Extending Shared-Memory Computations to Multiple Distributed Nodes

Waseem Ahmed
Department of Computer Science
Faculty of Computing and Information Technology
King Abdulaziz University, Jeddah, Saudi Arabia

***Abstract*—With the emergence of accelerators like GPUs, MICs and FPGAs, the availability of domain specific libraries (like MKL) and the ease of parallelization associated with CUDA and OpenMP based shared-memory programming, node-based parallelization has recently become a popular choice among developers in the field of scientific computing. This is evident from the large volume of recently published work in various domains of scientific computing, where shared-memory programming and accelerators have been used to accelerate applications. Although these approaches are suitable for small problem-sizes, there are issues that need to be addressed for them to be applicable to larger input domains. Firstly, the primary focus of these works has been to accelerate the core kernel; acceleration of input/output operations is seldom considered. Many operations in scientific computing operate on large matrices - both sparse and dense - that are read from and written to external files. These input-output operations present themselves as bottlenecks and significantly effect the overall application time. Secondly, node-based parallelization limits a developer from distributing the computation beyond a single node without him having to learn an additional programming paradigm like MPI. Thirdly, the problem size that can be effectively handled by a node is limited by the memory of the node and accelerator. In this paper, an Asynchronous Multi-node Execution (AMNE) approach is presented that uses a unique combination of the shared-file system and pseudo-replication to extend node-based algorithms to a distributed multiple node implementation with minimal changes to the original node-based code. We demonstrate this approach by applying it to GEMM, a popular kernel in dense linear algebra and show that the presented methodology significantly advances the state of art in the field of parallelization and scientific computing.**

***Keywords*—GPU; OpenMP; shared memory programming; distributed programming; CUDA**

I. Introduction

Many applications in engineering and scientific computing involve operations on dense and sparse matrices [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. At the core of many of these applications is the General Matrix-Matrix multiplication (GEMM) operation, which is regarded as one of the most widely used high performance kernels. GEMM is also used in graph theory and in the design of recommender systems and machine learning algorithms like page rank and logistic regression. It is used by many kernels in dense linear algebra (DLA) and has been part of many fast DLA libraries [12], [13], [14].

It is generally described by the operation $C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$, where *A*, *B* and *C* are matrices and α and β are scalar constants between 0 and 1. The *GEMM* kernel is composed of three basic matrix-based operations - matrix-matrix multiplication, matrix-matrix addition and scalar-matrix multiplication. Among these three operations, the matrix-matrix multiplication makes GEMM computationally expensive. Moreover, when large dense matrices are involved, the GEMM operation, in addition to being computationally expensive, becomes I/O intensive as well and making the operation holistically efficient becomes more challenging.

Although this operation could naively be implemented using a simple three-nested loop with algorithmic complexity in $O(n^3)$, it has been the subject of a lot of research over the last few decades. Indeed, areas related to the optimization of this operation and its parallelization using improvised algorithms and software paradigms and hardware architectures are still active areas of research [12], [9], [15], [3], [16], [17], [18], [19], [20]. More recently, with the advent of easily available yet powerful workstations equipped with sophisticated co-processors, researchers have parallelized GEMM on these single-node workstations using accelerators like Graphics Processing Units (GPUs), Many Integrated Cores (MICs) and Field Programmable Gate Arrays (FPGAs) [19], [21], [12], [17].

When parallelizing GEMM on a single node, shared memory programming paradigms like OpenMP and heterogeneous computing paradigms like CUDA, OpenACC and OpenCL have been the dominant choice among developers. But as scientists continue to expand their problem domain, either to investigate larger problem sizes or to obtain finer results, developers will face three limitations when developing applications for a single-node execution. Firstly, is the cap on the maximum speedup that can be theoretically achieved on a single node that is a function of the core/compute element per node. Secondly, as the size of matrices grow (>$10^5 \times 10^5$), they become too large to be accommodated in the limited memory of a single node with a moderately sized RAM. For example, a single dense matrix of size $50k \times 50k$ used to store elements of double floating-point precision will require approximately 18.6GB of RAM. This makes the GEMM operation for these large matrices non-trivial to implement on a node with 32GB of memory or a high-end GPU with 16GB of device memory. Thus, when dealing with large matrices, parallelization on a single node will not yield the desired speedup because of memory constraints. Thirdly, the I/O operations that deal with reading these matrices from files into memory and writing them back from memory to file will become prohibitively expensive and will become the main bottleneck in the GEMM

operation.

One way to address this problem is to distribute the computation to multiple nodes. But this will require the developer to port the existing code written for a single-node to a distributed architecture using a distributed-memory paradigm like MPI. Moreover, this exercise is not trivial - the process of rewriting and porting scientific applications from one paradigm (or architecture) to another has a prohibitively expensive cost [22]. Besides the learning curve associated with the learning of a new programming paradigm, testing and debugging will be an extra burden that the developer will have to bear, in addition to addressing problems of load balancing and optimization *[9]*. Also, with the proliferation of platforms, architectures and programming paradigms in the HPC arena, it is indeed challenging for an application developer to command sufficient expertise to fully exploit the unique architectural features of multiple platforms and different programming paradigms.

Thus, as researchers continue to expand their problem domains to larger ones, a more holistic approach will be needed to address these categories of applications. This paper presents Asynchronous Multi-node Execution (AMNE) that attempts to address the afore-mentioned challenges associated with parallelizing a class of applications known as embarrassingly-parallel applications [23].

The rest of the paper is organized as follows: the next section presents the motivation for this approach. Section 3 describes the AMNE approach that forms the core of this paper. Pseudo-replication using the launcher script is explained in the section following Section 4. Evaluation and results of applying the approach to parallelize GEMM are presented next followed by the related work section. Conclusion is presented in the last section.

II. Motivation

Consider a GEMM operation on a set of matrices. The total time taken to execute it sequentially on a single node comprises the time costs of both its core computation and I/O where the I/O costs could be disk-access related or network related. This can be expressed as

$$t_{seq} = t_i + t_{op} + t_o \tag{1}$$

where t_i represents the time taken to read the input matrices *A* and *B* from file into memory, t_{op} represents the time taken to execute the core GEMM operation on matrices *A* and *B* to produce a resultant matrix *C*, and t_o represents the time taken to write matrix *C* from memory to file. Various approaches have been proposed in literature to reduce t_{op}, ranging from scalar optimizations like loop transformations [24], [25] and algorithmic replacement [26], [27] to parallelization of the core operation on multiple compute elements [18]. Applying any of these strategies will significantly reduce t_{op}. However, there is limited work in literature that provides strategies or methods to reduce t_i and t_{op} on a single node or on multiple nodes; details on explicitly reducing t_i or t_o are generally absent in work related to parallelization of GEMM.

