Multi-GPU Work Sharing in a Task-Based Dataflow Programming Model

Joseph John[a], Josh Milthorpe[a,c], Thomas Herault[b], George Bosilca[b]

[a]*Australian National University, Canberra, Australia*
[b]*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA*
[c]*Oak Ridge National Laboratory, Oak Ridge, TN, USA*

Abstract

Today, multi-GPU computing nodes are the mainstay of most high-performance computing systems. Despite significant progress in programmability, building an application that efficiently utilizes all the GPUs in a computing node is still a significant challenge, especially using the existing shared-memory and message-passing paradigms. In this aspect, the task-based dataflow programming model has emerged as an alternative for multi-GPU computing nodes.

Most task-based dataflow runtimes have dynamic task mapping, where tasks are mapped to different GPUs based on the current load, but once the mapping has been established, there is no rebalancing of tasks even if an imbalance is detected. In this paper, we examine how automatic dynamic work sharing between GPUs within a compute node can improve the performance of an application through better workload distribution. We demonstrate performance improvement through dynamic work sharing using a Block-Sparse GEneral Matrix Multiplication (BSpGEMM) benchmark. Although we use PaRSEC, a task-based dataflow runtime, as the vehicle for this research, the ideas discussed here are transferable to any task-based dataflow runtime.

Keywords:
Tasks, Runtime, Work Sharing, PaRSEC, GPU, Load Balancing

1. Introduction

In many high-performance computing systems today, the main source of computing power is GPU devices, with their high memory bandwidth and powerful parallel computing capabilities. Although in 2007, none of the Top500 computer systems included GPU accelerators, by 2022, 168 of them were using at least one GPU per node, and only two of the top 10 supercomputers in November 2022 did not use accelerators [1]. The first Exascale machine, Frontier, comes with 4 GPUs per node, which provide more than 90% of its computing capability. To fully utilize this computing power, an application must distribute its workload over the GPUs. Despite significant progress, programming for such systems remains a difficult task, as the programmer must manage low-level issues of distributed and per-device memory spaces, scheduling, synchronization, device-to-host communication and load balancing between the many devices in the node.

PaRSEC [2, 3] is a task-based dataflow programming model that moves this low-level decision-making from the programmer to the runtime. In this model, an application is a collection of tasks with dependencies derived from the data flow among the tasks. Tasks can be executed in any order that maintains the dependency relations between them. For each type of task, the programmer can provide multiple task kernels aimed at different devices, providing the runtime with the flexibility to choose the best computational resource for each kernel. The runtime will then dynamically decide when and where to execute these kernels. PaRSEC promotes separation of concerns in application development: a domain-specific language (DSL) provides the Directed Acyclic Graph of tasks that defines the algorithm implementing the application while scheduling on the computing resources of a given platform is a runtime decision that targets some optimization criteria, such as the performance improvement for the target platform. As a fundamental element to achieve this separation of concerns, the PaRSEC runtime implements data movement between devices and between host and device. PaRSEC runtime also coordinates the execution of tasks on the devices without programmer intervention.

A key hurdle for the efficient utilization of multi-GPU systems is load balancing. An application may offload more work to a subset of the system's GPUs while the rest will remain underutilized. The reason for this imbalance is manifold: in some applications, it is not possible to find an ideal task mapping ahead of time, or the heuristic used by the runtime for task mapping may not suit an application, or the imbalance could be because of a programmer oversight as assumptions that are valid for one application may not hold in another. While PaRSEC addresses most of the challenges of a heterogeneous system, it does not address the problem of load imbalance between GPU devices. In this paper, we enhance PaRSEC to explore how dynamic load balancing between GPU devices improves performance.

There is a rich literature on how to partition and distribute/map tasks between CPUs and GPUs in a node (Sec-

Email addresses: joseph.john@anu.edu.au (Joseph John), josh.milthorpe@anu.edu.au (Josh Milthorpe), herault@icl.utk.edu (Thomas Herault), bosilca@icl.utk.edu (George Bosilca)

tion 2). Most of these partitions are based on data locality, heuristics, or a combination of both. While these mapping strategies work for regular applications, for which the Directed Acyclic Graph (DAG) of tasks is static, they can create a problem for irregular applications, for which the DAG of tasks is input-data or even computation-dependent, as an irregular application may have unpredictable memory access, data flow, or/and control flow.

Although most of these mapping strategies consider the load of the GPUs before mapping a task to the GPU, they do not migrate tasks between GPUs after the mapping decision has been made. We are interested in how load balancing can help *after* this partition results in load imbalances.

1.1. Contributions

The contributions of this paper are as follows:

1. Enhance the PaRSEC runtime to enable inter-GPU work sharing for automatic load balancing.
2. Investigate whether having a co-manager thread in addition to a manager thread helps improve the performance.
3. Investigate whether task selection policies play a role in improving performance.
4. Investigate whether migrating a chunk of tasks plays a role in improving performance.

2. Related Work

Zheng et al. [4] implement a data processing system in which data are divided into equal-sized chunks. A fuzzy neural network (FNN) is used to predict the real-time computational performance of a GPU and this prediction is used to assign chunks of data to the GPU for processing. While prediction enables better data mapping, the data is not moved to another GPU if there is a dynamic load imbalance. Chen et al. [5] implement a multi-GPU system on Flink where work is assigned to a GPU based on a locality-aware algorithm, but there is no dynamic load-balancing between the GPUs. Similarly, Gautier et.al [6] implements locality-based multi-GPU task-based runtime, where tasks are mapped to a GPU based on the locality of the data and are not subsequently migrated to other GPUs. Troodon [7] implements task migration between CPU and GPU for OpenCL applications using a machine learning framework. While there is load balancing between CPU and GPU, load balancing between GPUs is not considered.

Chen et al. [8] launch persistent worker thread blocks in the GPU. One thread among the worker threads takes charge of task management using a common queue shared between the worker thread blocks of all GPUs in the node. While this may allow robust reactions to imbalance, forcing GPU threads to be management threads can add significant overheads. Chatterjee et al. [9] treat each steaming multiprocessor as a worker block, where worker blocks can steal tasks from other blocks in the same GPU but not between GPUs in a node. Acceleration Engine for Multi-GPU Load-balancing (AEML) [10] implements a multi-GPU load-balancing on Spark. AEML uses the number of idle streams available to measure underutilization, and it also uses a feedback mechanism to adjust the number of streams on each GPU based on its computation capability. While AEML provides a good mapping of tasks to GPU based on the current load, there is no movement of assigned tasks from one GPU to another to compensate for dynamic load imbalances. DCUDA [11] uses a wrapper function over CUDA calls to dynamically balance CUDA kernels between GPUs. DCUDA records the kernel execution time when it is first called and uses this, along with memory and thread requirements of the kernel, to evaluate the load it will exert on a GPU. This metric is used to map tasks to GPUs, but there is no load balancing after task mapping.

XKaapi [12] and StarPU [13] implement dynamic task stealing, where each GPU has a task queue associated with it, and any GPU can steal tasks from other GPUs as long as the data the task requires is not already being transferred to a particular GPU. Hermann et al. [14] propose a system based on the affinity between GPUs. Tasks are distributed to device-specific queues at the start of each iteration, and if a device starves, it can steal from a list of devices with which it has an affinity.

3. PaRSEC

Parallel Runtime Scheduler and Execution Controller (PaRSEC) [2] is a task-based dataflow runtime, where the execution of tasks is fully distributed with no centralized components. Task scheduling, detection of local and remote dependencies, movement of data between nodes, and detection of distributed termination [15] are all the responsibilities of the runtime in PaRSEC, and under the control of the user via multiple mechanisms to control the execution of the DAG of tasks. PaRSEC provides multiple domain-specific languages [16, 17], which developers can use to express their applications as a DAG. The developer is responsible for defining the tasks and the dependencies amongst the tasks using these domain-specific languages.

