Elasticity in a Task-based Dataflow Runtime Through Inter-node GPU Work Stealing

Joseph John
National Computational Infrastructure
Canberra, Australia
School of Computing
Australian National University
Canberra, Australia
ORCID: 0000-0002-0031-4793

Josh Milthorpe
Oak Ridge National Laboratory
Oak Ridge, TN, USA
School of Computing
Australian National University
Canberra, Australia
ORCID: 0000-0002-3588-9896

***Abstract*—Most contemporary HPC programming models assume an inelastic runtime in which the resources allocated to an application remain fixed throughout its execution. Conversely, elastic runtimes can expand and shrink resources based on availability and/or dynamic application requirements. In this paper, we implement elasticity for PaRSEC, a task-based dataflow runtime, using inter-node GPU work stealing. In addition to supporting elasticity, we demonstrate that inter-node GPU work stealing can enhance the performance of imbalanced applications by up to 45%.**

***Index Terms*—Distributed Work Stealing, Elastic computing, Malleable computing, Task-based programming, Dataflow, PaRSEC**

I. Introduction

The computational capabilities of high-performance computing (HPC) systems are continually advancing, pushing the boundaries of exascale computing in recent times. However, the dominant process-centric programming models have not kept pace with this rapid progress, as they require significant programmer effort to efficiently manage communication and synchronization. The task-based dataflow programming model has emerged as an alternative, promising to provide superior performance and scalability for applications. In this model, an application comprises a set of tasks with dependencies established based on the data flow between them. These tasks can be executed in any order that preserves their dependency relationships. Despite these advantages, a notable limitation in task-based dataflow runtimes is the absence of features such as "elasticity" or "malleability" as part of the runtime capabilities.

Elasticity is the ability of a runtime to dynamically adjust to the changing resource based on the demand of the application or the availability of resources. In High-Performance Computing (HPC), achieving this flexibility necessitates two components: first, a runtime capable of reorganizing data and computations and second, a job scheduler that allows the allocated resources to expand or shrink as needed. In this paper, we investigate the first requirement- we explore whether work stealing can be leveraged as a useful tool for elastic computing in the context of the PaRSEC task-based dataflow programming runtime.

In PaRSEC, for each task type, programmers can provide multiple task kernels optimized for different devices, granting the runtime system the freedom to select the most suitable computational resource for each kernel. The runtime system then dynamically determines when and where to execute these kernels. PaRSEC advocates for a clear separation of concerns in application development. It employs a domain-specific language (DSL) to describe the Directed Acyclic Graph (DAG) of tasks that define the application's algorithm, while the scheduling of these tasks on the available computing resources becomes a runtime decision. This scheduling is driven by specific optimization criteria, aiming to minimize application execution time on the target system. Crucially, PaRSEC's runtime system takes on the responsibility of managing data movement between devices and between the host and devices and coordinating the execution of tasks on these devices. This occurs without intervention from the programmer, achieving the desired separation of concerns.

Work stealing has previously been demonstrated to enhance application performance by improving the load balancing between execution units. Most evaluations of work stealing, especially in a distributed context, have focused on multi-threaded CPU execution. In many high-performance computing systems today, GPUs are the primary source of computational power due to their greater memory bandwidth and available parallelism. Consequently, most modern applications are designed with the goal of fully harnessing the potential of GPUs. This paper describes the implementation of work stealing between GPUs in different compute nodes and how it can be utilized for elastic computing.

The main contributions of this paper are to:

1) implement an inter-node GPU work-stealing mechanism for a task-based dataflow programming model and show how an imbalanced benchmark can benefit from inter-node GPU work-stealing; and
2) demonstrate how an inter-node GPU work-stealing mechanism can be employed for elasticity within a task-based dataflow programming model.

II. Related Work

In certain scenarios, achieving load balancing through work stealing and work sharing is regarded as a form of elasticity; however, in this paper, elasticity solely pertains to the capacity to *expand* and *shrink* the number of compute nodes available for an application. *Expand* signifies the action of adding compute nodes, while *shrink* denotes the action of removing the compute nodes. We will also use the term *resize* to encompass both *expand* and *shrink* operations.

A. Elasticity via Fault Tolerance

Fault tolerance employs a checkpoint and restart approach to address faults. In this method, an application periodically records its current status to storage as checkpoints. When an error occurs, the application can resume from a prior state by retrieving the relevant checkpoint. This mechanism can also be used to implement elasticity, where the application can resume from a checkpoint during a resize operation instead of a fault.

Vadhiyar and Dongarra [15] implement a checkpointing library and a runtime support system that can use this library. The user is responsible for calling the function inside an application to specify the checkpoints and restoring the state in case of a resize. The main drawback of this system is that the user has additional responsibilities. Another issue is that the runtime support system relies on a daemon process, necessitating its initiation on each node prior to launching the application. Additionally, the user must supply a configuration file to the daemon, providing it with essential communication information such as port number.

Process Checkpointing and Migration (PCM) [4] employs the Internet Operating System [16] as its foundation for enabling elasticity. The Internet Operating System serves as middleware with capabilities encompassing the distribution of computations among actors, the orchestration of dynamically evolving resources, and the load distribution across different nodes. Within this system, it is possible to spawn new MPI processes or merge existing ones, thus facilitating elasticity. However, PCM's compatibility is limited to the Internet Operating system. Additionally, application developers are responsible for explicitly defining the data redistribution strategy in the event of a resizing operation.