So, if we assume t_i and t_o remain unchanged after the optimization/parallelization process, the speedup obtained on a single node ($S_{n=1}$) can be expressed as

$$S_{n=1} = t_{seq}/t_{par} = t_{seq}/(t_i + t_{op_{n=1}} + t_o) \tag{2}$$

where $t_{op_{n=1}} \leq t_{op}$ is the time taken to execute the optimized or parallelized core operation (kernel) on a single node. Theoretically, if $t_{n=1} \rightarrow 0$, $S_{n=1}$ reduces to

$$S_{n=1} = 1 + t_{op}/(t_i + t_o) \tag{3}$$

From this it can be seen that to obtain higher speedups, the second term in Eq. 3 needs to be maximized. But for large matrices, both t_i and t_o can become very large. For example, it takes about ~2745 secs to read a matrix of size $50k \times 50k$ from file to memory and to write it back from memory to file. This implies that to obtain higher speedups, a dedicated effort needs to be made to reduce both t_i and t_o. However, because of hardware limitations, reduction of t_i and t_o is indeed challenging on a single node regardless of which programming paradigm is used.

The primary focus of this paper is to help developers in addressing this challenge (to reduce t_i and t_o) while easily extending their single-node code implementation to a multiple-node implementation with minimal code change and with a zero or low learning curve for the developer. The next sections further elaborate this strategy.

III. Asynchronous Multi-node Execution

The approach described in this paper, referred to as asynchronous multi-node execution (AMNE), seeks to address the two limitations of single-node execution in handling large matrix operations highlighted in the previous sections, namely, 1) to overcome the memory constraint of a single-node, 2) to reduce t_i and t_o and 3) to have minimal code changes tot he original single-node code when doing so. To address the first challenge, the computation is sub-divided or *sliced* into smaller sub-problems and the computation distributed over multi-nodes. This is a classical approach and is a well researched and documented strategy adopted by various researchers working with MPI and other distributed computing paradigms. However, unlike a master-slave relation that exists between processes in a distributed-computing programming paradigm like MPI, the launched instances in AMNE are independent of each other and no communication or synchronization exists between them. This is applicable to any embarrassingly-parallel application implemented using the AMNE approach.

Slices, as defined in this approach are unlike *tiles* and *blocks* used in the *tiling* and *batching* approach done at the node or GPU level. In the latter, partial (intermediate) results of the resultant matrices are generated by multiple compute elements and has to be aggregated with other partial results to obtain a *block(s)* of the final resultant matrices. In AMNE, on the other hand, the generation of a *slice* (or a partial-slice) of the resultant matrices is done by a single compute element. Additionally, in tiling and blocking, the tiles and blocks are in-memory representations of the matrices. A *slice*, on the other hand, is an out-of memory representation.

These multiple slices are operated on by independent, albeit similar, instances that are launched on multiple dedicated nodes in the cluster. To enable this, the single-node code

is to be transformed such that it can be controlled at run time to operate on any particular slice. Once that is done, a *launcher* script is used to initiate an asynchronous multi-node execution (AMNE) of the code instances, where the independent instances are asynchronously launched on multiple nodes in the cluster to operate on their designated *slices*.

These steps are further elaborated in the following sub-sections:

A. Slicing

Deciding the number of *slices* needed for a given application and defining their sizes and shapes, is a prerequisite for applying AMNE. In case of a homogeneous cluster, the slice sizes could be uniform. For example, consider a 20-node homogeneous cluster with each node having 32GB of memory. Also, consider the GEMM operation to be performed on this cluster for a configuration of $(80k \times 40k \times 80k)$. For this GEMM configuration, matrix A of size $80k \times 40k$ and matrix B of size $40k \times 80k$ need to be loaded into memory from file and elements of resultant matrix C of size $80k \times 80k$ need to be generated in memory. Assuming a double takes 8 bytes for storage, matrices A and B require $\sim 23.8GB$ memory each and matrix C requires $\sim 47.63GB$, a total of $\sim 95.27GB$. Clearly, it cannot be performed efficiently in a single pass on a single-node because of memory constraints of the node. To solve the problem in one pass, the GEMM computations are distributed across the 20 nodes in the cluster, with each node assigned a block(s) of matrix *C* for independent calculation. To enable this, both matrices *A* and *B* need to be sliced; these slices of matrices *A* and *B* are complete, in the sense that they can independently generate a complete *block(s)* of the resultant matrix C.

To decide the slice dimensions of matrices *A* and *B*, consider a slice of matrix A with dimensions $(l \times m)$ and a slice of matrix B with dimensions $(m \times n)$, to generate a block of resultant matrix C of dimensions $(l \times n)$. The choice of the variables *l* and *n* should be such that they satisfy the following relation

$$f(l,m,n) = [(l \times m) + (m \times n) + (l \times n)] \times 8] \leq R \times 2^{30} \quad (4)$$

where *R* refers to the node memory in GBytes. This simplifies to $f(l, 40k, n) \leq 4295$ for the example above where $m = 40k$ and memory, $R = 32GB$. Uniformly distributing the computation load on the cluster of 20-nodes, we get values $l = 4$ and $n = 4$ that also satisfy Eq. 4. These values of *l* and *n* yield dimensions of $(4k \times 40k)$, $(40k \times 4k)$ and $(4k \times 4k)$ for a slice of matrix *A*, a slice of matrix *B* and a block of resultant matrix C, respectively.

Slicing, in the case of a heterogeneous cluster, is more intense and depends on the memory constraints of each node or device on the node, in case an accelerator (GPU/MIC) is being used on the node. For example, a quad-socket multi-core node with 32GB RAM and an accelerator (GPU or MIC) with 16GB of device memory may be assigned a bigger slice (s_{gpu}) when compared to a slice (s_n) allocated to a single-socket dual-core node with no accelerator and with 4GB of RAM. In this case, the slices sizes will be considerable different with $s_{gpu} \gg s_n$.

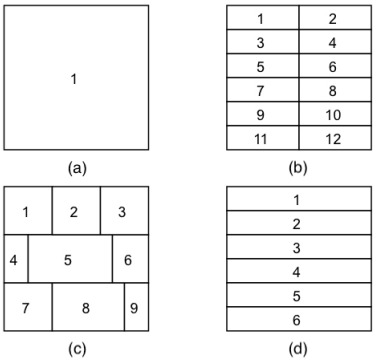


Fig. 1. Domain Decomposition

One way to obtain slices is to fix a value for *l*, and heuristically obtain values of *n* based on experience or based on empirically obtained node peak performance values. This could be repeated for other values of *x* such that optimum performance is obtained.

There are two extreme scenarios when solving for appropriate values of *l* and *n* in Eq. 4.