Each CPU thread in PaRSEC has access to a node-level scheduler and its queues. When a task is activated, it is first pushed to the scheduler's queue. CPU threads select tasks from this queue for execution. In PaRSEC, a task can have multiple implementations for different devices on the node. For instance, a general matrix multiply (GEMM) task can have one kernel that can be executed by a CPU and another kernel that can be executed by a GPU. When a CPU thread selects a task with a GPU-specific kernel, the CPU thread can hand over the task to a GPU manager thread. The runtime system is responsible for selecting the device type that will be used to execute the task, but the user can control this selection through a variety of mechanisms. In this paper, we assume that all the tasks have a CUDA kernel, and when we refer to GPU or device, we assume an NVIDIA GPU that can support CUDA 10 and above.

3.1. GPU Management

The PaRSEC runtime dedicates a manager thread to manage all aspects of task execution on a GPU. Any CPU thread can become a GPU manager thread, but at any instant there can be

only one manager thread per GPU. The transition of a CPU worker thread to a manager thread occurs on a first-come, first-served basis. Any worker thread that finds a task with a GPU kernel (*GPU task* from now on) tries to hand over the task to a GPU manager thread. If the GPU was previously idle and does not yet have a manager thread, the thread that is performing the handover itself becomes the manager. On the other hand, if a manager already exists for a GPU, other threads just push GPU tasks to the GPU-specific queue and leave the rest of the task management to the manager thread (see [18] for more details).

Each GPU in PaRSEC is divided into a configurable number of streams. One stream is reserved for moving data from host memory to GPU memory (*stage-in stream*), one stream is reserved for moving data from GPU memory to host memory (*stage-out stream*), while the rest of the streams (default is 2) are used to launch the task kernel to the GPU (*execution streams*). Using multiple streams in the GPU allows the overlap of new task submissions with execution on the GPU, and increases GPU utilization when each individual task does not expose enough parallelism to span over the entire GPU.

3.2. Memory Management

The memory of each GPU is managed by PaRSEC. A user-defined part of the GPU memory is allocated to the PaRSEC runtime when the GPU is initialized (typically 95% of the available GPU memory, but this can be controlled by the user). This memory is then divided into segments of equal size, and task data is mapped to contiguous segments of GPU memory as required. PaRSEC keeps track of the memory segments used for each data item in concert with the data management mechanism (see Section 3.3). This mechanism significantly reduces the overhead of memory allocation compared to allocating the data using the CUDA API each time a task requires it. The disadvantage of this method is that sometimes more than the required size of the memory will be assigned to a data item. An alternative to this method is to allocate memory to the GPU only when needed, but this is expensive, as the CUDA memory allocation operation is blocking.

3.3. Data Management

Each GPU in PaRSEC has its own copy of the data and each copy is versioned as shown in Fig. 1. The DSLs in PaRSEC are designed so that at most a single writer task is enabled for a particular copy of the data at any given time, and when a task writes to a data copy, its version is incremented. This ensures that only one task can change the data at any time. If task T has to execute on GPU D1, PaRSEC makes sure that the latest version of the data it needs is made available on GPU D1 during the stage-in step of the task progression. The latest copy is made available in the GPU memory using a Host-to-Device copy or a Device-to-Device copy based on which device (host or any GPU) has the latest version of the data. During stage-in, there are three scenarios that we may encounter:

1. The required version of the data is already available in the GPU D1; no transfer is required.

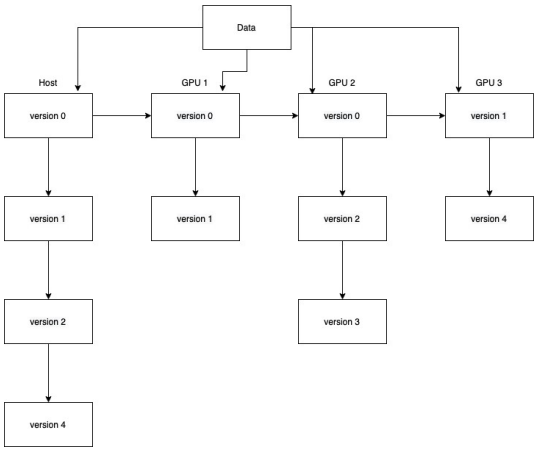

Figure 1: Device copies and data versioning

2. The required version of the data is available in another GPU D2; the data are transferred from the memory of D2 to D1 (D2D copy).
3. The required version of the data is available on the host; the data are transferred from the host to the device (H2D copy).

If the required data version is available on both the host and a GPU (or GPUs), the data are transferred from the GPU memory.

When operating under out-of-core conditions (the GPU memory is too small to hold the memory footprint of the application), the runtime system implements an eviction policy that ensures that some operations can proceed. The PaRSEC eviction policy is hardcoded in the PaRSEC CUDA device manager. The only policy available is Least-Recently Used (LRU), but others could be implemented by modifying the PaRSEC source code. As the PaRSEC CUDA device manager moves the data and schedules the tasks, it can easily track what data are currently used by the tasks, what data has been updated on the GPU, and transition the data that are not referenced by any task onto the potentially-evicted LRU.

The runtime system maintains two lists sorted in LRU order. The first list holds all read-only data (or, in general, all data that have an up-to-date version on RAM or another device), and the second holds data that have been modified and that do not have another up-to-date copy on another device. When memory is required on the GPU, the least recently read-only data are evicted, and asynchronous updates of the least recently used modified data are scheduled. When an update completes, the corresponding data moves from the modified LRU to the read-only LRU. If memory is necessary and no read-only data can be released, the GPU enters a thrashing mode and waits for the completion of tasks or data update operations to continue to the next operation.

One of the tools available to the user to fine-tune the data movement is the *preferred copy* mechanism: any given data copy can be marked by the user as the preferred copy representing this data. The PaRSEC scheduler will then use this copy as the source when the data needs to be copied from somewhere

else and multiple alternative copies have the correct version. Similarly, each data item can be assigned a *preferred device* by the user (dynamically during execution), and if multiple devices are available to host a copy of this data, the preferred device will be selected first by the runtime system.

3.4. Task management

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.

Step 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. Users can provide a *preferred device advice* to the data elements referenced by tasks to guide this selection. In this locality-based algorithm, if a GPU already holds some data the task requires, in GPU memory, the task is assigned to that GPU. If none of the task's data is currently in any of the GPUs, the task is assigned to a GPU based on the user's advice or the GPU's estimated load. The GPU load is updated whenever a task is mapped to a GPU or completes execution on a GPU. The information provided in the task description is used to estimate the amount of work associated with the task. Once the best GPU to map the task is selected, the CPU thread pushes the task into the device queue of the GPU. From this point on, all progression of the GPU task is the responsibility of the GPU manager thread. This load balancing strategy is only partially dynamic: it takes into account the current estimated load of the GPUs, but once a task is submitted to a GPU, it will not migrate to another device, so any error in the load estimate of the tasks (the task to be scheduled, but also all the tasks previously scheduled) can lead to load imbalance. We study later in the paper how dynamic task migration mitigates this issue.

Step 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 data of the task from the host memory to GPU memory (or GPU memory to GPU memory) if it is not already on the GPU, using the stage-in stream.

Step 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.

Step 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.

Step 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 host memory if such a transfer is required.

Step 6-. Task epilogue: The runtime system analyses the task to discover its successors and computes the necessary data movements and task completion notifications.

4. Adding inter-GPU load balancing to PaRSEC

GPU load is affected by a mix of user decisions (typically via the preferred device mechanism), load balancing heuristics, and data flow during execution. Although heuristics can successfully predict the load of a regular application, they easily fail for irregular applications. Similarly, although it is a simple task for the user to guide the load distribution in a regular case, deciding which device is the preferred device in an irregular data-dependent application is a very difficult task. Another approach that we advocate in this work is to rely on a mechanism that can dynamically load balance at any instance after the task has been mapped to a GPU. At any given time, there are potentially many tasks ready to be scheduled, much more than the number of tasks a single GPU can run in parallel. A dynamic approach will enable us to leverage the advantages of heuristic-based task mapping, as well as dynamic load balancing when the heuristic-based mapping fails to distribute tasks properly.

