DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Evaluation of Modern PGAS Libraries for Work Stealing in Distributed Memory

Nikola Nincic

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Evaluation of Modern PGAS Libraries for Work Stealing in Distributed Memory

Evaluation Moderner PGAS Bibliotheken für Work Stealing im Verteilten Speicher

Author:	Nikola Nincic
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Philipp Samfaß
Submission Date:	15/07/2021

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 14/07/2021 Nikola Nincic

Acknowledgments

I would like to thank Univ.-Prof. Dr. Bader and Philipp Samfaß for offering me the topic of this work. Further, special thanks go to Minh Thanh Chung and Philipp Samfaß for always supporting me throughout the months and answering all of my questions.

Abstract

The Partitioned Global Address Space (PGAS) is a programming model designed to offer scalable and efficient communication on distributed memory systems. In the last few years several libraries implementing this model have been developed. Two of them are the Berkeley Container Library (BCL) and the Hermes Container Library (HCL), both offering high-level programming constructs for distributed systems. In these systems, computation speed and workload are often times hard to estimate. Therefore, static load balancing does not offer a proper solution to load imbalances anymore. To deal with this problem, different dynamic load balancing approaches, like work stealing, are used. In work stealing, idle processors, called thieves, randomly select other processors, the victims, and attempt to steal work from them. In this thesis, various implementations of this approach are benchmarked with Chameleon, which utilizes push-based reactive load balancing. The results show that both libraries are outperformed by Chameleon, with one implementation in BCL performing close to Chameleon for higher thread counts. However, with several issues present in the two PGAS libraries, some program configurations and approaches to work stealing could not be benchmarked. Once they are fixed, the performances are likely to experience an increase.

Keywords: PGAS, Work Stealing, BCL, HCL, Chameleon, Load Balancing

Contents

1 Introduction

Load imbalances at runtime of a program are a major issue for its efficiency. Imbalances may occur for various reasons, like inconsistent CPU performances or dynamic workload. In order to prevent processors from being idle before all tasks are done, work needs to be rebalanced at runtime. Work stealing achieves that, by allowing idle CPUs to steal work from other CPUs that are part of the system. Making the stealing process one-sided is important, because otherwise the working processor would be interrupted every time it becomes a victim to stealing. BCL [BBY19] and HCL [Dev+20] are both C++ PGAS libraries promising one-sided communication and high-performance data structures. BCL provides different communication backends like MPI, OpenSHMEM [Cha+10], UPC++ [Zhe+14] and GASNet-EX [BH18]. With these, it hopes to achieve efficient communication across various hardware setups. HCL on the other hand relies on Remote Procedure Call (RPC) over Remote Direct Memory Access (RDMA) technology for the backend and is claiming to be 2x to 12x faster than BCL.

1.1 Motivation

With task-based reactive load balancing, Chameleon offers a general solution to the problem of work imbalance. The implementation is using MPI and OpenMP to rebalance the workload of processors within and between compute nodes. Experiments have shown that PGAS models can outperform MPI when used on compatible hardware [Sha+12]. By exploring modern PGAS libraries for work stealing, we hope to find an approach that delivers even better performance for imbalanced programs in distributed memory, than the current version of Chameleon. In my work I evaluated the performance of two libraries, BCL and HCL, after implementing approaches to work stealing in both of them.

1.2 Outline

First, I will cover some related work to provide a better understanding of the topic as a whole. In order to get an insight into Chameleon, BCL and HCL, which are the three libraries that I used for my benchmarks, will be discussed in Chapter 3 and 4. Further, my different implementations in BCL and HCL and issues I had with both libraries are described in Chapter 5. The general structure, as well as some variants of my work stealing approach are covered in this chapter. Next, benchmarks and comparisons between the different implementations are documented and debated. The benchmarks are conducted on two nodes, with the initial setup of one node holding all the tasks and the other one having none. This scenario is then used for one benchmark with load balancing and one without, where the underloaded node stays idle. Afterwards, the results of them are compared to each other to obtain the achieved speedup. Finally, the work is rounded off in the conclusion.

2 Related Work

In this chapter different sources of work that are related to the main topic, will be presented.

UPC++ [Zhe+14], the successor of UPC, is a low-level C++ library that offers an object-oriented approach to the PGAS programming model. The language uses a compiler-free approach to increase portability across platforms, achieve more flexible interoperability with other parallel language extensions and save maintenance and development costs. Using C++ generic programming and template specialization, the library is able to provide enough syntactic sugar for programming idioms, without having to use a custom compiler. Communication between compute nodes is achieved by implementing shared scalars and shared arrays. These two are globally accessible by every process and therefore allow the nodes to share data with each other. A special feature of the library is the availability of multidimensional domains and domain arithmetic, such as summation or multiplication.

GASNet-EX [BH18], an evolution of GASNet, is a portable, open-source, high-performance communication library, mostly used with PGAS runtime systems. The library offers support for multiple network devices and low-level network layers, by providing *conduits*, which are complete implementations of the GASNet API. On top of *conduits* that support specific types of network devices and layers, the library includes portable *conduits* that enable deployment on systems that lack a High Performance Computing (HPC) networking hardware. Improvements of GASNet-EX over GASNet-1 are mostly changes to better support future exascale machines.

OpenSHMEM [Cha+10] is an effort to unify various SHMEM implementations into one library. SHMEM is a communication library used for PGAS style programming. Features include one-sided point-to-point and collective communication, a shared memory view, as well as atomic operations on globally accessible variables. These variables can either be located on the heap of a process or on a special symmetric heap, on which allocation is handled by SHMEM. Synchronization between processes is provided through fences, barriers and *wait* routines. Opposed to lower-level communication libraries, such as GASNet [BH17] or ARMCI [NC99], with its ease of use, the SHMEM

library is aimed at application programmers.

To enable communication and computation overlap, different approaches to message progression are used in HPC. Hoefler et al. [HL08] describe the difference between the three approaches: Manual Progression, Hardware-based Progression and Threads for Message Progression. They further analyze the method of using a progress thread (pthread) for message progression, by comparing the two options of using a shared or a dedicated CPU core for the thread. Variants of the pthread method, can be divided in polling and interrupt based implementations. Results of the work showed that the earlier is only of benefit, when the progression threads each have a dedicated core. In contrast, the later can also be of use in cases where cores are shared between progression and user threads. The authors additionally found that a pthread needs to be scheduled immediately after a network event, in order to achieve the goal of asynchronous progression.