1) If dimensions of the matrices are very large or the cluster size is small or the memory/node is small, then Eq. 4 may not be satisfied. This implies that a full slice of either matrix *A* or matrix *B* or slices of both matrices cannot be accommodated in the node memory along with a block of matrix *C*. In this scenario, a slice may need to be read piece-wise (tiled) into memory by a node; this will mean that two or more passes will be required to calculate each block of matrix *C* on a node.
2) If the matrices are small or the cluster size is big, then uniformly distributing the GEMM computation on all the nodes may not result in the best speedup. In this scenario, it would be better to limit the computation to fewer than the maximum nodes in the cluster. This is further explained later in this section.

There is no single one-size-fits-all strategy for deciding a slice size and shape and how they should be distributed among the nodes. The slices can be assigned to nodes in one of five ways as shown in Fig. 1 and as described below

1) One slice assigned to only one node - This is the case where the entire matrix *C* (as a single slice) is calculated by a single node as illustrated in Fig. 1(a). This is, also, the default method adopted by single-node developers using OpenMP, CUDA, OpenACC or OpenCL
2) Same-sized slices assigned to all nodes - In a cluster of *n* nodes, the calculation of matrix *C* is divided uniformly among *n* nodes. For example, in a cluster of $n = 12$ nodes, the calculation of matrix C is uniformly divided among all 12 nodes in the cluster, with each node calculating only 1/12th portion of matrix C. One slicing option that satisfies Eq. 4 is shown in Fig. 1(b)

3) Different-sized slices assigned to all nodes - In a heterogenous cluster of *n* nodes, calculation of matrix C is non-uniformly divided among the nodes in the cluster. This is shown in Fig. 1(c) where nine non-uniform slices are divided among $n = 9$ nodes in a cluster
4) Optimal distribution - Small GEMMs cannot fully exploit parallelism. Thus, in a homogeneous cluster, for relative smaller matrices, slices are assigned to only some nodes in a cluster such that nodes reach their peak capacity and the total time to execute (t_{max}) is minimized. For example, a tile size of $5120 \times 5120 \times 5120$ achieves 93% of peak performance (15TFlops) on an NVIDIA Volta 100 GPU [21]. In this option, some nodes in the cluster are left unassigned. This is illustrated in Fig. 1(d) where only six nodes of the cluster ($n > 6$) compute their assigned slices of matrix C and the other nodes are left unassigned
5) Non-uniform optimal distribution - This is a combination of the previous two cases where non-uniform slices of matrix C are assigned to only some nodes in the cluster such that all nodes reach their peak capacity and the total time to execute (t_{max}) is minimized. Some nodes in the cluster are left unassigned. This is illustrated in Fig. 1(c) where only nine nodes of the cluster ($n > 9$) compute their assigned slices of matrix C and other other nodes in the cluster are left unassigned.

B. Out-of-Memory Slicing

Traditionally, *tiles* represented in the tiling and blocking approaches have been in-memory representations of matrix partitions. Slices, as described in this paper, on the other hand, are out-of-memory representations of a matrix partition. They are defined externally and on a *shared-file system*.

In a *shared-file system* present in most HPC clusters, every node in the cluster has access to a common file system. All nodes in such clusters see a single, unified file system regardless of whether the file system physically resides on a single node or is spread over thousands of individual storage servers. An example of one such shared file system is *Lustre*, which was developed for extreme-scale compute clusters [28], [29], and is currently deployed in more than 60% of the top 100 supercomputers *[30]*. Some features of *Lustre* that are directly relevant to this paper are mentioned below:

1) It can support tens of thousands of client systems with petabytes (PB) of storage and handle hundreds of gigabytes per second (GB/sec) as I/O throughput
2) All clients in the cluster see a *single, coherent, synchronized namespace* [31]; the file system support concurrent read/write by multiple clients with data coherency maintained between all clients by the Lustre distributed lock manager (LDLM) [31].
3) Files on Lustre are stored as one or more objects, with each object stored on a separate Object Storage Target (OST); this allows several clients to write to different parts of the same file simultaneously while allowing other clients to read from the file
4) Lustre supports file striping, where a file may be stored on multiple (≤ 2000) OSTs . This considerably increases the I/O bandwidth (aggregate) by a factor of almost 2000 for a single file when accessed by multiple nodes.

In the approach presented in this paper, all files that contain the input data (matrices *A* and *B*) are not stored locally but on such a shared-file system. Similarly, the output data (matrix *C*) is written to a file on the shared-file system. This implies that every node in the cluster can independently and simultaneously read the input data without having it to be explicitly broadcasted to the slave-nodes by a master-node as is the case in many MPI-based approaches. Also, the distributed storage of a file as multiple OSTs and the file striping feature of the shared-file system makes access to the files much faster when simultaneously performed by the nodes.

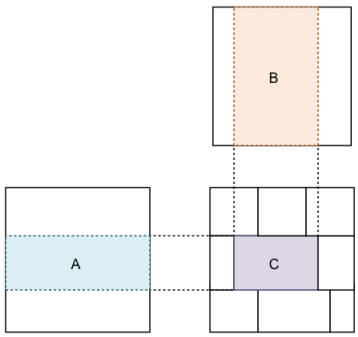


Fig. 2. Calculation of a Block of Matrix C

Once slices have been defined, the developer chooses the storage format in one of two ways

1) All slices of the input matrices stored in separate single files on the shared-file system. In the case of GEMM, this means all elements of matrix A are stored in a single file. And all elements of matrix B stored in a separate single file.
2) Each slice of the input matrices stored as a separate file on the shared-file system. In the case of GEMM, this means that if matrix *A* has been divided into twenty slices, there will be twenty separate files used to store elements of matrix *A*, with each file corresponding to elements of a particular slice.

Once slices have been stored as files in either of the two ways described above, they need to be accessed by their respective nodes. For example, consider a GEMM operation illustrated in Fig. 2 where a node has been assigned a complete block of C (shaded) for calculation. This node requires only the elements in the shaded slice of matrix *A* and the elements in shaded slice of matrix *B* to calculate elements in the shaded block of *C*, as indicated in the figure. The other nodes may also access the same slices of matrices *A* and *B* for their calculations, but these accesses are independent of each other as mentioned previously in this section.

These simultaneous access allowed by the shared-file system to the input data by the nodes in the cluster,

considerably brings down the sequential access times, t_i and t_o in Eq. 3. Rewriting Eq. 3 for multiple nodes ($n > 1$) we get

$$S_{n>1} = t_{seq}/t_{par} = t_{seq}/(t_{i_{n>1}} + t_{op_{n>1}} + t_{o_{n>1}}) \quad (5)$$

It was empirically observed that $t_{i_{n>1}} \rightarrow t_i/n$ and $t_{o_{n>1}} \rightarrow t_o/n$ for various values of slice dimensions and sizes. Theoretically, for an embarrassingly parallel application, and for an infinite cluster size, when $t_{op_{n>1}} \rightarrow 0$ then $S_{n>1} \rightarrow n$ and attains a close-to-peak linear speedup.

C. Code Modification

The main objective of this step is to ensure that code designed specifically for a single-node execution is extended to one that can be executed on a distributed-memory cluster.