The concepts discussed here have been implemented for NVIDIA GPUs, but these concepts are applicable to any GPU that supports asynchronous data movement and asynchronous kernel submission. The PaRSEC runtime system now supports Intel and AMD GPUs, and the same modifications could be ported to both hardware without changes at the programmer level.

4.1. Work Sharing

We use work sharing instead of work stealing for load balancing. In work stealing, a manager thread of a starving GPU tries to steal tasks from the queues of a busy GPU. This poses two problems - first, the manager thread of one GPU will need to access the queues of other GPUs, which can increase contention. Second, the tasks can be in any of the stages mentioned in Section 3.4, and this could complicate the process of work stealing. For example, if the memory of the busy GPU is already allocated to the task to be stolen, the manager of the starving GPU must initiate steps to relinquish this memory.

In work sharing, the busy GPU detects a starving device and pushes some of its work to the starving GPU. This resolves the contention problem, as only the manager of the overprovisioned GPU needs to access its queues, and as it migrates

tasks that it owns, it has full control of the task stage. The mechanism to push tasks to the starving GPU leverages the existing task submission mechanism that avoids contention on queues of the starving GPUs. The manager of the busy GPU selects a task from its queue to migrate to the starving GPU. These selected tasks are pushed to a node-level queue with the information on which GPU it is intended for. Then any CPU thread can move the tasks from this node-level queue to the starving GPU using the method detailed in Section 3.1.

The only disadvantage is that all the GPUs need to know the load of the other GPUs. In shared memory, this is not a costly operation, as we can keep track of the GPU load using a host-side data structure. The manager thread of each task checks if other GPUs are starving. If starvation is detected, it selects a list of tasks to be migrated to that GPU. The check for starvation is done in a round-robin fashion, while task selection can be performed using multiple selection policies. We assume starvation if the number of uncompleted tasks falls below the number of streams available in the GPU.

4.2. Task selection policies

The selection policies decide which tasks to select in case starvation is detected on a GPU. In each policy, the manager tries to select *chunk size* number of tasks. If the manager cannot find the entire chunk size from one queue, it moves on to the next queue (first the Stage-1 queue, then the Stage-2 queue and finally the Stage-3 queue). A queue can contain both compute tasks (defined by the user) and bookkeeping tasks (introduced by the runtime to manage bookkeeping events).

During the search of the queue, the manager thread has exclusive access to the queues. This can increase contention if we are using a dedicated thread, separate from the manager thread, to migrate tasks. In each policy, the maximum task selected is equal to the chunk size. Chunk size is the upper bound on the number of tasks migrated to GPU on a single task sharing.

We experiment with five different selection policies:

1. **Single-pass:** The manager selects the first computing task from a queue.
2. **Single-try:** The manager tries to select the task from the front of the queue. The search ends if a bookkeeping task is encountered at the front of the queue. The idea here is to relinquish the lock on the queue and move on to the next queue, thereby reducing the contention on the queue in the case of a dedicated load-balancing thread.
3. **Device-affinity:** The manager tries to select a compute task with an affinity with the starving GPU. A task has an affinity with the GPU if one or more of the data it needs is already available on the starving GPU. The idea here is that we can reduce the number of stage-ins if some required data are already on the starving node.
4. **Task-affinity:** The manager tries to select a compute task with affinity with the previously selected task. A task has an affinity with another task if one or more of the data the tasks needs are the same. For the first task selected, this affinity condition is not enforced. The idea here is that if the tasks have common data, they can be reused, thereby reducing the number of stage-ins required.
5. **Two-pass:** This policy is a combination of device-affinity and single-pass. In this policy, we make two passes on the queue. In the first pass, only tasks with an affinity to the GPU are selected. In the second pass, any compute task is selected. The idea here is as far as possible to find a task with affinity with the starving device; if such a device cannot be found, find any compute tasks.

4.3. Task Types

Based on whether the data of the task is already staged-in (available on the GPU), tasks can be divided into two types:

1. Task whose data are not yet staged in.
2. Task whose data has already been staged-in.

The first type of task is available in the Stage-1 queue and the Stage-2 queue, while the second type is available in the Stage-3 queue. In previous work, only task whose data has not yet been staged-in was migrated, but in our work, we migrate both types of tasks.

4.4. Delegating work to co-manager

In PaRSEC, the GPU manager thread handles all aspects of task management. In our work, we evaluate whether delegating some aspect of the GPU task life cycle to another thread will impact performance. For this, we introduce an additional thread, the co-manager thread, to assist the manager thread. As the first CPU thread that submits a GPU task to the GPU device becomes the manager thread, the second CPU thread that submits the work to a GPU becomes the co-manager thread. If there is no co-manager available, i.e. only one task was submitted to the GPU, then the manager will handle all the work. We evaluate two aspects of work delegation:

1. Delegate epilogue completion: Much of the management work is in the task *epilogue*, where all the successors to a completed task are evaluated and potentially enabled and scheduled. During the processing of the epilogue, the progress of the GPU streams is not checked, which can increase the latency of submitting new work to the GPU.
2. Delegate migration: We delegate the responsibility of task migration to the co-manager.

5. Results

5.1. Experimental Setup

Experiments were conducted on the machine *Leconte* in the Innovating Computing Laboratory at the University of Tennessee. *Leconte* contains two NUMA nodes each with a Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz CPU and 8 NVIDIA Tesla V100 SXM2 32GB accelerators connected in a Hypercube-Mesh model [19] as shown in Fig. 2.

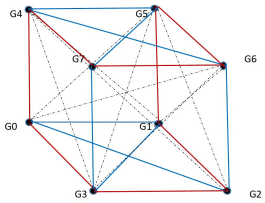

Figure 2: GPU layout in *Leconte*. The red line between GPUs represents NVLink2 connections and the blue line between GPUs represents the NVLink1 connections. The dotted black lines between GPUs represent connection traversing PCIe as well as the SMP interconnect between NUMA nodes.

5.2. Benchmarks

We use three widely used linear algebra algorithms, the Block-Sparse GEneral Matrix Multiplication (BSpGEMM) [20], Block GEneral Matrix Multiplication (GEMM) and Block Cholesky Factorization to test multi-GPU load balancing through task migration. Block-Sparse GEMM is an important operation used in computational science simulations and data science domain. It is inherently imbalanced due to the following attributes:

1. The rows and columns of the matrices are tiled nonuniformly due to the nonuniform structure of the physical problem it represents.
2. The matrices are block-sparse, with varying filling degrees. This means that a tile is either full, where each element is represented in memory, or empty, where all elements are zero.
3. The aspect ratios of the matrices can vary greatly from 1 (matrices will be square) to 100s (matrices will be tall-and-skinny, or short-and-wide).