Lemarinier et al. [7] extend the Scalable Checkpoint Restart (SCR) library [9] to implement elasticity in MPI applications, by enabling an MPI process to access a checkpoint file written by another MPI process. User-Level Failure Mitigation (ULFM) [1] provides functionalities to grow and shrink the number of MPI processes in an application. While this has been used to deal with faults, it can also used to implement application elasticity. In both cases, the disadvantage is that the application developer must integrate fault tolerance into the application logic.

The main disadvantage of elasticity based on fault tolerance is that it requires checkpointing. While the cost of checkpointing is acceptable to mitigate faults that would require an application restart, it is prohibitively expensive to use purely for elasticity. Another disadvantage is that the user must incorporate the checkpointing APIs into their application.

B. MPI Elasticity

Comprés et al. [3] extend MPI by introducing supplementary APIs designed to facilitate elastic computing. Application developers can incorporate specific function calls that enable the addition and removal of processes from an MPI communicator at synchronization points. Additionally, this framework is seamlessly integrated into the SLURM job scheduler. The main drawback of this method is that it does not work with other job schedulers and increases the complexity of the MPI implementation.

Flex-MPI [8] is a library that adds elasticity functionality to MPICH. Flex-MPI actively monitors the application's behavior using PAPI and PMPI, collecting information to predict the application's future performance. However, one notable limitation of Flex-MPI is that it can expand to accommodate any number of MPI processes, but it lacks the capability to reduce the number of processes beyond the initial count in the communicator. Additionally, users must specify a performance objective to determine whether adaptation to the number of processes is necessary.

ReSHAPE [13] has a Job scheduler that extends GEMS [14] with support for resizing the resources allocation for a job and a programming model that includes the resizing library. The library can redistribute the data via MPI using algorithms provided in the library. The library also contains API calls to interact with the GEMS scheduler. The decision to grow or shrink the number of processes is taken by the GEMS scheduler based on the performance feedback from the application. The main disadvantage of this system is that it works only with the GEMS Job scheduler.

The main disadvantage of MPI-based approaches is that users must make significant changes to the application code to integrate the APIs that support elasticity.

C. Task-based Runtime Elasticity

Dynamic Management of Resources (DMR) [6] is a framework built on the OmpSs runtime, designed to offer elasticity with the SLURM job scheduling system. In this model, the application periodically transmits information to SLURM at synchronization points. Based on this information, SLURM may either increase or decrease the number of MPI processes. Furthermore, the SLURM job scheduler is extended to execute resizing operations based on the overall status of the High-Performance Computing (HPC) system. Once a resizing decision is communicated to the OmpSs runtime, it undertakes data redistribution in line with the altered count of MPI processes. This approach is tailored to work exclusively with a specific SLURM implementation, and it necessitates active involvement from application developers in managing the resizing operation.

Charm++ [11] uses the Torque/Maui job scheduler to implement elasticity. In this model, a Converse ClientServer (CCS) interface is used to interface between the runtime and the job scheduler. When submitting a job to the Torque/Maui scheduler, an upper bound on resources needed is also required. The job scheduler can then decide on the resize operation based on the jobs in execution. The main disadvantage of this method is that it works only with the Torque/Maui job scheduler.

Task-based runtimes that support elasticity also suffer from the same disadvantage as the MPI-based elasticity, as they require a specific job scheduler and this approach will not be portable to different HPC systems with different job schedulers. None of these runtimes employ work stealing to implement elasticity.

III. PaRSEC

PaRSEC is a distributed heterogeneous task-based dataflow runtime. In PaRSEC user defines the task and the dependency relation between them using a representation model called Job Data Flow (JDF). A task can have multiple task bodies intended for different types of devices. At present, in addition to task bodies for CPUs, PaRSEC supports task bodies for NVIDIA, AMD and INTEL GPUs.

While a user can select a preferred device, the execution of the task is the prerogative of the runtime. PaRSEC also handles communication and synchronization and provides high-level APIs for data distribution. One of the main limitations of PaRSEC is that the task mapping to compute nodes is fixed. We remove this limitation in this paper.

A. GPUs in PaRSEC

In PaRSEC, when all the data a task needs to execute is available locally, it is said to be *ready*. The ready tasks are pushed to a queue that the worker thread can access. Depending on the scheduling policy the queue can be node-level, NUMA-level or thread-level. Work stealing is possible between each thread depending on the scheduling policy employed. A node-level scheduler can be used by each thread to dequeue a task from the scheduler queue. The worker thread then decides whether to execute the task on a CPU or a GPU.

If the decision is made to execute a task on the GPU, the task is handed over to a GPU manager thread. If no manager thread is found for a GPU, the worker thread itself becomes the manager thread. Once the task is available with the manager, it pushes the task to a GPU queue and then progresses the tasks in the order they arrived. Once a task is handed over to the GPU manager, its entire life cycle - from moving data to the GPU memory, executing the task in the GPU, activating all its successors and moving data from the GPU memory to the host memory - is managed by the manager thread.