Dinan et al. [Din+09] present a work stealing implementation, that makes use of the PGAS programming model provided by the Aggregate Remote Memory Copy Interface (ARMCI) [NC99]. In the program each process hosts a double-ended queue for storing tasks. A split divides the queues into a private and a public part. Each time the process needs a task to execute, it takes it from the head of the queue. No locking is required in this step, since the process only accesses the private portion. Each queue needs to have a balanced amount of tasks between the private and the public part at all times. Processes enable this, by moving the split towards the tail or the head of the queue accordingly. Once a processor runs out of work, it checks for victim processors to steal work from. This is done by iterating through every process that is part of the program in random order. When a victim is found, the thief steals half of the tasks from the tail of the victims queue. Further optimizations are made, by using spinlocks to allow processes to abort steals. This is of great benefit, when there are many thieves trying to steal tasks from the same victim, which runs out of work before every thief manages to steal tasks.

3 Chameleon

The goal of this chapter is to further describe Chameleon, as it will be used as a reference for benchmark results of my implementations.

3.1 Motivation and Goals

As the workload of programs and the speed of hardware become less and less predictable, modern approaches to reactive load balancing are needed. Chameleon [Kli+20] is a library implementing one possible approach to solving the problem of work imbalance. It is built upon MPI and OpenMP and balances the work load of processors as soon as it detects an imbalance. The library is implemented to achieve five main goals:

- Quickly react to load imbalance
- Decide whether and where to migrate tasks to in a smart manner
- Achieve high computation and communication overlap
- Be easy to integrate into an existing program
- Allow to customize the migration strategy or introspection/load specification, while providing a default behaviour

3.2 Approach

Tasking: The work inside of applications that integrate Chameleon needs to split up into tasks. Tasks are basic units of work that can both be executed locally or on a remote process. They contain an action that needs to be performed and data items that get accessed by the task. The data items can be of type *input*, *output* or *input and output*. Item types define if the data should be sent back to the creator process after being executed (output) or not (input).
Communication: The balancing mechanism Chameleon implements is push-oriented, meaning that processes that have too much load, push their tasks to other ones having too little. The communication is fully non-blocking, which is achieved by using a

dedicated communication thread on a separate core for each rank. To further increase communication and computation overlap, tasks that have been created on remote processes get executed before local tasks. After all tasks have been finished, the processes need to terminate. For that, the continuous communication to exchange load info is utilized, by attaching the amount of outstanding operations per rank to each message. This allows each process to obtain information of how many tasks are left to be computed and therefore when to terminate.

Migration Strategy: In order to obtain high performance, it is of very high importance to know when to offload tasks and which process to choose as a victim. A way of measuring the work load is needed. Chameleon uses the amount of tasks a rank holds as the default metric for this purpose. However, this can lead to poor decisions when tasks are of varying sizes, which is why users are not only able to customize load specification, but also introspection and migration strategy inside the tools interface of the library. At which degree of imbalance the pushing of tasks occurs, is depending on a configurable absolute/relative threshold. In the default configuration, the victim is selected by a sort-based approach. This solves the problem of contention that an intuitive approach, like migrating tasks to the rank with the lowest load, would have. A visualization of the two victim selection strategies can be found in Figures 3.1 and 3.2.

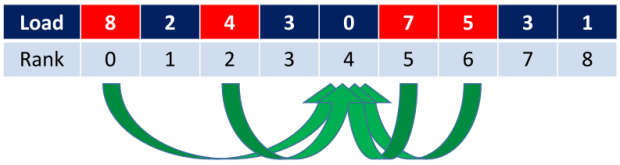


Figure 3.1: Migrate to rank with lowest load [Kli+20]

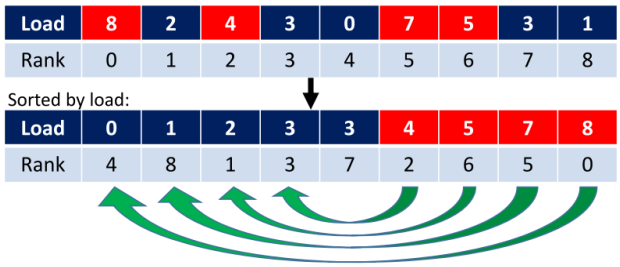


Figure 3.2: Sort-based approach [Kli+20]

4 PGAS Libraries

The work stealing algorithms that are benchmarked in Chapter 6 were implemented using the two modern PGAS libraries BCL and HCL. To offer an improved understanding of the benchmark results, the goals and functionalities of both libraries will be presented in the following sections.

4.1 BCL

4.1.1 Goals and Approach

BCL [BBY19] is a high-level PGAS library that hopes to bring a rich set of data structures to the PGAS programming model, while maintaining performance close to hardware limits. To help them achieve this goal, the authors chose three design-principles:

- Low cost for abstraction: While offering high-level data structures, such as hash tables, BCL hopes to keep performance costs for using them as low as possible.
- Portability: The library can be used with multiple communication backends.
- Software Toolchain Complexity: The library can be used simply by including the appropriate headers, without including new dependencies or re-writing the program.

For the purpose of achieving good performance across multiple hardware setups and fulfilling the second principle, the library offers support for multiple backends. Currently available backends are MPI, OpenSHMEM, GASNet-EX, and UPC++.

4.1.2 Functionalities and Programming Constructs

Shared Memory: In BCL, each process has its own shared memory segment. The segment size is adjustable via a parameter in the initialization method *BCL::init*. The shared memory can be used to allocate globally accessible memory through Global Pointers, which are pointers that point towards global memory. Further, if the hardware

of the system supports it, single Remote Direct Memory Access (RDMA) operations are used for reads and writes on shared memory.
Data Structures: The BCL data structures are divided in distributed and hosted data structures, but all make use of Global Pointers. Hosted data structures are located in the global memory segment of a single process, while distributed ones are, as the name suggests, distributed across the globally addressable memory of all ranks in a program. The different data structures and their locality are presented in Table 4.1. What both types have in common, is the attribute of being coordination free, so that primary operations, such as insertions, reads or deletions do not need to be coordinated with CPUs of other nodes. However, some operations, such as resizing or migrating a hosted data structure between nodes, may still require coordination. Another important attribute of the data structures in BCL, is all of them being generic, making them able to hold data of any type, even non byte-copyable data types. This feature is enabled by BCLs Object Containers. The containers recognize the data type that is stored inside them and automatically use the corresponding serialization and deserialization method, which are crucial for transferring complex data types.
Concurrency Promises: When modifying or reading from a data structure, it is sometimes known that no other process will access the data concurrently. To make use of that knowledge, the mechanism of concurrency promises has been formalized:

- *push* | *pop* allows concurrent pushes and pops during the operation.
- *push* only allows concurrent pushes during the operation. Accordingly, a concurrency promise for pops also exists.
- *local* performs the operation locally, not allowing any concurrent modifications on the data structure

Using an operation that supports them, concurrency promises can be passed as a method attribute. When using promises of lower order, such as *push*, instead of *push* | *pop*, less locking operations need to be performed. This ultimately leads to better performance.