All these characteristics decrease the potential for data reuse and diminish the arithmetic intensity making it difficult to find a fair static task mapping across the number of available GPUs in a compute node. We also use the Block-Dense GEMM and Block Cholesky Factorization benchmarks to show that task migration can improve the performance of regular, balanced application to some degree and it does not reduce the performance of such an application. We test inter-GPU task simulation on the following scenarios:

1. **Scenario 1:** Matrix A with dimensions $20k \times 200k$ with tile size 200×200. Matrix B of dimension $20k \times 800k$ with tile size $200 \times 8k$. Both matrix A and B has a density of 30% (Synthetic Benchmark).
2. **Scenario 2:** Matrix A with dimensions from $20k \times (10k \cdot \#GPU)$ with tile size $100 \times (100 \cdot \#GPU)$. Matrix B with dimensions $(10k \cdot \#GPU) \times 800k$ with tile size $(100 \cdot \#GPU) \times 100$. Both matrices, A and B, have a density of 30% (Synthetic Benchmark).
3. **Scenario 3:** Matrix A with dimensions $50k \times 100k$ and Matrix B with dimensions $10k \times 500k$, both with tile size $1k \times 1k$. Both matrices A and B are dense(Synthetic Benchmark).
4. **Scenario 4:** We calculate $A \times A^T$, where A is a matrix generated by a real Coupled-Cluster Singles and Doubles method (CCSD) electronic structure model for the matrix representation of the Yukawa integral operator $(exp(-r_{12}/5)/r_{12})$ in the cc-pVDZ-RIFIT Gaussian atomic orbital basis for the main protean of the SARS-CVO-2 virus in complex with the N3 inhibitor [21]. The matrix A has 44×4225 blocks where the size of the block varies between 361×696 and 792×576 and has a sparsity of 30% (Practical Benchmark).
5. **Scenario 5:** PaRSEC runtime allows the application developer to advise the runtime on the preferred GPU to execute a task, which gives the developer a degree of control over the load distribution. Although this advice mechanism can improve performance, it is very application-specific. All the Sparse GEMM scenarios mentioned above were run using this advice mechanism. In Scenario 5, we test how task migration performs when this advice mechanism is not used. Matrix A with dimensions $20k \times 200k$ with tile size 200×200. Matrix B of dimension $20k \times 800k$ with tile size $200 \times 8k$. Both matrices A and B have a density of 30% (Synthetic Benchmark).
6. **Scenario 6:** Matrix A and matrix B with dimensions $100k \times 100k$ both with tile size $1k \times 1k$. Both matrices A and B are dense(Synthetic Benchmark).

5.3. Imbalance

We quantify the imbalance in an application using Eq. (1), where T_{G^i} is the total count of compute tasks executed on the i^{th} GPU, without any task migration. $\sigma(T_{G^1}, T_{G^2}, ..., T_{G^N})$ is the standard deviation between these task count and $\frac{\sum(T_{G^1}, T_{G^2}, ..., T_{G^N})}{N}$ is the mean of these tasks count.

$$I = \frac{\sigma(T_{G^1}, T_{G^2}, ..., T_{G^N})}{\frac{\sum(T_{G^1}, T_{G^2}, ..., T_{G^N})}{N}} \quad (1)$$

Note that this definition of imbalance assumes that all tasks of the same type represent the same amount of work. In Section 5.5, we will show that this assumption does not hold in all cases; in practice, the same kernel with the same data dimensions can have different task execution times. In the above equation, we don't consider the granularity of individual tasks, only the number of tasks.

5.4. Work delegation to a co-manager

At present in PaRSEC, all the stages of GPU tasks are managed by the GPU manager. While most of these stages are related to the GPU the manager will also do task-related bookkeeping (such as identifying successors for tasks completed on the GPU, triggering communications, and so on). This extra work prevents the manager from completely focusing on the management of a GPU device and instead forces it to do potentially time-consuming work that might impact the GPU occupancy. To test if delegating some work to a co-manager is necessary and efficient, we compare the performance of the Block-Sparse GEMM benchmark when epilogue completion and task

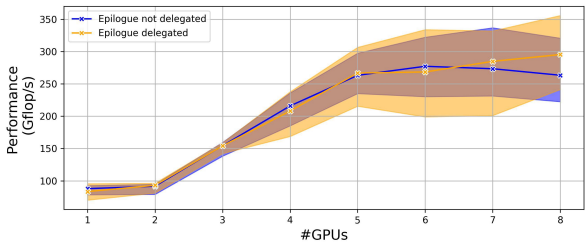

Figure 3: Performance of Block-Sparse GEMM, without migration, with and without delegating epilogue. Tested in scenario 1.

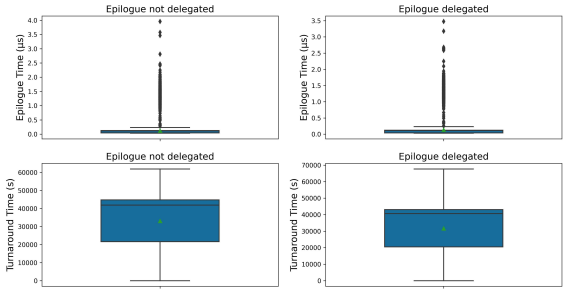

Figure 4: Performance of Block-Sparse GEMM (without migration) when epilogue completion is delegated and when it's not. Tested in scenario 1 with 4 GPUs. In the graph, *epilogue time* is the time taken for the manager/co-manager to execute the epilogue and the *turnaround time* is the time interval between when the task was first received by the manager and when all operation, including epilogue completion, was completed by manager/co-manager. The graph is a boxplot, where the solid box shows the inter-quartile range and the whiskers show the range of non-outlier values. The line across the box shows the median value, while the triangle shows the mean value.

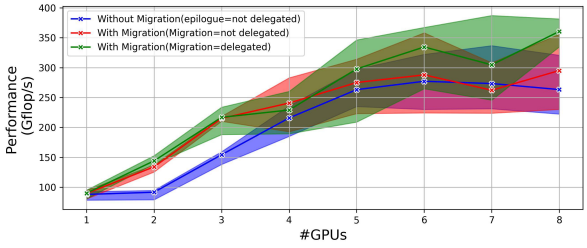

Figure 5: Performance of Sparse GEMM, with migration, when migration is delegated and when it's not. The performance is compared against Sparse GEMM performance without migration and without any epilogue delegation. Tested on scenario 1.

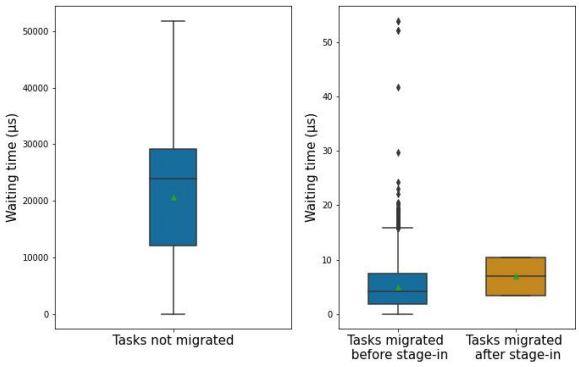

Figure 6: Waiting time for tasks that were migrated and tasks not migrated. Tested on scenario 1 with 4 GPUs. We calculate the waiting time as the interval between when a task was first mapped to a GPU and when the data stage-in for the task begins in the same GPU.

migration are delegated to a separate co-manager. We tested the following scenarios:

1. Without task migration, where the epilogue of a task is completed by the manager thread.
2. Without task migration, where the epilogue of the task is delegated to the co-manager thread.
3. With migration, where migration is undertaken by the manager thread.
4. With migration, where migration is delegated to the co-manager thread.

From Fig. 3 we can see that delegating epilogue completion to a co-manager does not seem to affect the performance for a smaller number of GPUs ($\#GPU \leq 6$). However, when the number of GPUs increases ($\#GPU > 6$), delegating epilogue completion shows an increase in performance. Without access to more GPUs per node, we cannot conclusively say that delegating epilogue completion of a co-manager will improve performance when the $\#GPU > 6$. When we profiled the tasks, we observed that the epilogue completion time and the overall turnaround time of the tasks are similar, irrespective of whether task completion is delegated or not. Furthermore, epilogue completion remains a small percentage of the overall turnaround time of the task. Thus, in these scenarios, the delegation of epilogue completion does not show performance improvement (Fig. 4). We believe that when an application has numerous tasks with a large number of successor tasks, the time taken to complete the epilogue will become non-trivial. In such cases, delegating the epilogue completion will significantly impact the overall performance.