A PaRSEC task goes through a series of steps before it is executed by the GPU. Each step consists of a set of host computations and asynchronous orders sent to the various streams of the GPU. Completion of the asynchronous commands in a stage triggers the start of the next stage. We call the transition of the task from one step to another *task progression*. Between each stage, the runtime manages a queue of tasks. Each stage consumes a task from its queue, and the asynchronous completion of a stage for a task triggers the insertion of this task in the next-stage queue.

1) Task mapping: The CPU thread decides which GPU to map the tasks to. PaRSEC relies on data locality and simple load estimates to make this decision.
2) Stage-in: The task is moved to the stage-in queue. For every task in this queue, the GPU manager moves the required version of the task's data from the host memory to GPU memory (or GPU memory to GPU memory) if it is not already on the GPU.
3) Task offloading: The task is moved to the execution queue. The task in this stage has all its data available on the GPU memory, and the manager offloads the task from this queue to the GPU for execution using the execution stream. PaRSEC still controls the task as long as it is in the execution queue and task offloading has not commenced.
4) Task execution: The task is launched on the GPU. Once launched, the task is not the responsibility of the GPU manager until it is completed, and the execution of the task depends on the CUDA internal scheduler policies.
5) Stage-out: Task execution is completed and the task is moved to the stage-out queue. From this point on, PaRSEC regains control of the task. The manager transfers the task data from the GPU memory to the host memory if such a transfer is required.
6) Task epilogue: The runtime system analyses the task to discover its successors and computes the necessary data movements and task completion notifications.

B. Handshake Mechanism in PaRSEC

During the epilogue phase of a task, it sends the required data to its successors using the communication module. If one of the successors is mapped to a remote node, the thread responsible will initiate a handshake mechanism, as shown in Fig. 1. The communication module in the source node will send the information to the communication module in the destination node. Based on the information provided, the communication module in the destination will allocate memory and *fetch* the data. Once the data *fetch* instruction is received, the communication module in the source will send the data to the destination. All aspects of the handshake happen asynchronously in PaRSEC. When all the data becomes available, the task is marked as *ready* and the task is pushed to the scheduler queue.

IV. Adding Distributed Work-Stealing to PaRSEC

We added a new distributed work-stealing module to the PaRSEC runtime. The module has three main functions:

1) Initiate stealing request on a starving node.
2) Select task and migrate task on a busy node.
3) Handle all actions related to a migrated task on a starving node.

Starvation is detected on a GPU when the number of tasks available on that GPU goes below a threshold. A particular GPU may be starving even though other processing elements

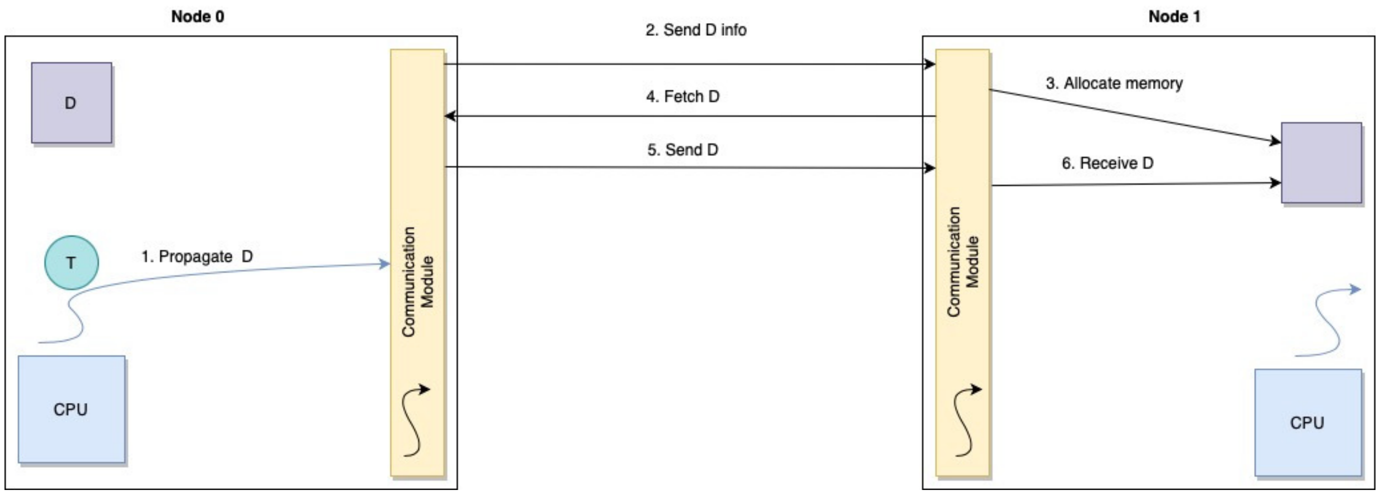

Fig. 1. PaRSEC handshake mechanism

in the node are not. In our previous work, we had shown that some applications may create a node-level imbalance where some GPUs have a glut of tasks while some may be starving. In this work, we also determine node-level starvation when the number of tasks available to all the GPUs in a node drops below a threshold. The starvation is detected by the worker threads, and any worker thread can initiate a steal request. While any worker thread can detect starvation and initiate a steal request, at any time, there can be only one outstanding steal request from a particular node.