Based on decisions taken in the previous step, each node will need to calculate only its specific portion of the slice. Code will, thus, need to be modified such it can be executed in a *slice*-specific fashion on a node in the cluster. When coding for a heterogeneous implementation, there is an additional challenge at this stage as multiple versions of the code need to be explicitly managed. For example, CPU-only code and code for a CPU+GPU execution will be different and have to be handled differently when being compiled and executed by the *launcher* script (described later). This could be managed in one of two ways:

1) same code for all nodes (controlled using pre-processor directives and conditional compilation to differentiate the CPU-only and CPU+GPU codes);
2) different code for different nodes (by explicitly separating the CPU-only and CPU+GPU codes).

Both options are supported in this approach and are managed by the job script (described later).

The first objective in this step is to ensure that all operation-specific variables are made amenable to modification at run time and are not hard-coded. This modification of variables at run-time could be done either through a configuration file or using command-line parameters. These variables, referred to as run-time parameters (RTP) in the rest of this paper, ensure that the same code can be used to operate on different slices and that it's execution behaves in a pure SPMD fashion. The second step is to ensure that all files needed for the read and write operations, including configuration files, are stored on the shared-file system.

To illustrate, consider a GEMM operation coded for a single-node execution using OpenMP, OpenACC or CUDA. In the original single-node implementation of the code, the dimensions of the matrices *A*, *B* and *C,* i.e. $(l \times m), (m \times n)$ and $(l \times n)$, respectively, and names of filenames that store data for these matrices may be hard-coded and may not be amenable to modification at run-time. In this case, code is modified such that these variables can be modified at run time as a RTP. The RTP in this case are the three-tuple $< l, m, n >$ and the full path-names for matrices *A*, *B* and *C*. Also, if the files that contain the elements of matrices *A* and *B* are stored locally on the node, they need to be moved to the shared-file system. Similarly, the file that is generated for matrix *C*, also has to be placed on the shared-file system.

IV. Job Execution and Launch

A. Job Script Preparation

Once the original single-node code has been modified in the previous steps, it needs to be compiled on the node that it needs to be executed on and then launched as an executable on that node. This is done using a job script specifically prepared for each defined *block* of matrix *C*. The responsibility of the job script is to load the modules, libraries and the environment necessary for successfully compiling the code and executing it on the node. For example, code based on CUDA and cuBLAS needs to be compiled on a node in the cluster that has one or more Nvidia GPUs installed on it. Prior to compilation, the appropriate compilers and libraries, in this case, the appropriate version of the *nvcc* compiler and the cuBLAS libraries, need to be loaded into the environment on the node where the code is to be compiled. After creating the required environment, the job script has to compile the code using the specified compiler flags and links to libraries. If the compilation is successful, the executable is to be launched on that node with it's *node*-specific RTP supplied as command-line arguments or through a configuration file. Each *block* of the output has its own job script. These job scripts can either be manually created or automatically generated if the slices are large in number.

The collection of these job scripts is then submitted to the cluster's job scheduler by a *launcher* script, which queues the jobs based on the job's priority and the requested type and number of nodes in the cluster; the jobs are launched as and when the requested nodes becomes available.

The application is said to have completed execution when all the jobs in the collection have successfully terminated.

B. Pseudo-Replication

AMNE results in a *pseudo-replication* of the instances on the cluster. In pseudo-replication, concurrent instances of an executable are asynchronously launched on multiple nodes in the cluster with each instance executing to create a distinct portion of output. Although the AMNE behavior may initially appear to be similar to *task replication* used in distributed systems [32], [33], [34], [35], it differs in many aspects as described below:

1) In task replication, all replicated instances of the executable are identical and they work on the same data. The job scheduler in the cluster launches the same executable multiple times. In pseudo-replication, on the other hand, the replicated instances work on different data. The job script is different for each instance and is dynamically generated depending on the size of the cluster and other parameters.
2) In task replication, only the result from the first successful execution is used; all other executing instances of the executable are either explicitly terminated or ignored. This results in poor and inefficient resource utilization as the outputs from the (other) slower replicated instances are completely

ignored. In pseudo-replication, results generated by all the replicated instances are used and no launched instance is prematurely terminated. This results in better resource utilization of the cluster resources.

3) The primary objective of task replication is to address latency in nodes, to improve quality of service or to improve application fault tolerance [34]. Parallelization of tasks is not an objective. On the other hand, in pseudo-replication, parallelization of tasks is the primary objective and is explicitly desired.
4) In task replication, the total execution time of an application is calculated as the time taken by the fastest successfully completed instance ($t = min(t_1, t_2, \ldots, t_n)$, where t_i denotes the time take by node i to complete a task). On the other hand, the total execution time of an application in pseudo-replication is calculated as the time taken by the slowest successfully completed instance ($t = max(t_1, t_2, \ldots, t_n)$).

Pseudo-replication also differs from an MPI-based execution. In applications coded using MPI, identical multiple instances are concurrently launched on nodes in the cluster using the *mpirun* command from the shell. These multiple instances work together as one coherent entity (based on a master-slave model) with active message-passing based communications between them. The instances in pseudo replication, on the other hand, are launched independently using a launcher script and work in the form of a loose asynchronous fork-join model. The instances are completely independent of each other and no form of direct or indirect communication exists between them.

V. Evaluation and Results

In this section, the proposed approach is evaluated for a GEMM operation on large matrices and results presented.

A. Experimental Testbed

All experiments for this work were carried out on the *Aziz* supercomputer at King AbdulAziz University. *Aziz* was ranked 359th in the top500 list in the June 2015 ranking with a peak performance of 228.6TFlops [36].

The Aziz cluster has 380 standard compute nodes with 96GB of memory and 112 high memory compute nodes with 256GB of memory. All nodes have a dual-socket Intel Xeon E5-2695 12-core processor running at 2.40 GHz. Nodes with GPU-based accelerators have 96GB of RAM and are equipped with Nvidia's Tesla-based K20 GPUs, which consists of 2496 CUDA cores and 5GB of device RAM.

The approach was evaluated on two types of nodes of Aziz - The first set consisted of nodes with 256GB of RAM with no accelerators, and the second set consisted of nodes with 96GB of RAM with K20 GPUs. These two sets are referred to as *setA* and *setB*, respectively, in the rest of this section.

Code was written in C/C++ and used Intel's MKL and NVidia's cuBLAS libraries. Code that used Intel's MKL libraries was compiled using Intel's *icc* version 17.0.2 and CUDA code was compiled using NVidia's CUDA 9.2. Jobs on *Aziz* were scheduled using the PBS Pro job 12.2.0 scheduler [37].