Fig. 5 shows the performance of the Sparse GEMM benchmark for different design choices related to delegating task migration when compared to existing PaRSEC behavior. From the figure, we can see that task migration improves performance, especially when the task migration is delegated to the co-manager. Migration shows this improvement because migrating tasks reduces the waiting time of tasks, as shown in Fig. 6. We calculate the waiting time as the time interval between when a task was first mapped to a GPU and when the data stage-in for the task begins in the same GPU. When we measured the waiting time, we found that the average waiting time for a task that was not migrated was 100 times more than the average waiting time of a migrated task on the GPU to which it was migrated.

Delegating task migration to a co-manager thread improves the performance, especially as the number of GPUs increases because the manager thread can proceed with the progression of other tasks while task migration is taken care of by the co-

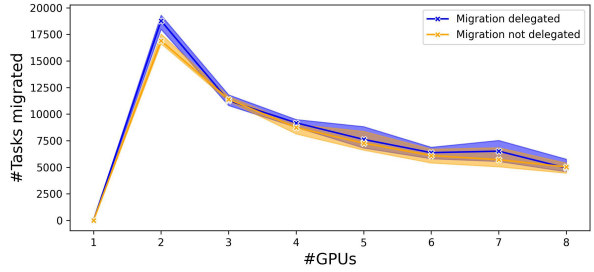

Figure 7: Number of tasks migrated with and without migration delegation. Tested on scenario 1.

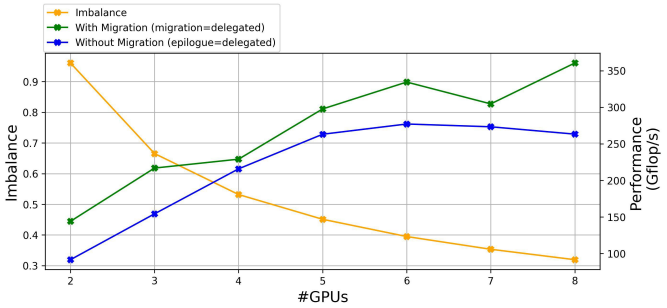

Figure 8: Imbalance calculated when tasks are not migrated and speedup achieved for migration with migration delegation when compared to without migration without epilogue delegation. Tested on scenario 1.

manager thread. Also, when we counted the number of tasks migrated (Fig. 7), it remained almost the same irrespective of whether the migration was delegated or not. Thus the performance of improvement of delegation does not come from migrating more tasks but allowing the progression of tasks to proceed with less hindrance.

Fig. 8 gives the performance of the BSpGEMM benchmarks under scenario 1, both with and without migration, in relation to the degree of imbalance. The performance metric represents the mean of all observed values from Fig. 5. The figure shows that without migration, the performance levels off after engaging 6 GPUs. Conversely with migration, scalability improves notably. In all cases, runs with migration consistently deliver superior performance compared to those without migration. From the figure we can see the most performance gain is achieved when the overall imbalance is high. However, the correlation between the imbalance and performance is not perfect. This discrepancy arises from the fact that imbalance is calculated based on the final task mapping to the GPUs (without migration), while the actual migration and subsequent performance gains are based on the instantaneous load imbalance.

We also tested the weak scaling performance of migration, Fig. 9. From the figure, we can see that delegating task migration to a co-manager thread always performs well, while not delegating migration results in erratic behaviour.

From these experiments, we conclude that delegating epilogue completion to a co-manager does not conclusively show performance improvement. At the same time, delegating task migration to a co-manager shows performance improvement.

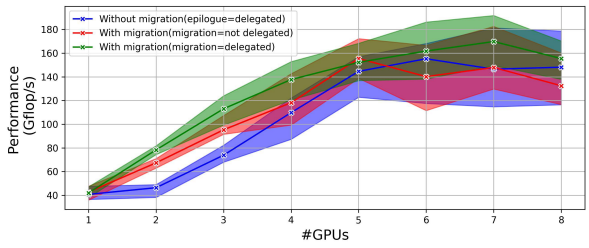

Figure 9: Weak scaling performance of Sparse GEMM. Tested on scenario 2.

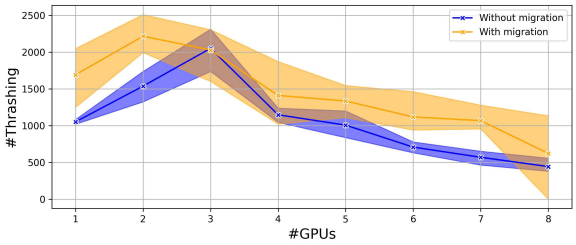

Figure 10: #Thrashing migrated with and without migration delegation. #Thrashing gives the number of data with write access evicted from the GPU. Tested on scenario 1.

5.5. *Performance Variation*

As we can see from Fig. 5 and Fig. 9, there is a significant variation in performance irrespective of whether we are using task migration or not. One aspect that influences performance, irrespective of whether tasks are migrated or not, is the task execution time. From profiling the application, we understood that there is significant variation in the execution time of tasks with the same granularity ($1.5\mu s - 20\mu s$ on average, in some instances it goes up to $200\mu s$, Fig 11). While the number of tasks with significant variation in execution time is small, this can still hinder performance. The dynamic nature of migration adds further variation as different tasks are migrated in each run.

Another factor influencing the performance is memory thrashing in the GPU. When operating under out-of-core conditions (the GPU memory is too small to hold the memory footprint of the application), the runtime system implements an eviction policy that ensures that normal operation can proceed. The policy implemented by PaRSEC is based on Least-Recently Used. The runtime system maintains two lists sorted in the LRU order. The first list holds all read-only data (or in general all data that has an up-to-date version on RAM or another device), and the second holds data that has been modified and that do not have another up-to-date copy on another device. When memory is required on the GPU, the least recently read-only data are evicted, and asynchronous updates of the least recently used modified data are scheduled. When an update completes, the corresponding data moves from the modified LRU to the read-only LRU. When no read-only data can be released, the GPU enters a thrashing mode by evicting data with write access and waits for the completion of tasks or data update operations to continue to the next operation. As Fig. 10 shows, thrashing increases when we are migrating tasks, which

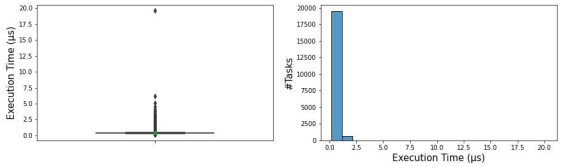

Figure 11: Task execution time for GEMM task with the same tile sizes. Tested on scenario 1 with 4 GPUs.

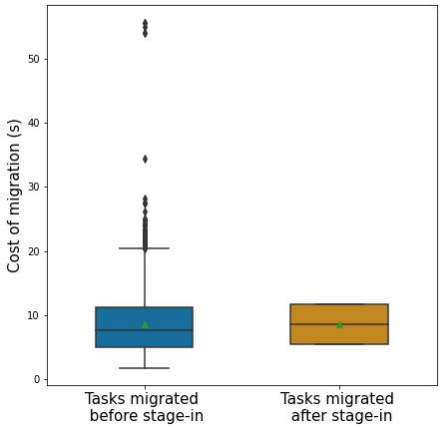

Figure 12: Cost of migration for task migrated before and after stage-in. Tested on scenario 1 with 4 GPUs.

hurts performance.

5.6. Task Type

Another aspect of task migration we explored is the types of tasks we can migrate. In previous works, only tasks whose data had not been staged-in in a GPU were migrated away from that GPU. In our work, in addition to migrating tasks that have not been staged-in, we also migrated tasks whose data were already staged-in to a GPU. We calculated the cost of migration as the time interval between when the task was selected in the source GPU for migration and when all its data were staged-in to the destination GPU. When we measured the cost of migration, we saw that there was not much difference between the average cost of migrating different types of tasks (Fig. 12). This is because the major contribution to the cost of migration is the stage-in cost. When we migrate tasks whose data has not been staged-in, we may have to do both host-to-device (H2D) data transfer as well as device-to-device (D2D) data transfer (from our experiments, we see that in case of tasks that have not been staged-in, we have had a high percentage of H2D data transfer). On the other hand, tasks whose data has already been staged-in the data transfers are exclusively D2D. When we measured the cost of data transfer, we found that it is less for tasks whose data are already available on some GPU, due to the bandwidth of NVLink (Fig 13). Due to these reasons, we conclude that we may migrate tasks irrespective of whether their data were staged-in.