Table 4.1: A summary of BCL data structures [BBY19]

Data Structure	Locality	Description
BCL::HashMap	Distributed	Hash Table
BCL::CircularQueue	Hosted	Multiple Reader/Writer Queue
BCL::FastQueue	Hosted	Multi-Reader or Multi-Writer Queue
BCL::HashMapBuffer	Distributed	Aggregate hash table insertions
BCL::BloomFilter	Distributed	Distributed Bloom filter
BCL::DArray	Distributed	1-D Array
BCL::Array	Hosted	1-D Array on one process

4.2 HCL

4.2.1 Goals and Approach

The Hermes Container Library (HCL) [Dev+20] is a high-performance PGAS library that offers a set of distributed data structures (DDSs), residing in global address space. It uses a Remote Procedure Call (RPC) over RDMA technology to significantly decrease communication time compared to BCL, in which the client executes all necessary instructions to interact with a remote data structure. With the client-side programming model BCL is using, several limitations exist, including increased network congestion, caused by each client making numerous remote calls. Additionally operation concurrency is heavily limited by the memory region locking that "compare-and-swap"-operations are relying on. The RPC protocol supports server-side callbacks, but requires the participation of the remote CPU. HCL faces this issue by utilizing RDMA technology, which allows it to offload RPC instructions to the Network Interface Card (NIC). The library also supports multiple network protocols, like InfiniBand (IB) or Transmission Control Protocol (TCP) by using the Open Fabric Interface (OFI).
Designing and implementing the RPC-over-RDMA protocol, enabled HCL to achieve high performance and coordination free communication for its DDSs. However, for local memory access, no RPC calls are needed. Instead, the hybrid access model allows for a faster Direct Memory Access (DMA).

4.2.2 Functionalities and Programming Constructs

Distributed Data Structures: At initialization, one or more processes in a node can create a shared memory segment, which is then part of the global address space. HCL provides the user with different DDSs that reside inside the global address space and

are accessible by every node. Every DDS has its own use cases, although they all have some common attributes to them:

- Support of complex data types and entries of variable length
- Operations, including collective operations, are lock-free and do not require any global synchronization
- Support of user-defined comparators and operators for defining custom data distributions and/or ordering
- Option of tuning the level of atomicity by setting the appropriate concurrency control parameter
- Support of asynchronous operations

Data Boxes: To obtain the first attribute, the data structures need a way to serialize and deserialize data. HCL deals with this using DataBoxes. "A DataBox is a template that provides mechanisms for defining, serializing, transmitting and storing complex data structures" [Dev+20]. The template provides a standard serialize/deserialize method, which can use different serialization libraries in the backend. This is relevant, due to different serialization libraries excelling in different environments. MSGPACK, Cereal, and FlatBuffers are the three libraries currently supported as a backend for HCL. Additionally, *DataBox Event Handling via Callbacks* enables a conditional execution of multiple operations in one call. This feature minimizes network congestion and, therefore, increases performance.
Another characteristic of the library is the support of data persistence. In HCL, the PGAS model can be extended to include persistent devices, such as NVRAM or a HDD, providing a unified memory and storage address space.

5 Work Stealing Implementations

Using BCL and HCL, three approaches to work stealing have been implemented. All programs use matrix multiplications as tasks, in order to achieve good comparability with the matrix-example of Chameleon-Apps [Jan]. The example allows the user to define the matrix size and the initial task distribution across MPI ranks. Once the tasks are created, the ranks start executing them. During this phase, overloaded ranks push tasks to underloaded ones to achieve a balanced work distribution. The balancing-strategy behind this can be configured using environment variables. Eventually, all ranks will run out of work and the program terminates. Beyond the benefit of comparability, matrix multiplication tasks offer independence and scalability. That means no task is dependent on another tasks data or result and they can be made computation-intensive by setting the matrix size to a large number. Each task consists of a Task-ID and two matrices of same size, which are to be multiplied with one another, before being saved in the form of task results. The exact structure of tasks and task results can be seen in Tables 5.1 and 5.2.
The implementations are presented in two separate sections, due to major differences between the programs in BCL and HCL. These were forced by issues that arised when using multiple queues or multithreading in HCL.

Name	Type
Task-ID	int
matrixA	double[]
matrixB	double[]

Table 5.1: Task Structure

Name	Type
Task-ID	int
matrixR	double[]

Table 5.2: Task Result Structure

5.1 BCL

The Program: As displayed in Figure 5.1, in the BCL program each rank starts by creating and enqueuing a given amount of tasks one by one into its own globally accessible Task Queue. The number of tasks can be defined for each rank individually.

This part of the program is parallelized using an *omp for*-loop. To make sure each task has its own unique Task-ID, the ID is a combination of the creator rank and a counter that gets incremented after each loop-cycle. Having unique IDs even when using multiple threads, was ensured by incrementing the counter by the amount of threads, as shown in Listing 5.1.

Listing 5.1: Task-ID Creation

```
#pragma omp parallel
{
    int nextId = omp_get_thread_num();
#pragma omp for
    for (int i = 0; i < taskAmount; i++)
    {
        task t;
        t.taskId = (int)(BCL::rank());
        t.taskId <<= 24;
        t.taskId += nextId;
        ...
        nextId+=omp_get_num_threads();
    }
}
```

After the Task Queues are ready, multiple threads attempt to pop a task from their queue and execute it in case they succeed. Once the result is computed, it needs to be sent back to the rank that created the task originally. This is done by extracting the origin rank from the Task-ID and pushing the result to the ranks Result Queue. As in the task creation, this part of the program is also parallelized using an *omp for*-loop. The pop-operation fails if the Task Queue is empty. In that scenario, the process needs to secure new tasks by stealing from a victim process in one of two ways. The common practice is stealing one task at a time. However, in some cases attempting to steal half the tasks achieves better performance [Din+09]. For that reason, both stealing methods have been implemented. When stealing one task at a time, each thread is working independently. However, when stealing half the tasks only one thread is executing the method, while the others wait for it to finish. Finally, if the stealing operation succeeds, the process continues executing tasks, otherwise, it waits for other ranks to finish, before eventually terminating.