If there is no outstanding steal request, the work-stealing module will proceed with sending the steal request to a *victim* node. One of the main design principles of our stealing module is not to collect load information to choose the victim node. Collecting load information is expensive in terms of data transfer, and it is not scalable. Another option some runtimes have used is to collect load information from a subset of nodes and limit the stealing to this subset of nodes. This strategy reduces the load balancing opportunities while still requiring data collection. Another drawback is that the load information collected may not represent the current load as the task execution is dynamic. Due to the above reasons, we have employed a random victim selection policy. Once the victim node is decided, a steal request is sent to the node requesting a *chunk* size number of tasks.

Once the steal request is received by the victim node, it is pushed to a node-level queue. Any worker thread can process a steal request by transitioning to a *stealing* thread, but at any instant, only one thread in a node can assume this role. This design choice was made to reduce contention. In our previous work, we found that contention on the GPU level queue is one of the main factors that hamper performance. Allowing only one thread to process the steal request ensures that only one thread competes with the manager thread for the GPU tasks.

As mentioned earlier, PaRSEC has four queues per GPU: device queue, stage-in queue, execution queue and stage-out queue. For node-level migration, tasks are only taken from the device queue. For each GPU, the stealing thread locks the device queue and attempts to remove a task. A task is stolen only if it does not result in starvation for that GPU. We don't migrate tasks from the stage-in queue to ensure that the manager thread has some tasks to progress and there is no starvation on the victim node. The tasks in the execution queue already have all their data in the GPU memory, so migrating these tasks to another node would be expensive. The tasks in the stage-out queue have already been executed, so there is no need to migrate tasks in that queue.

A. Victim Selection

The steal request policy determines the victim node to which a steal request will be sent. In cases where global information about the load is not collected, the *Random* policy has demonstrated significant effectiveness in selecting the victim node [10]. Under the *Random* policy, a steal request is dispatched to a randomly chosen node. Regardless of whether the request is fulfilled by the victim node, a response is sent back to the thief node, and the process is repeated.

We modified the *Random* policy to achieve greater success in victim selection. Each thief node records the last victim node that resulted in a successful steal. A thief first sends each new steal request to its last-recorded successful victim node. If the victim node cannot fully satisfy the request, it will resend the steal request to the last successful victim recorded on this victim node, acting on behalf of the original thief node. This process is repeated N times, where N is the maximum hop count for the steal request. In the absence of a successful victim, the request will be sent to a random node.

B. Migrating Tasks

Tasks in PaRSEC are instances of a *task class*. The member functions of a task class describe each aspect of the task. For example, a task class will describe the kernels a task executes, which devices the tasks can be executed on, how it sends the data to the successors and so on. Tasks of the same task class execute the same code, but each task has a unique identifier, and the data they operate on may differ.

In PaRSEC, all nodes have access to all task classes. As every task in a class has the same properties, we can recreate a task in any thief node as long as we make the data items and the task unique ID available on the thief node. As the stealing thread steals from a device queue, every task it steals is ready, i.e., all the data items it needs are available locally. So the stealing thread at the victim node can create a message for the thief node that describes the data items of the task and its unique ID. The stealing thread then sends this information to the thief node. The thief node uses the handshake mechanism described in Section III-B to fetch each data item of the stolen tasks from the victim node. Once the data items are available, the thief node can recreate the tasks using the unique ID of the task received from the victim.

V. Adding Elastic Mapping to PaRSEC

In PaRSEC, the mapping of a task to a node is fixed, and each task can independently discover the fixed mapping of its successor task. This is an important restriction, as the fixed mapping of a task allows the communication module to send the data item to specific nodes using the handshake mechanism (Section III-B). While this fixed task mapping facilitates cost-effective and asynchronous data transfers, it becomes a hindrance when attempting to implement a flexible runtime system.

We extended PaRSEC to support elasticity for iterative applications, by allowing a *resize* operation between iterations to either expand or shrink the number of compute nodes. When engaging in the *expand* operation, the newly added nodes can utilize the migration module to acquire tasks from other nodes. Once these tasks have been migrated to the new nodes, any subsequent attempts to migrate them to different nodes will be prohibited. In subsequent iterations, the predecessor task of this particular node will automatically send the specifics of the associated data item to the new node. To facilitate this procedure, when a task is migrated, information about the migrated task is recorded on the victim node, while corresponding details about the received tasks are recorded on the thief node. If this information is available for a particular task, the new task mapping overrides the fixed task mapping. This strategy ensures that a task, once migrated, will remain on the newly added nodes unless a shrink event occurs, preventing further migrations.

Every node within the system is paired with a backup node, facilitating the implementation of *shrink* operations. During the shrink operation, all nodes in the system are notified about the reduction in scale via a broadcast message. In subsequent iterations, any messages directed to the node that was taken offline will be routed to its designated backup node. As the backup node(s) may now have significantly more work than other nodes, task migration will be employed to redistribute tasks to achieve load balancing. In this phase, any task may be migrated; not only those tasks that were moved to the backup node(s) due to shrinking.