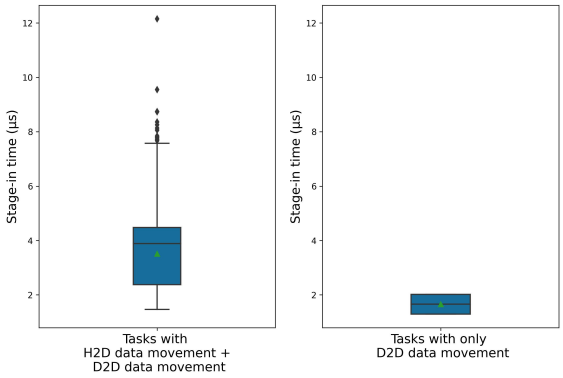

Figure 13: Stage-in cost of task with both H2D and D2D data movements and tasks with only D2D data movements. Tested on scenario 1 with 4 GPUs.

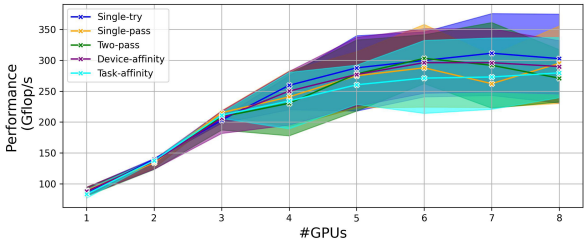

Figure 14: Performance of different selection policies. Tested on scenario 1.

5.7. Selection policies

Fig. 14 shows the performance of different selection policies for the Sparse GEMM benchmark. From the figure, we see that Single-try gives a slightly better average performance than the rest, while Task-affinity gives the worst performance. Single-try gives the best performance even when the tasks it migrates are less than the rest of the selection policies (Fig. 15). This is because it holds the lock on the shared queues only while examining the first task. All the other policies hold the lock until a required task is found or all the tasks in the queue have been examined. During this lock, the progression of other tasks is stalled. The performance is worst from Task-affinity as it is difficult to find tasks that operate on the same data, and a lot of clock cycles are wasted looking for tasks. So while migrating tasks is important for better performance, we can conclude that it is also important not to hold up the progression of other tasks.

5.8. Chunk Size

We also tested to see if chunk size affects performance. From Fig. 16, we can see no direct correlation between performance and chunk size. Neither is there a correlation between chunk size and average task migrated. This result is in contrast with distributed load balancing, where chunk size plays an influential role in performance [22]. We are not claiming that migrating a chunk of tasks is entirely pointless; rather, we claim that it will not work in a runtime like PaRSEC. In PaRSEC, each data movement is separate. For instance, if a task requires

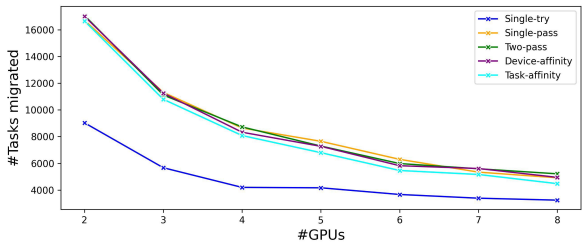

Figure 15: Number of tasks migrated for different selection policies. Tested on scenario 1.

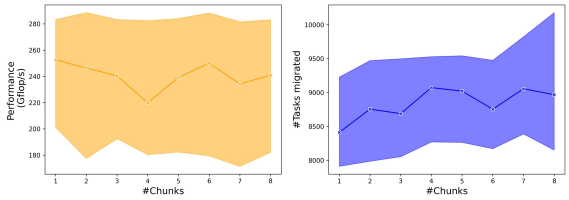

Figure 16: Impact of chunk size(selection policy Single-pass, #GPU=4). Tested on scenario 1.

three data copies, we will have to do three separate data transfers into the GPU that will execute the tasks. At present, in PaRSEC, there is no option to batch these together into a single data transfer. Similarly, when we migrate a chunk of tasks, we are not batching the data transfers together. This lack of batching may be why migrating a chunk of tasks does not seem to affect performance.

5.9. Without advice on preferred GPU

In previous scenarios, the application developer provided strong hints for load balancing via the preferred GPU mechanism. To evaluate how automatic migration can simplify the development of multi-GPU applications, we now consider scenarios where the programmer does not provide these hints. Tasks are mapped entirely based on PaRSEC's locality and load-based algorithm without any input from the developer. From Fig. 17(a), we can see that the advice does not work in all instances, and in some cases, actually worsens performance. We can also see that with (Fig. 17(b)) and without the advice (Fig. 17(c)), dynamic task migration gives better performance. So with dynamic task migration, the developer can design an application without worrying about the load imbalance of the application.

One of the main challenges of building multi-GPU applications is designing the application such that the work is fairly distributed among the available GPUs. In the past, these load-balancing decisions had to be made by the application developer. PaRSEC solved this to a point where the decision is made by the runtime with some input from the application developer. Dynamic task migration takes this further where load balancing is achieved without any input from the application developer and moreover, this does not require any additional programming effort from the application developer. In addition, the load

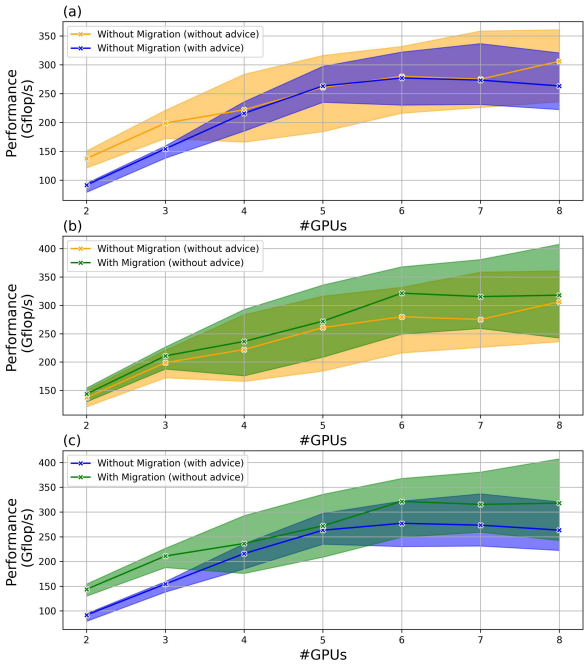

Figure 17: Performance of migration without device advice, with both epilogue completion and migration delegated to co-manager. Tested on scenario 5.

balancing is not bound by the type of tasks as it will work for any kind of tasks without any additional programming effort.

5.10. Without load-based mapping

PaRSEC employs a load-based task mapping when the locality-based mapping fails. To do this, PaRSEC keeps track of the load in each GPU and when PaRSEC cannot find a locality-based mapping for a task it maps the tasks to the least loaded GPU. We experimented to see if dynamic task migration can replace this load-based task mapping. To do this we mapped the tasks to the first GPU if the locality-based mapping fails. From Fig. 18 we can see that dynamic task migration cannot replace load-based task mapping, moreover, it performs better when employed alongside load-based task mapping.

At the same time, we can see that when load-based mapping is not used, the runs with migration vastly outperform the runs without migration, irrespective of the number of GPUs employed. From this experiment, we conclude that the performance of an application that does not employ task migration will largely depend on the load calculation used in the underlying runtime. To test this beyond this experiment, we need to test migration on a more imbalanced application and observe its effects on different numbers of GPUs. Unfortunately, there aren't any other applications implemented in PaRSEC that have an imbalance between GPUs in the node, and implementing such a real-world application in PaRSEC is beyond the scope of this work.