Issues with the Library: The implementation was successfully compiled using Intel-MPI and GASNet-EX with different conduits. However, only the program compiled with MPI worked as intended, because of an occurring progression issue with GASNet-EX as the backend. When a rank is executing tasks, while other ranks try to steal from it, the stealing gets ignored by the working rank. The only scenario that avoids this issue, is when the ranks are located on the same node. In order to be sure that this is a progression issue, I wrote a simple program that pushes a significant amount of integers to a *BCL::CircularQueue*, located on Rank 0. After obtaining a value from the queue , the fourth root of the integer is computed. This part of the program can be seen in Listing 5.2. The essential question of this test is: Which rank is getting the values? I answered that question for multiple scenarios:

- Only Rank 0 executes the while-loop
- Only Rank 1 (located on a remote node) executes the while-loop
- Both ranks execute the while-loop

The results showed that every scenario worked as intended, apart from the last one. In that case, Rank 0 obtained all values, which makes it very likely that the issue lies in message progression.

Listing 5.2: GASNet-EX Progression Test

```
while (0 < queue.size())
{
        int k;
        queue.pop(k);
        double a = sqrt(sqrt(k));
        printf("[%ld]Calculated %lf\n", BCL::rank(), a);
}
```

Another problem I had to face was that my parallel stealing implementation was not thread-safe, even when initializing the library in the thread-safe mode. Every time the program was executed using more than ten threads for 2400 tasks, it hung up and never finished. A likely reason for this is a deadlock. As I found out after writing a simple program that enforced this scenario, the issue was caused by simultaneous push- and pop-operations on the Task Queue. Although the concurrency promises for the two operations were chosen to use the atomic versions, the test program hung up after using too many threads. These outcomes led me to the conclusion that this might

be an additional issue with the BCL library. To still be able to use the parallel version of task stealing, I added thread locks to the push-operation inside of the stealing method. This led to concurrent *pushes* and *pops* being much less likely and, therefore, allowed me to run the program using more threads than before.
The last issue I want to mention, is that having data types that contain pointers to the heap, leads to a memory leak problem when transferring the data between nodes. The memory allocated at the host node does not get de-allocated after the data is transferred, e.g. by a pop-operation.

5.2 HCL

The Program: The HCL implementation is using multiple MPI ranks instead of OMP-Threads to express parallelism. Additionally one rank functions as the server and is the only rank to host a globally accessible Task Queue and Result Map. Since only one server exists, only one node can be the host of it. Therefore, there are remote and local ranks. Local ranks are the ones located on the same node as the server, remote ranks are the remaining ones. The implementation simulates the remote ranks being out of work from the very beginning, thus, each pop is as a stealing operation. Another difference is the utilization of the *Unordered Map* data structure from HCL for the task results. This change was necessary because of a hanging issue that occurs when using more than one *HCL Queue*.
As presented in Figure 5.2, the task creation is approached in a similar fashion to the approach in the BCL program. One difference is that the tasks are created in a serial manner, using only one rank. The next phase also works very similarly to the other implementation. If the global Task Queue is not empty, tasks are obtained from it, before being executed. Finally, the computed results are put into the Result Map. In case no more tasks are left, the ranks wait for each other to finish processing and then terminate.

Issues with the Library: Aside from the mentioned issues regarding multiple queues, I faced issues when using multithreading. To obtain better comparability between the libraries, I initially planned using OMP-Threads for parallelism. This, however, caused an issue when multiple threads are accessing the same distributed data structure. The last problem I had with HCL was caused by too large matrix sizes. When using a matrix size larger than 208, errors occured when accessing a distributed data structure from a remote node.

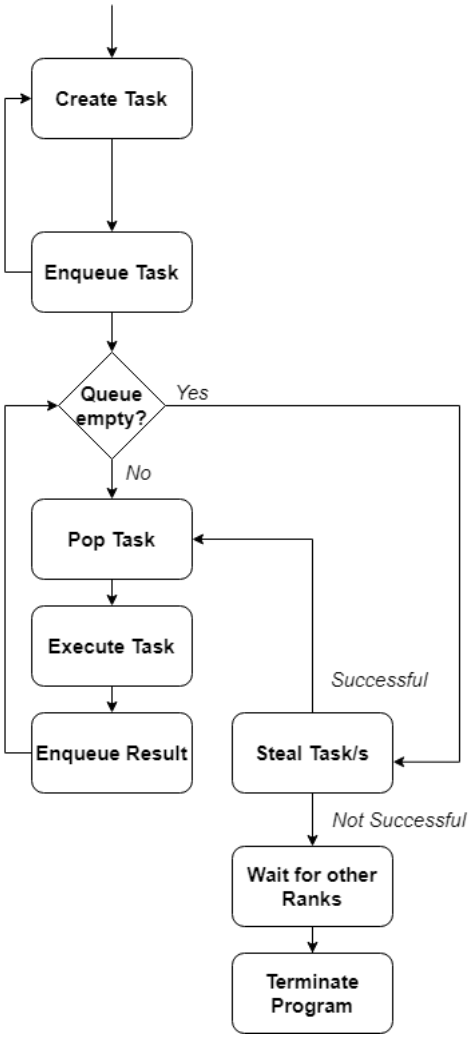


Figure 5.1: BCL Program-Flow of each rank

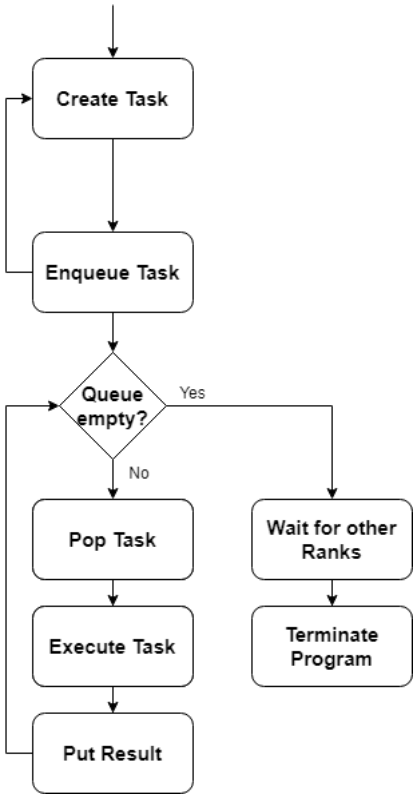


Figure 5.2: HCL Program-Flow of each computation rank

6 Benchmarks

In favor of capturing the potential of these two PGAS libraries, this chapter covers different benchmarks of my programs and the Chameleon application. The results will be interpreted and finally compared to each other. All tests were conducted on the Bavarian Energy-, Architecture- and Software-Testbed (BEAST) Cluster, provided by the Leibniz-Rechenzentrum. The part of the system on which I executed the benchmarks is using modern AMD Rome-Processors with AMD MI50-GPUs.
Comparability between the different implementations was made sure in different ways. First, I set the compiler optimization level to -O0, avoiding any optimization that may influence the results. Next, I measured the average time for a single matrix multiplication to make sure the task execution time is the same for every program. Lastly, I kept the amount and size of messages equal for each implementation.

