDA Toolkit 4.0. Available Online: https://developer.nvidia.com/cuda-toolkit-40.

[45] ASCAC Subcommittee for the Top Ten Exascale Research Challenges, U.S. Dept. Energy State, Washington, DC, USA, 2014.

[46] Hennecke, Michael: 'Measuring power consumption on IBM Blue Gene/P.' Computer Science-Research and Development 27, 2012, no. 4, pp. 329-336.

[47] Leung. Ed. Handbook of Scheduling: 'Algorithms, Models, and Performance Analysis.' CRC Press, 2004.

[48] Ashraf, M. Usman, Fathy Alboraei Eassa, and Aiiad Ahmad Albeshri. "High performance 2-D Laplace equation solver through massive hybrid parallelism." 2017 8th International Conference on Information Technology (ICIT). IEEE, 2017.

[49] Gallivan, Plemmons: 'Parallel Algorithms for Dense Linear Algebra Computations.' SIAM Review 1990. vol. 32, no. 1, pp. 54-135. Available: 10.1137/1032002.

[50] Anzt, Haugen: 'Experiences in auto-tuning matrix multiplication for energy minimization on GPUs.' Concurrency and Computation - Practice and Experience 2015. vol. 27, no. 17, pp. 5096-5113. Available: 10.1002/cpe.3516.

[51] Ashraf, Muhammad Usman, Fathy Alboraei Eassa, and Aiiad Ahmad Albeshri. "Efficient Execution of Smart City's Assets Through a Massive Parallel Computational Model." International Conference on Smart Cities, Infrastructure, Technologies and Applications. Springer, Cham, 2017.