VI. Results

A. Experimental Setup

The experiments were conducted on the *Gadi* supercomputer in the National Computing Infrastructure, Australia. Each GPU node on *Gadi* has two 24-core Intel Xeon Scalable Cascade Lake processors with 3.2 GHz clock speed and 192 GiB of host memory, and four NVIDIA V100 GPUs with 32 GiB of HBM2 memory. All experiments were run using OpenMPI (v4.1.4), Intel-MKL (v2020.3.304), CUDA (11.7.0) and GCC (v10.3.0). As there is only one MPI process per node, *node* and *process* are used interchangeably in this section.

Gadi only permits a maximum of 20 GPU nodes to be allocated per job, so our experiments are limited by this number. As explained earlier in this paper, we only investigate whether task-based dataflow runtime can accommodate elasticity through distributed work stealing. We are not concerned with building a scheduler that supports elastic resource allocation. In a complete system, this task-based dataflow runtime would be complemented by a job scheduler that instructs the runtime to resize. Due to the absence of such a scheduler, in these experiments, we perform multiple iterations of the application task graph and perform the resize operation during the third iteration of each application.

B. Benchmarks

We use three benchmarks in the experiments - the Block-Sparse GEneral Matrix Multiplication (BSpGEMM) [5], Block-Dense GEneral Matrix Multiplication (GEMM) and Block-Dense Cholesky Factorization [2]. BSpGEMM is inherently imbalanced due to the non-uniform structure of the physical problem it represents.

C. Inter-node GPU Work Stealing

We first investigated the potential benefits of inter-node GPU work stealing for an imbalanced application, by conducting strong scaling experiments using the BSpGEMM benchmark, which is inherently imbalanced according to the tile sparsity pattern. Depending on the tile pattern, some nodes may have a lot of tasks, while others may have very few. Establishing a balanced static distribution for all possible tile patterns is challenging. Table I illustrates the strong scaling performance of the BSpGEMM benchmark, where both Matrix A and Matrix B exhibit 50% sparsity. The results indicate a significant improvement (between 16% - 45% speedup) in application performance when work stealing is employed.

TABLE I
Weak scaling performance of inter-node GPU work-stealing tested on the BSpGEMM benchmark. Sparse Matrix A dimensions $150k \times 300k$, tile size $1k \times 1k$, sparsity 50%. Sparse Matrix B dimensions $300k \times 300k$, tile size $1k \times 1k$, sparsity 50%.

Nodes	Performance without Migration (GFLOP/s)	Performance with Migration (GFLOP/s)
4	841 - 875	1255 - 1270
8	1495 - 1841	2162 - 2664
16	3120 - 3813	4140 - 4445

We also tested whether the inter-node GPU work stealing is effective when the application is already balanced statically. Table II gives the performance of the GEMM benchmark and Table III gives the performance of the Cholesky factorization benchmark. In both tables, we can see a small improvement in performance with work stealing.

TABLE II
STRONG SCALING PERFORMANCE OF INTER-NODE GPU WORK-STEALING TESTED ON THE GEMM BENCHMARK. MATRIX A DIMENSIONS $100k \times 200k$. MATRIX B DIMENSIONS $200k \times 100k$. TILE SIZE $1k \times 1k$.

Nodes	Performance without Migration (GFLOP/s)	Performance with Migration (GFLOP/s)
4	14957 - 19975	15329 - 20975
8	31077 - 35486	33650 - 35994
16	49525 - 52667	51246 - 53870

TABLE III
STRONG SCALING PERFORMANCE OF INTER-NODE GPU WORK-STEALING TESTED ON THE CHOLESKY FACTORIZATION BENCHMARK. MATRIX A DIMENSIONS $200k \times 200k$, TILE SIZE $1k \times 1k$.

Nodes	Performance without Migration (GFLOP/s)	Performance with Migration (GFLOP/s)
4	34843 - 36252	34197 - 37311
8	41414 - 44330	41614 - 45051
16	63513 - 66739	63571 - 66815

D. *Elastic Expand*

The performance of Dense GEMM benchmarks under various expansion scenarios is presented in Table IV. To determine the expected performance, we executed the same benchmark using a total of N nodes throughout, where $N =$ Initial nodes $+$ Expanded Nodes. Actual performance closely aligns with the expected values when either two or four nodes are added; however, actual performance falls short of expected performance when eight nodes are added.

TABLE IV
PERFORMANCE OF *Expand* OPERATION TESTED ON THE GEMM BENCHMARK. MATRIX A DIMENSIONS $100k \times 200k$. MATRIX B DIMENSIONS $200k \times 100k$. TILE SIZE $1k \times 1k$.

Initial nodes = 8			
Performance before expansion (GFLOP/s)	**Expand by**	**Performance after expansion (GFLOP/s)**	**Expected performance (GFLOP/s)**
31077 - 35486	2	37606 - 37652	36429 - 38842
	4	40198 - 43140	40775 - 43021
	8	45677 - 46929	49525 - 52667

The disparity in performance can be attributed to the fact that, as the number of nodes increases during expansion, the expanded nodes receive fewer tasks. This is because the ability of work stealing to distribute the tasks depends on the available parallelism (in this case, the number of ready tasks) at each instant. When the number of nodes increases, the available parallelism is not sufficient to make full use of the expanded nodes. Therefore, while the performance improves compared to the baseline, it falls short of the expected level.