6.1 Results

A wide range of benchmarks was performed to display the performance of the libraries in various scenarios. Additionally, for BCL, different implementations of the stealing mechanism were used and compared to each other. This allows us to see which kind of task stealing is most effective on the given hardware setup. Also, since we are interested in the efficiency of the load balancing, only the task execution phase is benchmarked. If not explicitly specified, the matrix size is set to 150 (meaning the total amount of cells per matrix amounts to 150x150) and 2400 tasks are initially located on the first node, while zero tasks are located on the second one. Plots without given thread count are created using 12 threads/node. Also, every benchmark is executed three times on two identical nodes. In order to further even out any possible performance fluctuations, every run iterates five times over the whole code and takes the average results. I additionally implemented separate programs that specify the amount of time spent in different areas, which enabled me to better interpret the results. A sample output of this for one rank is presented in Figure 6.1.

```
[1]Cumulative benchmarks of all threads:
   ---------------------------------
   Processor time:        8.623438
   ---------------------------------
   Multiplication time:    6.051826 = 70.178817%
   Idle time:              2.571611 = 29.821183%
   ---------------------------------
   Communication time:     2.364361 = 27.417844%
   Lock time:              0.175902 = 2.039813%
   Other:                  0.031348 = 0.363526%
   ---------------------------------
   Tasks stolen:           309
Total execution time: 2.154174
```

Figure 6.1: Sample Output of Statistical Benchmark

BCL Benchmarks: The graphs shown in Figure 6.2 and 6.3 display the performance of the baseline, where only one node operates, and the sequential/parallel stealing strategy of BCL. In the sequential stealing method one thread attempts to steal half the tasks from the other node, once its queue is empty. In the parallel one, each thread acts individually and steals one task when out of work. The graphs of this stealing strategy stop after a thread count of 16, because of the threading issues in BCL that have been mentioned in the previous chapter.
Observing the graph describing the speedup of sequential stealing, it is visible that one thread/node outperforms the doubled amount of threads. Due to the long execution time when using as little as one thread, the underloaded node has a lot of time to steal tasks from the remote node. More tasks get stolen, therefore, the workload is more balanced. After this point, the speedup is rising until 8 threads/node, before it starts falling again. To better understand this course of the graph, I analyzed the results with my statistical benchmark program. The benchmark showed that a growing amount of threads means less time per stolen task is spent in communication. Although, as I mentioned earlier, more threads also mean less time for task stealing. These two factors together lead to the peak at 8 threads/node, since this is when the negatives of more threads outweigh the positives.
In the parallel approach a similar pattern is occurring. The main reason the speedup is declining after 8 threads/node here, is that the lock in the stealing method is increasingly contended. The rising speedup up to a thread amount of 4 is, as in sequential stealing, produced by the lower communication time.
Further, we can observe that parallel stealing is generally outperforming sequential stealing. This is caused by all remaining threads having to wait for a single thread to finish stealing in the sequential approach, while all threads are permanently active in the parallel one.

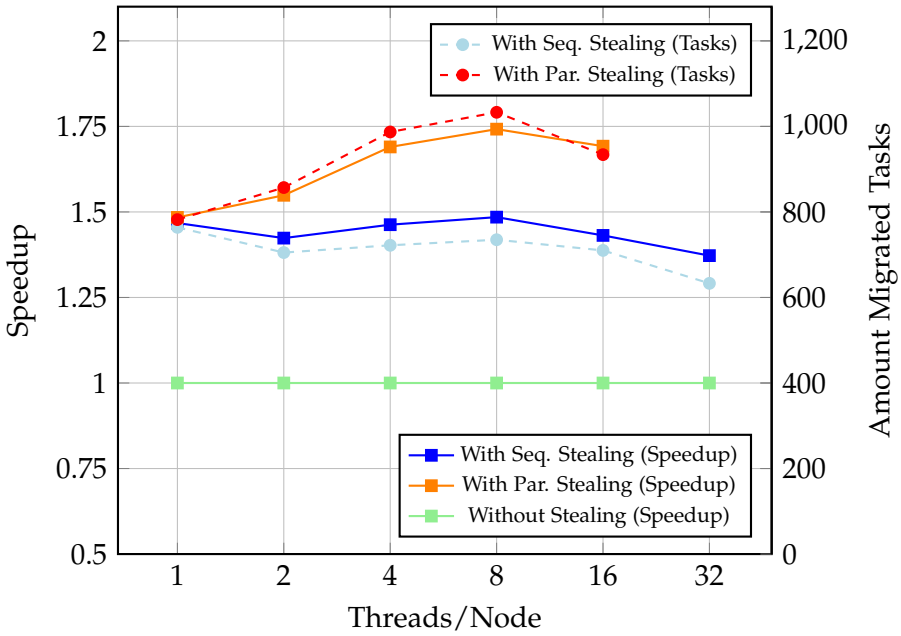


Figure 6.2: BCL Speedup for different thread counts.

Looking at the graphs in Figure 6.4 it becomes clear that the stealing performs better with bigger tasks. The reason for this lies in the relation between computation and communication time. With growing tasks, the computation time is growing too. This leads to the communication time being smaller in comparison, ultimately resulting in a higher speedup.

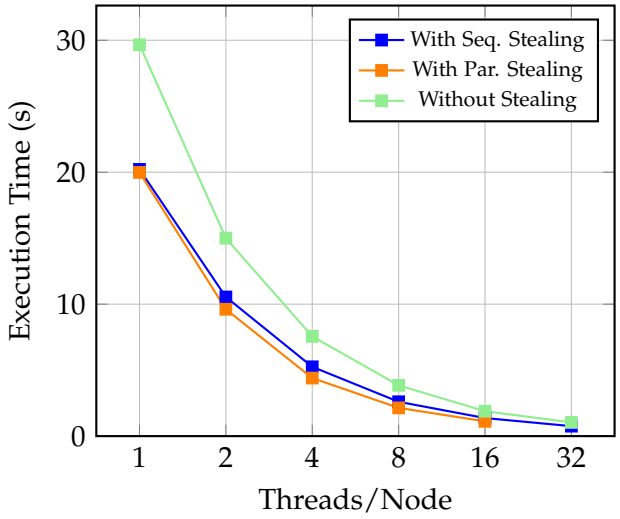


Figure 6.3: BCL Execution Times for different thread counts.