TABLE I. Size of MTX files on disk

	M	N	Size of file
1	10000	10000	1.8 GB
2	20000	20000	7.5 GB
3	30000	30000	17.0 GB
4	40000	40000	31.0 GB
5	50000	50000	48.0 GB
6	100k	100k	84GB

B. Benchmarks

Although the approach is applicable to any embarrassingly parallel application, the GEMM operation is specifically used here to illustrate and evaluate the approach. The GEMM operation, described by $C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$, is an operation on two input matrices *A* and *B* to produce a resultant matrix *C*. The values one and zero and were used for the scalars α and β, respectively.

All matrices used in this work were generated as double-precision elements as files on disks in the MTX format, the format which is used for storing large sparse matrices in the SuiteSparse Matrix Collection [38]. These MTX files are very large even for a single large matrix. Table I serves as an approximate indicator to the size of files when floating-point value matrices of size ($M \times N$) are stored in the MTX file format. It is apparent from the size of the files, that operations involving large matrices cannot be easily executed in one-pass on nodes with moderately sized RAMs or on GPU-based accelerators with small device memory.

For experiments on setA, matrices that corresponded to a GEMM configuration of $(50k \times 50k \times 50k)$ were used and for experiments on setB, matrices that corresponded to a GEMM configuration of $(10k \times 10k \times 10k)$ were used.

All file were stored on Aziz on the Fujitsu Exabyte File System (FEFS) which is a scalable parallel file system based on Lustre and designed for Fujitsu HPC clusters [39]. Elements of matrix *A* and matrix C were stored in the row-major order and elements of matrix *B* were stored in column-major order to enable easier slicing.

C. Slicing

For each experiment, matrices *A* and *B* were divided into *slices* based on the number of nodes and the type of nodes available in the cluster. Since the type of nodes in both sets was homogeneous and the matrices used in the GEMM operation were sufficiently large, the number of slices (s) was set equal to the number of nodes.

The *shapes* (dimensions) of the slices, however, can be decided in various ways. For example, a matrix of size $40k \times 40k$ could be sliced across eight homogeneous nodes in various shapes - as eight equal-sized slices of $20k \times 10k$ elements or eight equal-sized slices of $5k \times 40k$ elements, etc. based on the memory configuration of the node and other criteria. On a heterogeneous cluster, the *slice* sizes and shapes assigned to nodes could be different.

The largest slice used for a single-node implementation on setA was for GEMM configuration $(50k \times 50k \times 50k)$. For setB, the slice size was limited to a GEMM configuration $(10k \times$

TABLE II. SLICES USED FOR EXPERIMENTS ON SETA

No.	number of slices	shape <l,m,n>
1	1	<50k,50k,50k>
2	2	<25k,50k,50k
3	5	<10k,50k,50k>
4	10	<5k,50k,50k>
5	20	<2.5k,50k,50k>
6	50	<1k,50k,50k>

TABLE III. SLICES USED FOR EXPERIMENTS ON SETB

No.	number of slices	shape <l,m,n>
1	1	<10k,10k,10k>
2	2	<5k,10k,10k>
3	4	<2.5k,10k,10k>
4	5	<2k,10k,10k>
5	10	<1k,10k,10k>
6	20	<0.5k,10k,10k>

$10k \times 10k$) for a single-node implementation to ensure slices for both matrices *A* and *B* simultaneously fit in the memory of the GPU (NVidia K20). The shapes of slices used in this work for experiments on setA and setB are given in Tables II and III, respectively.

D. Code Modification

Two different node-based code versions were used to implement the *GEMM* operation - one was based on Intel's MKL [14] and the other was based on CUDA and its cuBLAS libraries [13]. Both code versions consisted of four main code sections, that performed the following four main operations:

1) Allocation of memory for the matrices on the host and/or device.
2) Reading the input matrices A and B from files into (host and/or device) memory.
3) Execution of the core GEMM (kernel) operation on matrices *A* and *B* to produce matrix *C*.
4) Writing the resultant matrix C from (device and/or host) memory onto to files.

Both Intel's MKL library and NVidia's cuBLAS provide single-node implementations for GEMM that is optimized for their respective target platforms. They are *cblas_dgemm()* and *cublasDgemm(),* respectively, for matrices with double-precision elements. *cblas_dgemm()* expects the two input matrices *A* and *B* to be provided in the row-major format and produces matrix *C*, also in the row-major format. On the other hand, cublasDgemm() expects the two input matrices *A* and *B* to be provided in the column-major format and produces matrix *C*, also in the column-major format. Thus, the second section in code, responsible for reading data from files on disk into memory, in their respective implementations, had to be slightly modified to accommodate for this slight variation. Similarly was the case for the last section in code responsible for writing the resultant matrix to the file.

There were six run-time parameters (RTP) required for the GEMM implementation - three configuration parameters l, *m* and n, which represented the dimensions $l \times m$, $m \times n$ and $l \times n$ of the slices of matrices A, B and C, respectively and three file names with the full pathnames where the slices/blocks of matrices *A*, *B* and *C* are or will be stored on the FEFS shared-file system. The filenames associated with blocks of the resultant matrix *C* are kept unique to enable a block to be differentiated from other blocks of matrix *C*. All relevant sections in code were modified to accept these six RTP from the command-line instead of a configuration file. The third section in code responsible for executing the kernel operation consisted of the GEMM function call in the library. The function calls in both versions of code (MKL and cuBLAS) were modified to reflect the RTPs.

In general, all hard-coded references to *L*, *M* and *N* in both versions of code were changed, where needed, to reflect ones provided as RTP at run time from the command line.

E. Launcher and Job Scripts

The Aziz cluster consists of nodes with different configurations and each of this configurations has a unique queue associated with it. Thus, a job placed on the queue of the PBS scheduler [37] may need to be configured individually. Additionally, as each *block* generation is unique, it is controlled through a unique job script. For each defined *block* of matrix *C*, the following details have to be specified in the job script for the node responsible for generating the *block*.

1) Node environment required for compilation (PBS directives, compiler version and libraries to load, number of threads, etc.)
2) Command for compilation (compiler path, optimization flags, linker options, etc. specific to the slice).
3) Absolute pathname of files on the FEFS shared-file system where the node-specific slices need to be read from or written to.
4) Command for execution along with the node-specific RTP.

For a homogeneous cluster, these details will be similar for all slices. For a heterogenous cluster, the details will vary depending on the node configuration on which the block is to be generated. The job details can either be supplied by an external configuration file or hard coded into the launcher script. The launcher script then loops through the job details for each slice, dynamically generates a job script and submits the jobs collectively to the PBS job queue for execution on the cluster. The algorithm for the *launcher* script is described in Algorithm listing 1. It basically consists of a loop which is executed *n* times, with each iteration creating a node-specific job script (*job.i*) based on the node configuration and block allocated to it. After the job script has been generated for the node, it is then submitted to the appropriate queue on Aziz using the *qsub* command of PBS. The launcher script terminates once all jobs have been successfully submitted to the PBS queue, which then schedules them based on the priority of the jobs and the availability of resources on Aziz. It should be noted that in any multi-user cluster environment, the availability of nodes is largely non-deterministic and influences all approaches similarly, and is not considered as a parameter in evaluating the efficacy of any approach or algorithm in a multi-user distributed-computing environment.