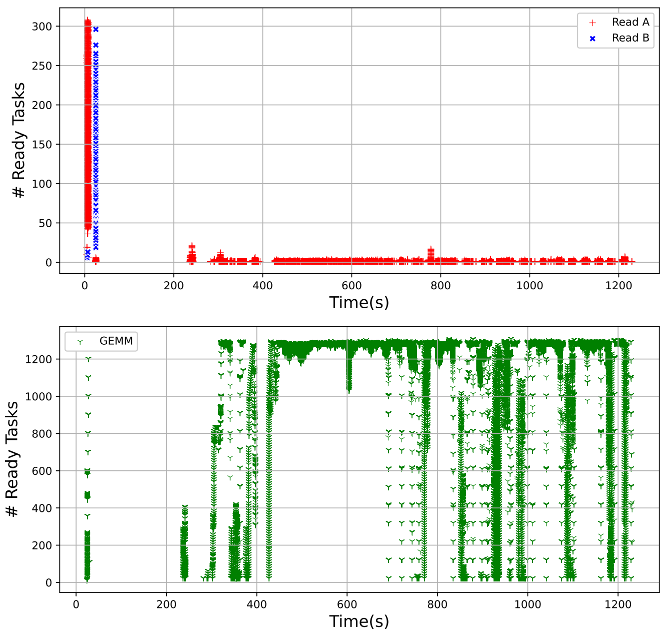

Fig. 2. Ready tasks available for the GEMM benchmark. Matrix A dimensions $100k \times 200k$, Matrix B dimensions $200k \times 100k$. Tile size $1k \times 1k$.

Similarly, the performance of the Dense POTRF benchmark is presented in Table V, where the initial number of nodes remains constant, but the number of expanded nodes varies. It follows the same pattern as the GEMM benchmark, with expansion not being as effective when eight nodes are added.

TABLE V
PERFORMANCE OF *Expand* OPERATION FOR THE CHOLESKY FACTORIZATION BENCHMARK.
MATRIX DIMENSIONS $200k \times 200k$, TILE SIZE $1k \times 1k$.

Initial nodes = 8			
Performance before expansion (GFLOP/s)	**Expand by**	**Performance after expansion (GFLOP/s)**	**Expected performance (GFLOP/s)**
41414 - 44330	2	46643 - 49348	46750 - 49559
	4	50134 - 55706	50160 - 55974
	8	50622 - 57113	63513 - 66739

In both cases, the expansion is effective for fewer added nodes, as there are enough ready tasks to be stolen during the *expand* operation. Fig. 2 displays the number of ready from one of the nodes when the GEMM benchmark is run on 8 nodes. The GEMM application comprises three types of tasks: *READ A* tasks, *READ B* tasks, and *GEMM* tasks. The *READ A* and *READ B* tasks read the tiles that form the input for the GEMM tasks. In this application, Matrix A is copied across all nodes, while Matrix B is distributed among nodes. The number of GEMM tasks ready at a given node may vary widely over time depending on the order in which tiles are read. This observation is further supported by Figure 3,

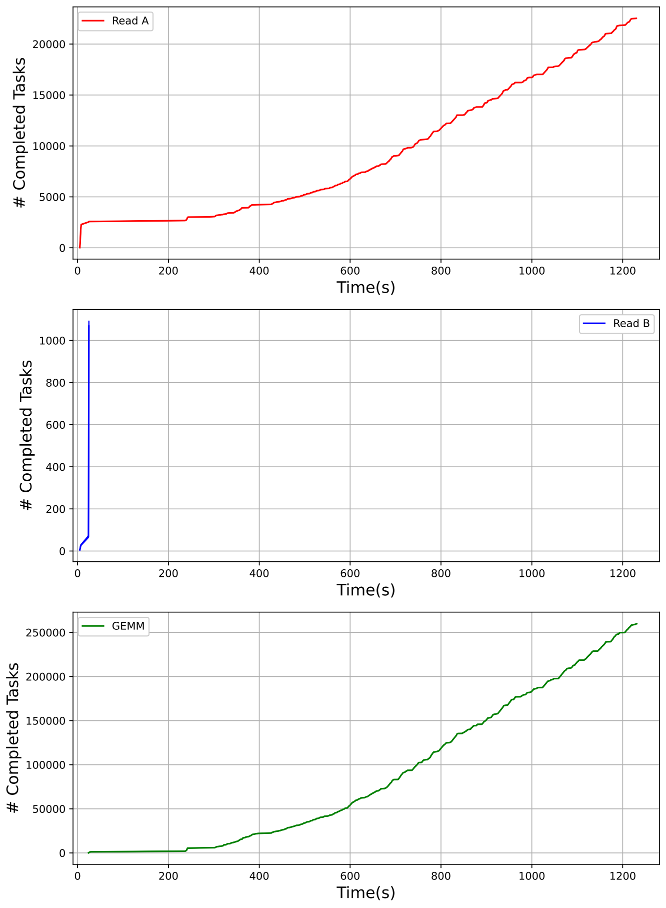

Fig. 3. Tasks completed for the GEMM benchmark. Matrix A has dimensions $100k \times 200k$ and Matrix B has dimensions $200k \times 100k$. Tile size is $1k \times 1k$.