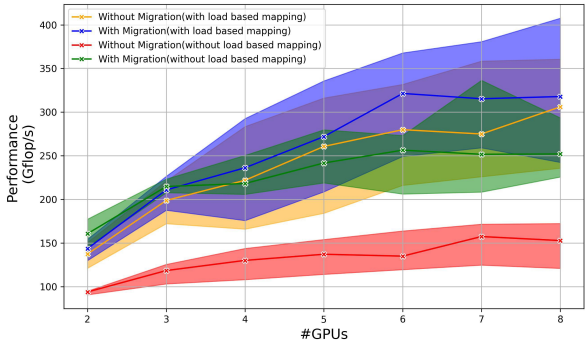

Figure 18: Performance of migration without load-based task mapping, with both epilogue completion and migration delegated to co-manager. Tested on scenario 1.

#GPU	Without Migration (Gflops/s)	With Migration (Gflops/s)
2	10213 - 10241	10221 - 10235
4	15805 - 15873	15396 - 15877
8	18037 - 18109	18002 - 18080

Table 1: Performance of Dense GEMM benchmark with and without task migration. Tested on scenario 3.

5.11. Regular Application

As explained in Section 3.4 PaRSEC uses a locality-based algorithm to map tasks to GPUs. If a GPU already holds some data required by the task in its memory, then the task is assigned to that GPU. If none of the task's data are currently held by any GPU, the task is assigned to a GPU based on the user's advice or the GPU's estimated load. If this load calculation is wrong, the task mapping can be imbalanced. In PaRSEC the user can also set a *load skew*. For instance, a load skew of 20% means that PaRSEC will schedule tasks on the preferred GPU or the GPU that has the task data except if it is loaded 1.2 times as much as the best load balance option. We used the load skew value to compare how task migration can be helpful when the load calculation goes wrong.

Tables 1, 2, and 3 show the performance of a regular application with and without task migration. For regular applications, we chose the dense GEMM and dense Cholesky factorization from the DPLASMA suite [23]. DPLASMA is a dense linear algebra package for distributed, accelerated, heterogeneous systems implemented using PaRSEC. It ports the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) algorithms to distributed memory.

Table 1 shows the performance of dense GEMM benchmark with and without task migration. From the table, we can see that there is not much difference between the performance of both. On the other hand, the situation is different for dense Cholesky benchmark. Table 2 shows the performance of dense Cholesky benchmark when there is no load skew. From the table, we can

#GPU	Normal Load	
	Without Migration	With Migration
2	8590 - 9218	9198 - 9341
3	11634 - 11963	13384 - 13688
4	14200 - 14805	15535 - 15615
5	15400 - 16148	17374 - 17537
6	16207 - 16878	17900 - 18074
7	16822 - 18321	18690 - 18954
8	17969 - 18556	18316 - 18590

Table 2: Performance of Dense Cholesky benchmark, with and without task migration, when load calculation is normal. Tested on scenario 6.

#GPU	Skewed Load	
	Without Migration (Gflops/s)	With Migration (Gflops/s)
2	7476 - 7742	9569 - 9928
3	9534 - 9760	13370 - 13643
4	12479 - 13544	15611 - 15798
5	14699 - 16850	17355 - 17825
6	17481 - 18089	18015 - 18239
7	17509 - 18235	19021 - 19216
8	17704 - 18518	18705 - 18899

Table 3: Performance of Dense Cholesky benchmark, with and without task migration, when load calculation is skewed. Tested on scenario 6.

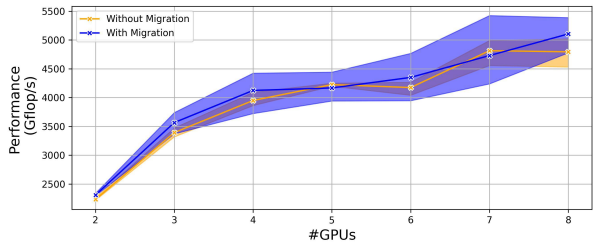

Figure 19: Performance of task migration for the BSpGEMM benchmark on a real-world sparse matrix generated by a Coupled-Cluster Singles and Doubles method (CCSD) electronic structure model. Both epilogue and migration delegated. Scenario 4.

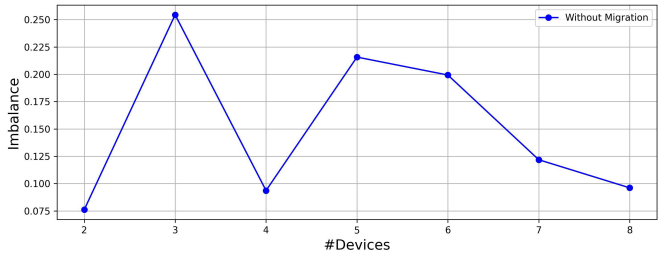

Figure 20: Imbalance of BSpGEMM benchmark on a real world sparse matrix generated by a Coupled-Cluster Singles and Doubles method (CCSD) electronic structure model. Both epilogue and migration delegated. Scenario 4.

see that the performance is better with task migration. Table 3 shows the performance of dense Cholesky benchmark with 80% load skew. In this case, also, the performance is better with task migration. On the other hand, a skew of 80% did not show much change in the performance of dense GEMM benchmark.

From these experiments, we can conclude that regular applications can also benefit from task migration and while task migration may not improve the performance for some regular applications, it does not degrade the performance. We also conclude that when load calculations are wrong for an application, task migration can improve performance, but the improvement is not on par with a well-balanced application.

5.12. Practical Application

Finally, we also tested task migration when computing $A \times A^T$, where A is a sparse matrix generated by a practical application - Coupled-Cluster Singles and Doubles (CCSD) method electronic structure model. As established earlier, one crucial element impacting performance of load balancing is the imbalance of the task mapping across the different GPUs. Fig 20 calculates this imbalance for Scenario 4 using Eq. (1). From Fig 20, it becomes apparent that the imbalance falls within the range of 0.075 to 0.250 (especially when compared to Scenario 1 detailed in Fig. 8), which is notably small. Consequently, opportunities for achieving load balancing through task migration are also limited due to this reduced level of imbalance.

Another factor influencing the performance is the problem size. The Sparse Matrix generated by the CCSD method is small compared to the synthetic benchmarks, so the number of tasks available for dynamic migration is also small. Even with these limitations, we can see that the performance of the $A \times A^T$ operation is better with task migration (Fig 19), even if the performance gain is small (speedup between 3%-12%).

6. Conclusion

In this paper, we examined whether application performance can be improved by migrating tasks between GPUs, and tested different aspects of migration using Block Sparse GEMM. We showed that dynamic load balancing through work-sharing improves performance if an application is imbalanced. We also showed that the performance of task migration is better when the imbalance in the application is higher and that having a co-manager thread, in addition to a GPU manager thread, to migrate tasks improves performance. Moreover, the performance improvements, through dynamic load balancing between GPUs, were achieved without any additional programming effort from the application developer.

The benchmark we used is most imbalanced when the number of GPUs is fewer than 5, so the performance gain from migration is also greatest for fewer than 5 GPUs. At the same time, when load mapping is not used, dynamic migration gives better performance irrespective of the number of GPUs. We think this is application-dependent: based on the application's DAG and how runtime calculates the load, the imbalance can vary for different applications for different numbers of GPUs. Due to the dearth of imbalanced applications implemented for PaRSEC, we were not able to explore this further.

When choosing tasks to migrate, we found that whether task data are already resident on the GPU is not an important factor. In addition, we showed that selection policies and chunk sizes do not have a significant effect on the performance of task migration, unlike in distributed work stealing. Our experiments also showed that dynamic task migration is a good alternative to load balancing with the help of users' advice on task distribution.