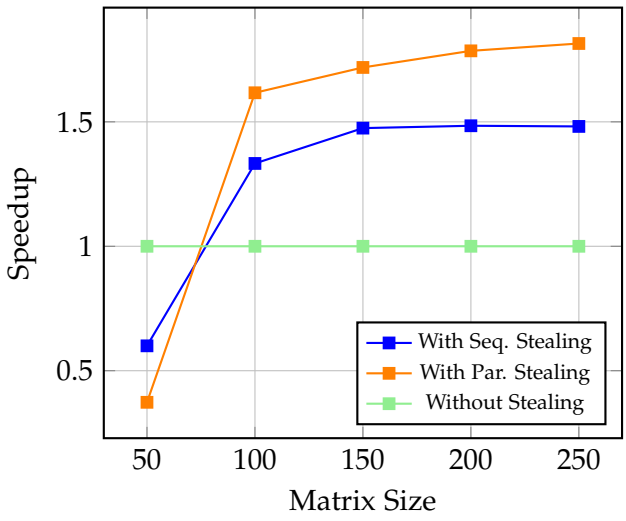


Figure 6.4: BCL Speedup for different matrix sizes.

HCL Benchmarks: The graphs in Figure 6.5 and 6.6 present the performance of my HCL work stealing implementation and compare it to the baseline, where one node executes all the tasks. The program performs well for a low amount of ranks, but rather poorly for higher ones. A reason for this is that tasks get executed faster with more ranks, although the communication time is not changing. In addition to this, HCL has only one communication rank, which can become a bottleneck, once more ranks are active. This results in less tasks being stolen and ultimately less speedup.
The plot in Figure 6.7 shows similar results as the corresponding plot in BCL. The reasoning is the same: Higher computation time leading to a more favorable computation/communication time relation.

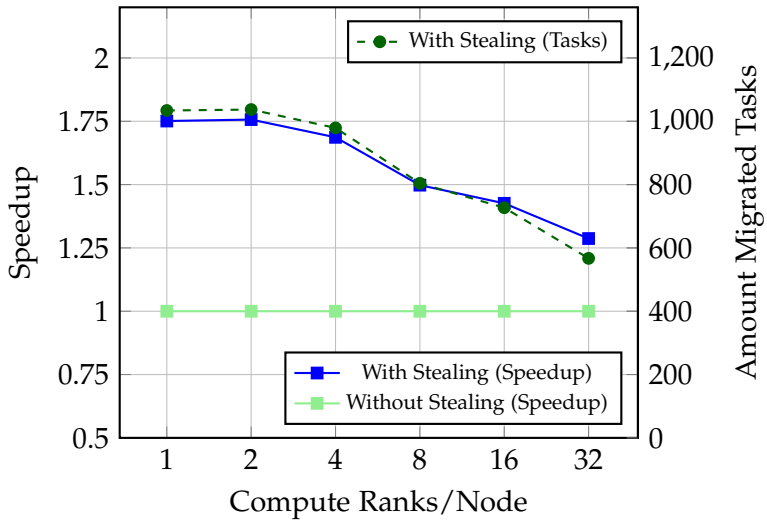
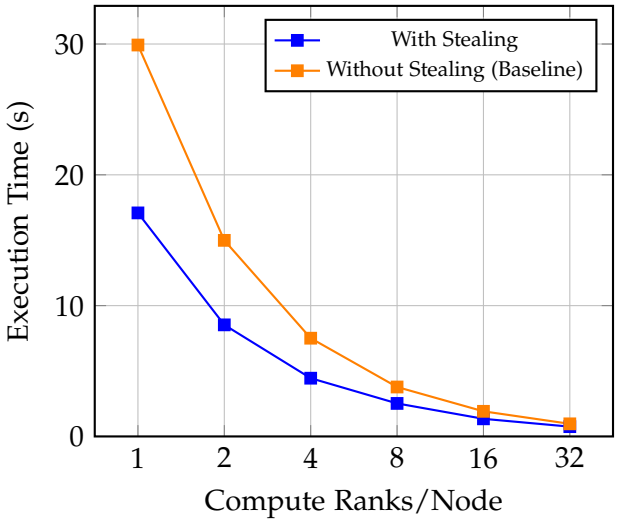


Figure 6.5: HCL Speedup for different computation rank counts.



1

Figure 6.6: HCL Execution Times for different computation rank counts.

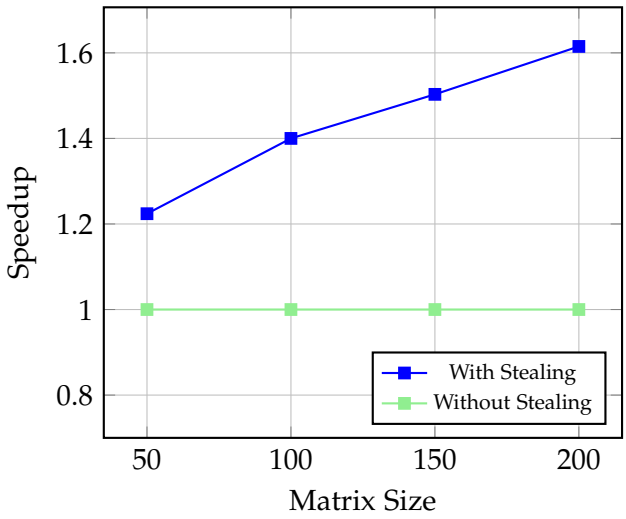


Figure 6.7: HCL Speedup for different matrix sizes.

Chameleon Benchmarks: For the Chameleon benchmarks I used the provided matrix-example of Chameleon-Apps [Jan]. After some testing, I found the following configuration to consistently deliver good performances:

- MIN_REL_LOAD_IMBALANCE_BEFORE_MIGRATION=0.5
- MAX_TASKS_PER_RANK_TO_ACTIVATE_AT_ONCE=1
- PERCENTAGE_DIFF_TASKS_TO_MIGRATE=0.5
- MIN_LOCAL_TASKS_IN_QUEUE_BEFORE_MIGRATION=2**threads/rank*

In the plot of Figure 6.8 we can see the load balancing model of Chameleon compared to the baseline without balancing. For smaller thread counts Chameleon achieves speedups close to the optimal 2x. The processors have enough time to reach a balanced distribution of work, because of the slow task execution when using only one thread. This, however, changes when the thread count increases. The threads rapidly execute the tasks and, therefore, leave little time for balancing, which leads to a decreased speedup.
In Figure 6.10 a similar speedup course to the corresponding one in BCL and HCL is observable. Interestingly, the performance for a size of 50 seems to be better than the one for a matrix size of 100. A possible cause is given by lower communication times for smaller tasks.