Algorithm 1 Launcher script

```
struct block {
    rtp   // run-time Parameters
    env   // compiler environment
    compile // compile command
    exec // launch command
    queue    // Q name to submit on Aziz
}

def prepare_job(i, b):
    s = get_env(b.env)
    s += compile_cmd(b.compile)
    s += exec_cmd(b.exec, b.rtp)

    write_file(jobfile.i, s)
end_def

b[] = init(N) // init block details
for i in N
do
    prepare_job(i, b[i])
    submit_q(jobfile.i, b[i].queue)
done
```

F. Results and Analysis

Experiments to evaluate the proposed approach were done on both setA and setB by varying *n*, the number of nodes in the virtual cluster. Slices were uniformly distributed between the nodes in both the experiments. Readings in seconds were recorded using the *omp_get_wtime()* function for multiple runs and different *slice* size combinations as tabulated in Tables II and III. Due to the similarity that exists between single-node instance execution times on the cluster for AMNE, multiple readings for single-node instance executions were recorded on a single node for all slice and cluster sizes ($n > 1$).

Individual times were recorded for the following: 1) read operation, i.e. to read a slice of matrix A and a slice of matrix B from the shared-file system into memory; 2) GEMM operation on slices of matrices *A* and *B* to produce a block of matrix *C,* and 3) write operation, i.e. to write the slice of matrix *C* from memory to the shared-file system. When a GPU was used, transfer times for moving data between host and device and back were included in the GEMM operation time and not in the read and write. Timings for the experiments were recorded as $t = max(t_1, t_2, ...t_n)$ which represented the time taken to perform the GEMM operation by the slowest node in the virtual cluster on its allocated slices where t_i is the time taken in seconds to execute the operation on node *i,* and $\forall i, 1 \leq i \leq n$.

The read and write operations in both the experiments were performed by a single thread on the node and these operations were not parallelized at the node-level. All slices of matrices A and B, required for the experiments, were stored as separate files on the shared-file system. Blocks of resultant matrix C were also generated as separate files on the shared-file system by the multiple instances of the AMNE. Slices in all cases were stored in the MTX format, as stated earlier.

Timings for the read, write and the core GEMM operations were recorded separately and plotted as shown in Fig. 3. The total time represented in the figure is the sum of the time taken for the read, write and the core GEMM operation. Also, all readings in the figure represent readings recorded with minimal change to the original code; no extra node-based optimizations were performed to the code in any of the experiments.

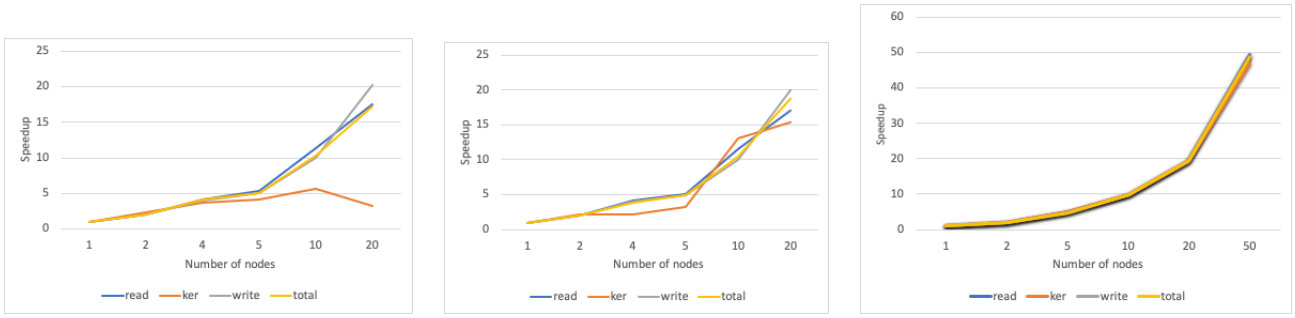
Fig. 3(a) shows the readings for experiments on setB using a K20 GPU for different six different values of *n*, the virtual cluster size for a GEMM configuration of $10k \times 10k \times 10k$. The size of the configuration was chosen so as to comfortably enable a single-pass execution when *n=1*. The code was compiled using CUDA 9.0 and used the *cublasDgemm()* function from the cuBLAS library. Fig. 3(b) shows the readings for experiments on setB but without using any accelerator. Six different values of *n*, were used in this case as well for a GEMM configuration of $(10k \times 10k \times 10k)$. The code in this case was compiled using *icc* and used the *cblas_dgemm()* function from Intel's MKL. Fig. 3(c) shows the readings for experiments on setA without using any accelerator. Six different values of *n*, were used in this case as well for a GEMM configuration of $(50k \times 50k \times 50k)$. The code in this case was compiled using *icc* and used the *cblas_dgemm()* function from Intel's MKL. A GPU was not used in the experiments on setA as the matrix size was too large to fit in the device memory without making a substantial change to the original code/algorithm and node-based optimization was not an intent of this research.

Speedups for all operations with $n > 1$ were calculated relative to the timings obtained for a single-node execution ($n = 1$). It can be observed from the Figure, that the speedup for both the read and write operations varies almost linearly with the increase in the number of nodes. However, for the core GEMM operation, a small dip was observed when the sizes of the matrices became smaller. In the case of the GPU this was expected as the data transfer time (communication) between device and host start to dominate the computation time. This has also been documented in [21] which notes that the efficiency of the Dense Linear Algebra (DLA) libraries for smaller sizes of matrices is not as high as for larger matrix sizes. Thus, having slices that lie beyond the dip would not be profitable. This gives a clear indication that the optimal size of slices has to be decided, either based on developer experience or using results from prior auto-tuner software runs for the application. However, in this experiment, as the time taken to execute the core operation was negligible when compared to the time taken to read and write data, the total speedup for the operation was not affected much and showed an almost linear speedup as well. No such dip was observed in readings for the experiment on setA as the slice sizes were sufficiently large to enable optimized parallelization.

For these GEMM configurations, node-scalability performance was observed to be almost linear with *n.* Extrapolating from the graphs in the figure it can stated that $S_{n>1} \rightarrow n$ in Equation 5.

G. Comparison with other Approaches

Since this work is a multi-node implementation, comparisons with any single-node approaches would be



(a) (b) (c)

Fig. 3. Speedups obtained by Asynchronous Batching on Cluster Nodes

unfair as the speedup obtained using this approach will always be much higher for large matrices. Also, the size of matrices that can be handled on single-nodes in a single-pass is constrained by node-memory. On the other hand, using the presented approach, matrices of very large sizes can be handled comfortably in a single-pass on reasonably large clusters.