The concepts discussed here are applicable to any runtime that keeps a list of tasks earmarked for GPU assignment but pending scheduling. The runtimes should also necessitate task specifications to describe the data it works on, and where they reside. These conditions are true for most task-based dataflow programming models but not for control task-based control flow programming models. In addition to adhering to the aforementioned conditions, PaRSEC employs its own GPU memory management protocol. This approach proves highly economical, given that the cudaMalloc() function is invoked just once throughout application execution. The disadvantage of this scheme is that memory will be allocated in blocks, and there are situations where more memory will be allocated to tasks than required. This is an acceptable shortcoming when we consider that each task may require N number of data allocations, and there can be thousands or millions of tasks in an application. The cost of migrating a task may be more expensive for a different task-based dataflow model if the memory allocation is done in an ad-hoc manner.

It's important to highlight that in PaRSEC, each data item is treated as an independent entity, even if it's linked to multiple

tasks. It has a separate life cycle that depends on the tasks that operate on it but also on other aspects, such as the number of readers the data has (which can be manipulated independently of the tasks that operate on it). This approach enhances the flexibility of data movement across memory spaces. In contrast, certain task-based dataflow programming models package tasks and data as a unified unit, potentially complicating task migration.

One aspect of CUDA optimization we tried to explore in PaRSEC is CUDA Graphs. As we understand CUDA Graphs, they would be an interesting tool to submit an entire DAG of operations directly onto the GPU. The main challenge in implementing CUDA Graphs in PaRSEC is that CUDA Graphs can only manage dependencies of tasks that run on the same device. In PaRSEC, this is not guaranteed - a successor of a task may have a CPU-only kernel, in which case it cannot be executed on a GPU, or the successor may be mapped to execute on a different GPU. Additionally, in PaRSEC, all the tasks submitted to the GPU are ready-tasks (tasks whose inputs are all ready to be used). If we build a CUDA Graph with the set of tasks ready to execute, that graph will be trivial, as all tasks are independently ready (although some tasks could depend on the same input transfer, no task would depend on the completion of another task). If we take a sub-graph of the task DAG that the scheduler decides will execute on the same device, and there are no other dependencies in input, it would be possible to build a CUDA Graph, but this would involve DAG unrolling. DAG unrolling is a scenario where we store the entire task DAG or a section of the task DAG in memory. As can be imagined, this is a costly operation memory when memory is taken into consideration, and the sub-graph we derive from this DAG unrolling can be very small, negating all the advantages derived from the CUDA Graphs.

We based our decision on the feasibility of migration on the count of available tasks within a GPU. This approach can be readily substituted with a cost model or a heuristic model. Many of the existing cost models are developed around task execution duration, yet this approach's accuracy comes into question, as highlighted in Section 4. For instance, tasks of the same type working on identical data dimensions can exhibit substantial variations in execution times. An alternative approach involves identifying the critical path within the task graph and constructing a cost model centred on this critical path. However, this method also assumes uniform execution times for tasks of the same type. Furthermore, it necessitates foreknowledge of the overall task DAG. Another challenge lies in the application-specific nature of these cost models, demanding updates for varying applications and more work for the application developer. Utilizing task count presents a more universally applicable approach across all applications, even though it might not yield optimal performance for every scenario.

As a future extension of this work, we will explore whether migrating GPU tasks between nodes will improve performance in a distributed setting.

7. Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] J. Dongarra, P. Luszczek, TOP500, Springer US, Boston, MA, 2011, pp. 2055–2057. doi:10.1007/978-0-387-09766-4_157.

[2] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A Generic Distributed DAG Engine for High Performance Computing, Parallel Computing 38 (1-2) (2012) 37–51. doi:10.1016/j.parco.2011.10.003.

[3] Q. Cao, Y. Pei, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, J. Dongarra, Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications, in: Platform for Adv. Scientific Comput. Conf., PASC '20, ACM, 2020. doi:10.1145/3394277.3401846.

[4] C. Zhang, Y. Xu, J. Zhou, Z. Xu, L. Lu, J. Lu, Dynamic Load Balancing on Multi-GPUs System for Big Data Processing, in: International Conference on Automation and Computing (ICAC), 2017. doi:10.23919/IConAC.2017.8082085.

[5] C. Chen, K. Li, A. Ouyang, Z. Zeng, K. Li, Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data, IEEE Transactions on Parallel and Distributed Systems 29 (6) (2018) 1275–1288. doi:10.1109/TPDS.2018.2794343.

[6] T. Gautier, J. V. Lima, N. Maillard, B. Raffin, XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures, in: IPDPS, 2013. doi:10.1109/IPDPS.2013.66.

[7] Y. N. Khalid, M. Aleem, U. Ahmed, M. A. Islam, M. A. Iqbal, Troodon: A Machine-learning Based Load-balancing Application Scheduler for CPU–GPU System, Journal of Parallel and Distributed Computing (2019). doi:10.1016/j.jpdc.2019.05.015.

[8] L. Chen, O. Villa, S. Krishnamoorthy, G. R. Gao, Dynamic Load Balancing on Single- and Multi-GPU Systems, in: IPDPS, 2010. doi:10.1109/IPDPS.2010.5470413.

[9] S. Chatterjee, M. Grossman, A. Sbîrlea, V. Sarkar, Dynamic task parallelism with a GPU work-stealing runtime system, in: Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 203–217. doi:10.1007/978-3-642-36036-7_14.

[10] Z. Tang, L. Du, X. Zhang, L. Yang, K. Li, AEML: An acceleration engine for multi-gpu load-balancing in distributed heterogeneous environment, IEEE Transactions on Computers (2022). doi:10.1109/TC.2021.3084407.

[11] F. Guo, Y. Li, J. C. S. Lui, Y. Xu, DCUDA: Dynamic GPU Scheduling with Live Migration Support, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '19. doi:10.1145/3357223.3362714.

[12] J. V. Lima, T. Gautier, N. Maillard, V. Danjean, Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs, in: International Symposium on Computer Architecture and High Performance Computing, 2012. doi:10.1109/SBAC-PAD.2012.28.

[13] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, in: Euro-Par 2009 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 863–874. doi:10.1007/978-3-642-03869-3_80.

[14] E. Hermann, B. Raffin, F. Faure, T. Gautier, J. Allard, Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations, in: Euro-Par, 2010. doi:10.1007/978-3-642-15291-7_23.

[15] G. Bosilca, A. Bouteiller, T. Hérault, V. Le Fèvre, Y. Robert, J. J. Dongarra, Distributed Termination Detection for HPC Task-Based Environments, Research Report RR-9181, Inria - Research Centre Grenoble – Rhône-Alpes (Jun. 2018).

[16] R. Hoque, T. Herault, G. Bosilca, J. Dongarra, Dynamic Task Discovery in PaRSEC- A Data-flow Task-based Runtime, ScalA'17: Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. (2017). doi:10.1145/3148226.3148233.

[17] G. Bosilca, R. Harrison, T. Hérault, M. Javanmard, P. Nookala, E. F. Valeev, The Template Task Graph (TTG) - An Emerging Practical

Dataflow Programming Paradigm for Scientific Simulation at Extreme Scale, IEEE/ACM 5th Int. Workshop on Extreme Scale Programming Models and Middleware (2020). doi:10.1109/ESPM251964.2020.00011.

[18] T. Herault, Y. Robert, G. Bosilca, J. Dongarra, Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC, in: Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2019. doi:10.1109/ScalA49573.2019.00010.

[19] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, K. J. Barker, Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect, IEEE Transactions on Parallel and Distributed Systems (2019). doi:10.1109/TPDS.2019.2928289.

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

[21] Z. Jin, et al., Structure of Mpro from SARS-CoV-2 and discovery of its inhibitors, Nature (582(7811)) (2020) 289–293. doi:10.1038/s41586-020-2223-y.

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

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA, in: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 1432–1441. doi:10.1109/IPDPS.2011.299.