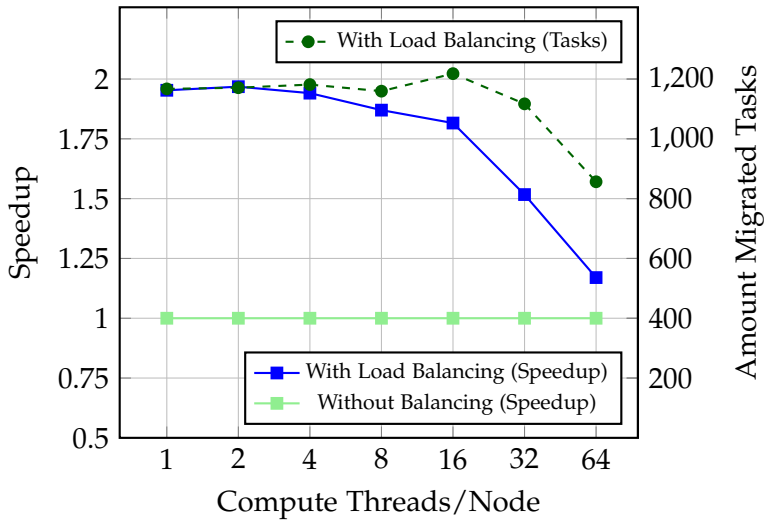


Figure 6.8: Chameleon Speedup for different computation thread counts.

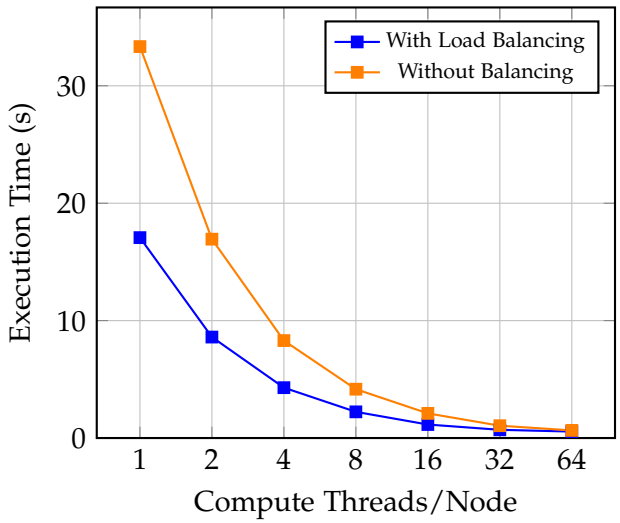


Figure 6.9: Chameleon Execution Times for different computation thread counts.

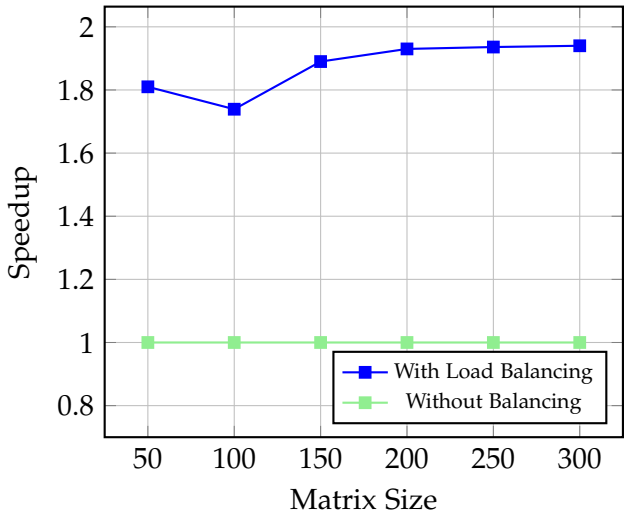


Figure 6.10: Chameleon Speedup for different matrix sizes.

6.2 Comparison

The benchmarks presented in the previous section show a noticeable discrepancy in performance between the different implementations. To better comprehend in which scenarios an approach outperforms another, I will perform a direct comparison in this section. Aside from the benchmarks in the last section, the comparison will include a throughput benchmark, which describes the average data throughput when performing task migrations between nodes.

Multithreading: In Figure 6.11 we can see that Chameleon outperforms HCL at any given thread count and they both rapidly lose on performance at higher counts. Nevertheless, both BCL implementations are able to hold their speedup, even for a higher number of threads. A reason for this occurrence is that BCL with MPI backend is achieving higher data throughput with more threads (see Figure 6.12), which is not the case for the other libraries. The advantage for BCL comes from not having a communication thread that can become a bottleneck when more threads are active. This allows BCL to not only get increased computation speed with more threads, but also to perform faster task steals.

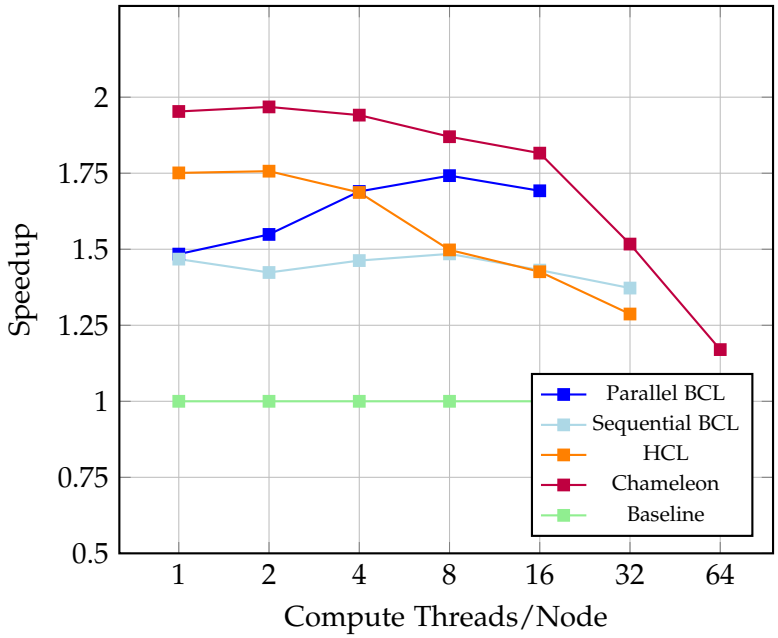


Figure 6.11: Speedup Comparison for different thread (in HCL: rank) counts.