When compared to other multi-node methods, this approach performs equally well if not better. This is because in most MPI-based and distributed-computing approaches, reading data from files is mainly done by a single process [15]. Although the overall operation time could be reduced by overlapping communication and computation using clever heuristics [40], [41], it is still an extra exercise. Else, dealing with large matrices would mean that the speedup would be constrained by the large values of t_i and t_o in Equation 5. Also, the criss-cross message passing between processes to aggregate the partial (intermediate) results, creates an increase in the network activity on the cluster. The approach proposed in this paper has an added advantage of simplicity and increased developer productivity over MPI-based approaches, as coding, testing, debugging and workload balancing using CUDA or OpenMP single-nodes are relatively simpler operations than their equivalents on traditional distributed-memory clusters using MPI [42], [10].

VI. Related Work

The optimal distribution of computations across compute elements to reduce an application's overall execution time is a decades-old, well-researched and well-documented problem [43], [44], [45]. A few common approaches used by researchers when dealing with large matrices in applications is to change the mathematical formulation of the operation[46], [20], to use secondary memory [18], [20], to pipeline the operations [13], to operate on data in multiple passes, or to distribute these operations over compute elements either on a single node or across multiple nodes.

The approach adopted on distributed platforms, is generally based on a master-slave process model where blocks of data are broadcast by a node to other nodes in the cluster [15]. Another popular approach is to use *tiling* followed by *batching*, where *tiling* refers to the partitioning of the matrices into tiny blocks or tiles, while *batching* refers to the assignment of these tiles to threads or computing elements for computation. *Batching* has been used on GPUs in combination with tiling to parallelize GEMM [12], [21]. Here, the GEMM operation is broken down into smaller GEMMs, the computation of which is then distributed among the available compute elements on a single-node (CPU or GPU). The compute elements then generate partial results of the resultant matrix *C,* which are then combined to obtain the final resultant matrix. Here, blocks are generally of uniform size and are in-memory representation of the matrices. Also, data reuse is an important concern in efficient batching as memory is shared between computing elements. In AMNE on the other hand, *slicing* (a combination of tiling and batching) is an out-of-memory operation and data reuse at the inter-node cluster level is not a concern. Also, in these approaches, the reduction of the time involved in I/O when dealing with large matrices is not explicitly addressed.

The number of nodes in an exascale system is expected to grow to more than 50 times of what it was in 2010 [47]. When programming using OpenMP or CUDA, parallelism is restricted to the node on which the program executes and cannot be extended beyond that node. Also, it becomes increasingly difficult for DLA libraries like PLASMA and MAGMA to handle large-scale matrix multiplication on a single-node due to hardware resource limitations [15]. To extend parallelism beyond a single node, a pure distributed programming paradigm like MPI could be used. But some researchers prefer a hybrid combination over a pure MPI implementation [48] and there have been many who have used multiple programming paradigms on heterogeneous platforms in various combinations [4], [11], [49], [48], [8], [10], [22], [50], [3].

However, for a developer who is only proficient in using single-node-based programming models like OpenMP or CUDA, learning to code, test and debug using an additional distributed memory paradigm, like MPI, could be a big challenge. On the other hand, not utilizing the potential parallelism inherent in a large distributed cluster, also, is not desirable. But to efficiently combine programming paradigms like MPI and OpenMP/CUDA requires the developer to have intricate knowledge of the multiple hardware architectures and programming paradigms to fully exploit the combined platform's capability. For example, besides the knowledge about the number and type of streaming processors (SM) and cores on the GPU, the size and memory hierarchy on the device, registers per thread, the CUDA version that it supports and its compute-capability, support for unified memory, the

optimal dimensions for the grid and block combination to use, etc. which are required to achieve optimized performance on the GPU, a developer will be required to know the network-topology of the cluster, communication modes between processes, sophisticated algorithms that relate to efficient overlapping of computation and communication, [40], [41], etc. when using MPI. This leads to lower developer productivity.

To handle large matrices by overcoming memory-constraints of a single-node or a GPU, out-of core computations [51], [1], [7], [9] have been used which seek to decompose a matrix into smaller pieces and operate on them in multiple passes; data resides in disk and has to be explicitly moved in and out of memory for the passes. While this addresses the problem of handling large matrices on memory-constrained single-nodes and accelerators, the achievable speedup is constrained by the sequential passes that the algorithm has to make.

VII. Conclusion

Single-node programming paradigms like OpenMP, CUDA and OpenACC have recently gained popularity among researchers in Engineering and scientific computing and optimized libraries like Intel's MKL, NVidia's cuBLAS, and PLASMA are available to them that have been optimized for execution on single-nodes. However, as HPC moves to Exascale, and the number of nodes available in a cluster is set to dramatically increase, researchers will continue to expand their problem domains, either to investigate larger problem sizes or to obtain finer results. These domain sizes will be much larger than can be comfortably handled on resource constrained single-nodes. In such cases, efficient ways of handling the large I/O associated with these large applications and distributing the computations beyond a single-node and across multiple nodes will be needed. For the latter, porting single-node implementations to multiple-nodes will be needed which is a large and non-trivial exercise and will require time and developer expertise to fully exploit the unique architectural features of multiple platforms and different programming paradigms.

In this paper, an Asynchronous Multi-node Execution (AMNE) approach was proposed that works in combination with existing node-based optimizations while addressing the afore-mentioned challenges. AMNE combines shared-file storage commonly available in HPC clusters, pseudo-replication and out-of-memory matrix slicing in a unique way, which reduces the I/O time considerably with minimally-required code changes which positively influence developer productivity. AMNE was evaluated for the GEMM operation on large matrices. Matrices, stored as files on the shared-file system, were partitioned into slices at the node-level and slices allocated to nodes such that a block of the resultant matrix could be independently calculated by a node. The calculations were then independently run in an AMNE fashion to generate the final resultant matrix. Different combinations of nodes were used with different *slice* shapes and sizes. Results showed that the speedup obtained on multi-nodes using AMNE for the overall GEMM operation, including both I/O and computation times, over single-node execution time was almost linearly scalable with the number of nodes allocated to it in the cluster.

Although the approach was evaluated for GEMM on large matrices in this paper, AMNE can be easily extended to any HPC application that falls in the category of embarrassingly-parallel applications.

Acknowledgment

This work was supported by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah under grant number DF-403-611-1441 . The authors, therefore, gratefully acknowledge the DSR technical and financial support. All experiments for this work were conducted on the Aziz supercomputer managed by the HPC Center at the King Abdulaziz University.

References

[1] M. G. Awan, F. Saeed, and F. Saeed, "An out-of-core GPU based dimensionality reduction algorithm for big mass spectrometry data and its application in bottom-up proteomics," in *in Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, August 2017.

[2] P. Czarnul, "Parallelization of large vector similarity computations in a hybrid CPU+GPU environment," *Journal of Supercomputing*, pp. 768–786, 2018.