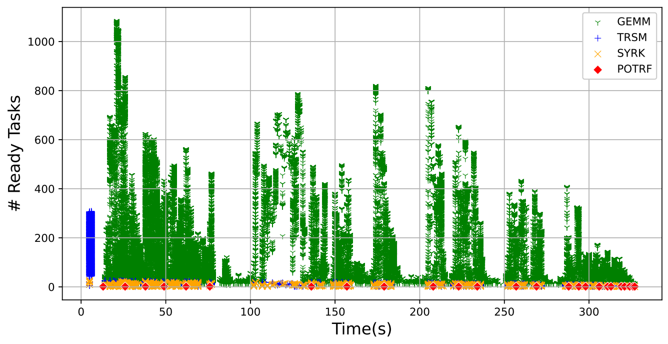

Fig. 4. Ready tasks available for the Cholesky factorization benchmark. Matrix A has dimensions $200k \times 200k$. Tile size is $1k \times 1k$.

depicting the number of completed tasks during the course of execution. Over an extended duration, GEMM tasks are not triggered as their corresponding READ A tasks have not yet completed execution. From the figures, we can see that the Dense GEMM application has enough ready tasks (GEMM tasks) to support stealing from another node. Simultaneously, this work stealing becomes less effective when the number of thief nodes competing to steal increases, as the tasks available

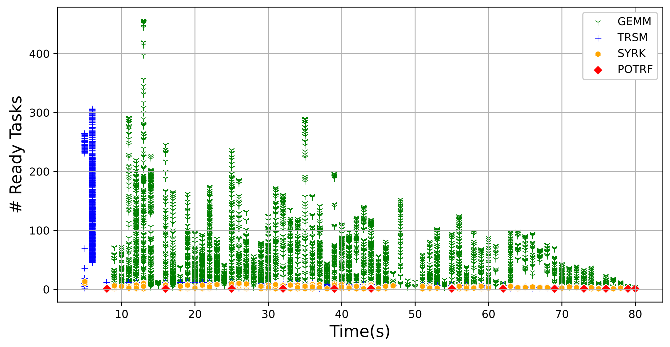

Fig. 5. Ready tasks available for the Cholesky factorization benchmark. Matrix A has dimensions $100k \times 100k$. Tile size is $1k \times 1k$.

to steal per thief decreases.

Similarly, Fig. 4 reveals the number of ready tasks for the Cholesky factorization benchmark, indicating a sufficient number of ready tasks to facilitate the *expand* operation. However, upon reducing the matrix dimension by half, as illustrated in Fig. 5, the application exhibits a scarcity of ready tasks and consequently lower than expected performance.

E. Elastic Shrink

We executed the Dense GEMM benchmark under different shrinking scenarios, comparing the performance after shrinking with the expected performance calculated by running the same benchmark using a total of N nodes throughout, where N = Initial nodes − Shrunk Nodes. Table VI shows that the performance after shrinking is much lower than expected.

TABLE VI
PERFORMANCE OF *Shrink* OPERATION TESTED ON THE GEMM BENCHMARK. MATRIX A DIMENSIONS $100k \times 200k$. MATRIX B DIMENSIONS $200k \times 100k$. TILE SIZE $1k \times 1k$.

Initial nodes = 16			
Performance before shrinking (GFLOP/s)	**Shrink by**	**Performance after shrinking (GFLOP/s)**	**Expected performance (GFLOP/s)**
49525 - 52667	2	17571 - 20799	43912 - 48930
	4	15321 - 17203	40775 - 43021
	8	12338 - 14606	31077 - 35486

To understand this performance disparity, we measured the total communication arising from an activation message. From Table VII, we can see that before the shrink operation, the total communication is consistent across all nodes. However, this pattern shifts during the *shrink* operation, with increased communication observed both on the backup node and on nodes without backup designation. Furthermore, the processing of steal requests introduces additional communication. After the *shrink* operation, communication levels remain elevated.

VII. CONCLUSION

In this paper, we show that an imbalanced application can benefit significantly from inter-node GPU work stealing. In these applications, predicting work distribution beforehand

TABLE VII

COMMUNICATIONS DURING A *Shrink* OPERATION TESTED ON THE GEMM BENCHMARK. MATRIX A DIMENSIONS $100k \times 200k$. MATRIX B DIMENSIONS $200k \times 100k$. TILE SIZE $1k \times 1k$, INITIAL NUMBER OF NODES IS 16, SHRINKING TO 12.

	Communications		**Steal Requests**	
	Backup node	**Normal node**	**Backup node**	**Normal node**
Before shrink	18700	18700	0	0
During shrink	157000	37400	2880	0
After shrink	107300	56400	0	0

is impractical, making it challenging to achieve fair static distribution, particularly within a decentralized runtime like PaRSEC, where runtime decisions are entirely decentralized. We utilized work stealing because it operates independently at the node level and does not necessitate information about the load on other nodes within the cluster.

Importantly, although inter-node work stealing is less effective for an already well-balanced application, it does not compromise performance.

Additionally, we show that inter-node GPU work stealing can function as a mechanism for elasticity. In particular, applications with a high degree of available parallelism (ready tasks) approach a fair static distribution for the expanded nodes. Conversely, applications with fewer ready tasks experience some benefits from elasticity, but their performance falls short of the anticipated level. Furthermore, our findings reveal that when resources are shrunk, performance fails to meet expectations, which we attribute to the increased communication resulting from the shrink operation.

This paper demonstrates the use of inter-node GPU work stealing to support elasticity within a task-based dataflow runtime. In the context of high-performance computing systems, true elasticity can be realized only by integrating an elastic runtime with support from a batch system such as PBS or SLURM. The runtime must initiate compute resource allocation requests to the batch system when expansion operations are underway and should be capable of releasing compute resources during shrink operations. Facilitating this communication between the runtime and batch system is essential. Thus, modifications will be needed to commonly-used batch systems to support truly elastic applications.

Although PaRSEC runtime was used for this study, any distributed runtime that supports tasks and maintains segregation between tasks and the data on which they operate can implement distributed GPU task stealing and elasticity. Our future plans to extend this research will incorporate a scheduler module similar to Dask [12] into the runtime. This module will be designed to facilitate dynamic allocation and deallocation of compute resources seamlessly. Importantly, it will be designed to function with any batch system without necessitating any modifications to the existing setup.

ACKNOWLEDGMENT

This research is undertaken with the assistance of resources and services from the National Computational Infrastructure (NCI), which is supported by the Australian Government. We thank George Bosilca and Thomas Herault (Innovative Computing Laboratory, University of Tennessee, Knoxville) for the detailed design discussions.

REFERENCES

[1] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of MPI communication capability: Design and rationale. International Journal of High Performance Computing Applications **27**(3), 244–254 (2013). https://doi.org/10.1177/1094342013488238

[2] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Hérault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., Yarkhan, A., Dongarra, J.J.: Distributed Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA. Research report, INRIA (2010), `https://hal.inria.fr/hal-00809712`

[3] Comprés, I., Mo-Hellenbrand, A., Gerndt, M., Bungartz, H.J.: Infrastructure and API extensions for elastic execution of MPI applications. In: Proceedings of the 23rd European MPI Users' Group Meeting. p. 82–97. EuroMPI 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2966884.2966917

[4] El Maghraoui, K., Szymanski, B.K., Varela, C.: An architecture for reconfigurable iterative MPI applications in dynamic environments. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics. pp. 258–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/https://doi.org/10.1007/11752578_32

[5] Herault, T., Robert, Y., Bosilca, G., Harrison, R.J., Lewis, C.A., Valeev, E.F., Dongarra, J.J.: Distributed-memory Multi-GPU Block-sparse Tensor Contraction for Electronic Structure. In: IPDPS (2021). https://doi.org/10.1109/IPDPS49936.2021.00062

[6] Iserte, S.: High-throughput Computation through Efficient Resource Management. Ph.D. thesis, Universitat Jaume I (Nov 2018). https://doi.org/10.6035/14101.2018.176272

[7] Lemarinier, P., Hasanov, K., Venugopal, S., Katrinis, K.: Architecting malleable MPI applications for priority-driven adaptive scheduling. In: Proceedings of the 23rd European MPI Users' Group Meeting. p. 74–81. EuroMPI 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2966884.2966907

[8] Martín, G., Singh, D.E., Marinescu, M.C., Carretero, J.: Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration. Parallel Computing **46**, 60–77 (2015). https://doi.org/10.1016/j.parco.2015.04.003

[9] Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10, IEEE Computer Society, USA (2010). https://doi.org/10.1109/SC.2010.18

[10] Perarnau, S., Sato, M.: Victim Selection and Distributed Work Stealing Performance: A Case Study. IPDPS (2014). https://doi.org/10.1109/IPDPS.2014.74

[11] Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A batch system with efficient adaptive scheduling for malleable and evolving applications. In: IEEE International Parallel and Distributed Processing Symposium. pp. 429–438 (2015). https://doi.org/10.1109/IPDPS.2015.34

[12] Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling. In: Python in Science Conference (SciPy) (2015), https://api.semanticscholar.org/CorpusID:63554230

[13] Sudarsan, R., Ribbens, C.J.: Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In: International Conference on Parallel Processing (ICPP). p. 44 (2007). https://doi.org/10.1109/ICPP.2007.73

[14] Tadepalli, S.S.: GEMS: A Fault Tolerant Grid Job Management System. Master's thesis, Virginia Polytechnic Institute, (2003), https://api.semanticscholar.org/CorpusID:110908258

[15] Vadhiyar, S.S., Dongarra, J.J.: SRS: A framework for developing malleable and migratable parallel applications for distributed systems. Parallel Processing Letters **13**(02), 291–312 (2003). https://doi.org/10.1142/S0129626403001288

[16] Varela, C., Maghraoui, K.E., Desell, T.: Load balancing of autonomous actors over dynamic networks. In: 37th Hawaii International Conference on System Sciences. vol. 10, p. 90268a. IEEE Computer Society, Los Alamitos, CA, USA (Jan 2004). https://doi.org/10.1109/HICSS.2004.10046