[3] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, "GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems," *International Journal of Parallel Programming*, vol. 45, no. 5, pp. 1046–1072, 2017.

[4] V. Lončar, L. E. Young-S., S. Škrbić, P. Muruganandam, S. K. Adhikari, and A. Balaž, "OpenMP, OpenMP/MPI, and CUDA/MPI C programs for solving the time-dependent dipolar Gross-Pitaevskii equation," *Computer Physics Communications*, vol. 209, pp. 190 – 196, 2016.

[5] F. Rabbi, C. Daley, H. Aktulga, and N. Wright, "Evaluation of directive-based GPU programming models on a block Eigensolver with consideration of large sparse matrices," Lawrence Berkeley National Laboratory, Tech. Rep., 2020.

[6] F. Yu, P. Strazdins, J. Henrichs, and T. Pugh, "Shared memory and GPU parallelization of an operational atmospheric transport and dispersion application," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 729–738.

[7] M. Chillarón, G. Quintana-Ortí, V. Vidal, and G. Verdú, "Computed tomography medical image reconstruction on affordable equipment by using out-of-core techniques," *Computer Methods and Programs in Biomedicine*, vol. 193, p. 105488, 2020.

[8] X. Guo, J. Wu, Z. Wu, and B. Huang, "Parallel computation of aerial target reflection of background infrared radiation: Performance comparison of OpenMP, OpenACC, and CUDA implementations," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 4, pp. 1653–1662, 2016.

[9] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication for billion-node graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, May 2017.

[10] D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370069

[11] S. R. Miri Rostami and M. Ghaffari-Miab, "Finite difference generated transient potentials of open-layered media by parallel computing using OpenMP, MPI, OpenACC, and CUDA," *IEEE Transactions on Antennas and Propagation*, vol. 67, no. 10, pp. 6541–6550, 2019.

[12] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," *Procedia Computer Science*, vol. 108, pp. 495 – 504, 2017.

[13] Nvidia, "Cuda toolkit documentation," *https://docs.nvidia.com/cuda/cublas/index.html*.

[14] Intel, "Developer reference for intel math kernel library," *https://software.intel.com/content/www/us/en/develop/*.

[15] R. Gu, Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang, "Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2539–2552, 2017.

[16] K. Li, Y. Pan, and S. Q. Zheng, "Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 8, pp. 705–720, Aug 1998.

[17] R. Lim, Y. Lee, R. Kim, and J. Choi, "OpenMP-based parallel implementation of matrix-matrix multiplication on the Intel Knights landing," in *Proceedings of Workshops of HPC Asia*, 2018, pp. 63–66.

[18] M. Marques, G. Quintana-Orti, E. S. Quintana-Orti, and R. A. van de Geijn, "Solving large dense matrix problems on multi-core processors," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–8.

[19] J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, "Towards a multi-array architecture for accelerating large-scale matrix multiplication on FPGAs," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[20] J. Alman and V. V. Williams, "Limits on all known (and some unknown) approaches to matrix multiplication," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, 2018, pp. 580–591.

[21] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 229–241.

[22] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'16. New York, NY, USA: ACM, 2016, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2883404.2883420

[23] K. Hwang, *Advanced computer architecture with parallel programming*. McGraw-Hill, 1993.

[24] P. Boulet, A. Darte, G. A. Silber, and F. Vivien, "Loop parallelization algorithms: From parallelism extraction to code generation," *Parallel Computing*, vol. 24, no. 3-4, 1988.

[25] V. Sarkar, "Optimized unrolling of nested loops," *International Journal of Parallel Programming*, vol. 29, no. 5, 2001.

[26] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice." *ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland,*, vol. 44, pp. 38–49, June 2009.

[27] J. Ansel, "Petabricks: a language and compiler for algorithmic choice," Master's thesis, MIT, 2009.

[28] P. Dickens and J. Logan, "Towards a high performance implementation of MPI-IO on the Lustre file system," in *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, 2008, pp. 870–885.

[29] P. M. Dickens and J. Logan, "A high performance implementation of MPI-IO for a Lustre file system environment," *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 22, pp. 1433–1449, September 2009.

[30] L. Grandinetti, G. Joubert, and M. Kunze, *Big Data and High Performance Computing*, ser. Advances in Parallel Computing. IOS Press, 2015.

[31] Lustre, "Lustre file system," *http://lustre.org/documentation/*.

[32] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Comput.*, vol. 33, no. 3, pp. 213–234, April 2007.

[33] G. D. Ghare and S. T. Leutenegger, "Improving speedup and response times by replicating parallel programs on a snow," in *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 264–287.

[34] Z. Qiu, J. F. Pérez, and P. G. Harrison, "Tackling latency via replication in distributed systems," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, March 2016.

[35] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, 2014, pp. 599–600.

[36] Top500, "Top500 list - june 2015," *https://www.top500.org/list/2015/06/*.

[37] Altair, "PBS professional," *https://www.altair.com/pbs-professional/*, February 2020.

[38] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, December 2011.

[39] Fujitsu, "Fujitsu Exabyte file system (FEFS)," *https://www.fujitsu.com/downloads/TC/sc11/fefs-sc11.pdf*.

[40] S. Ghosh, J. R. Hammond, A. J. Peña, P. Balaji, A. H. Gebremedhin, and B. Chapman, "One-sided interface for matrix operations using MPI-3 RMA: A case study with Elemental," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 185–194.

[41] X. S. Li and J. W. Demmel, "SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, June 2003.

[42] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared memory programming," *IEEE Computational Science and Engineering*, pp. 46–55, January 1998.

[43] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, August 1979.

[44] A. P. Reeves, "Parallel pascal: An extended pascal for parallel computers," *Journal of Parallel and Distributed Computing*, vol. 1, no. 1, pp. 64 – 80, 1984.

[45] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, J. Kolodziej, A. Y. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, P. Bouvry, H. Li, L. Wang, D. Chen, and A. Rayes, "A survey on resource allocation in high performance distributed computing systems," *Parallel Computing*, vol. 39, no. 11, pp. 709 – 736, 2013.

[46] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 58:1–58:12.

[47] B. Klenk and H. Fröning, "An overview of MPI characteristics of exascale proxy applications," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Springer International Publishing, 2017, pp. 217–236.

[48] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb 2009, pp. 427–436.

[49] F. J. M.-Z. . J. R. P. Alonso, R. Cortina, "Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA," *Journal of Supercomputing*, pp. 215–225, 2011.

[50] D. D. Nikolić, "Parallelisation of equation-based simulation programs on heterogeneous computing systems," *PeerJ Computer Science*, 2018.

[51] L. Dongha, O. Jinoh, and Y. Hwanjo, "OCAM: Out-of-core coordinate descent algorithm for matrix completion," *INFORMATION SCIENCES*, vol. 514, pp. 587 – 604, April 2020.