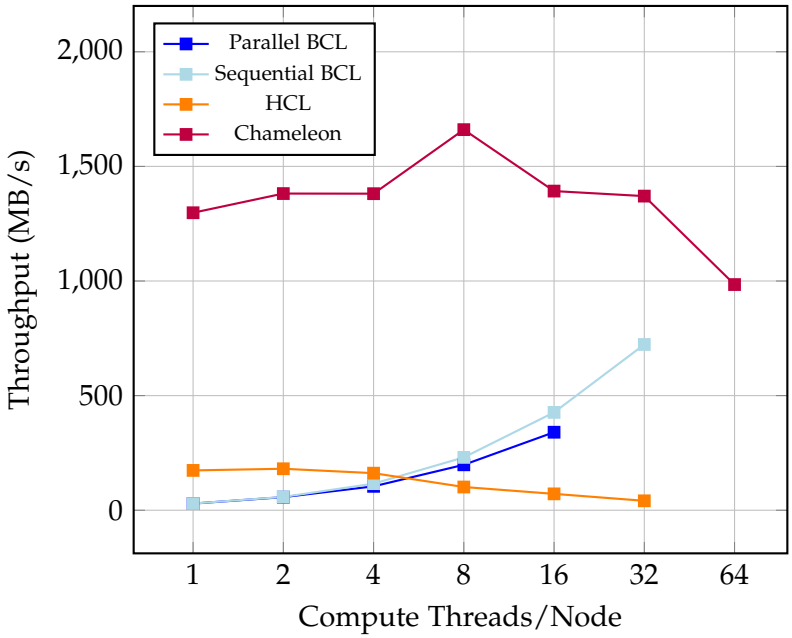


Figure 6.12: Throughput Comparison for different thread (in HCL: rank) counts.

Matrix Sizes: Figure 6.13 shows once again that all approaches profit from larger matrices. The speedup is also strongly correlated to the throughput, which can be observed when comparing the speedup plot to the one in Figure 6.14. The only point at which the correlation seems slightly off, is at the matrix size of 50. At this size tasks get executed too fast, which leaves too little time for migration.

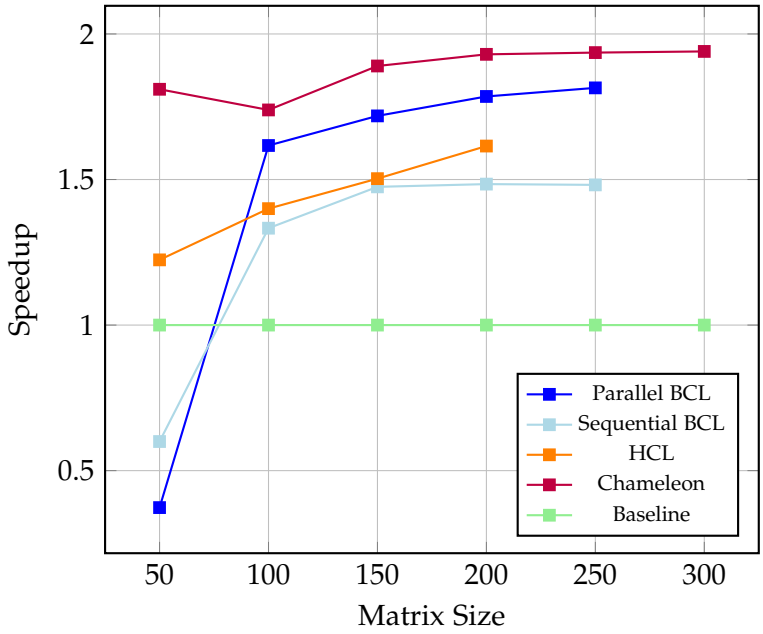


Figure 6.13: Speedup Comparison (12 threads/ranks per node) for different matrix sizes.

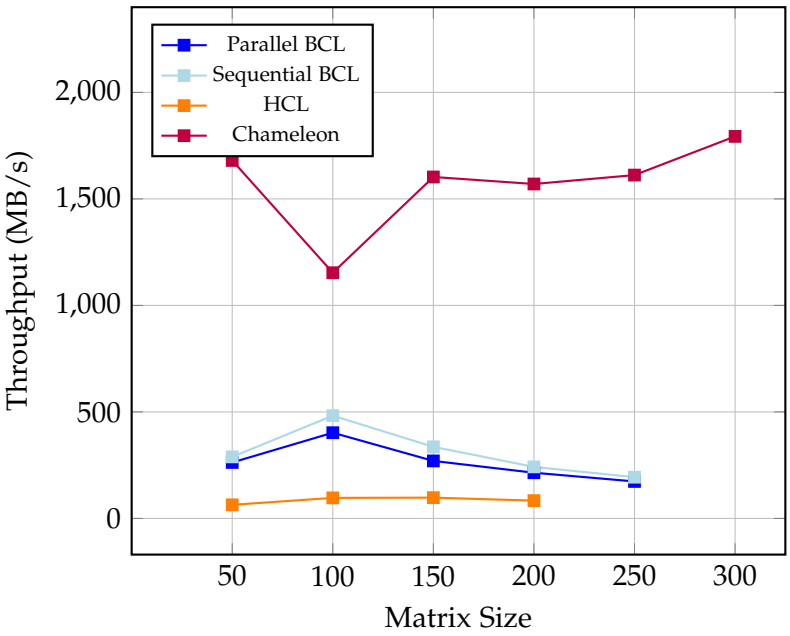


Figure 6.14: Throughput Comparison (12 threads/ranks per node) for different matrix sizes.

7 Conclusion

The goal of this thesis was to explore and evaluate modern PGAS libraries for work stealing in distributed memory. By implementing different approaches to this load balancing method in BCL and HCL, I was able to gather data about the two PGAS libraries. Investing a lot of time in exploring their capabilities, I found that they both offer easy-to-use distributed data structures, while maintaining decent performance. I also discovered that parallel work stealing, as implemented in BCL, can achieve performances that compete with Chameleons load balancing approach. However, both libraries are still pretty young, leaving some issues to be fixed, as demonstrated in Chapter 5. With further refinement they could become a very powerful and scalable tool for future work stealing implementations. Finally, I believe that the PGAS programming model is an important technology to support coming exascale architectures, since scalability will play an important role for these systems.

List of Figures

List of Tables

Bibliography

[BBY19] B. Brock, A. Buluç, and K. Yelick. "BCL: A cross-platform distributed data structures library." In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[BH17] D. Bonachea and P. Hargrove. *GASNet Specification, v1. 8.1*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2017.

[BH18] D. Bonachea and P. H. Hargrove. "GASNet-EX: A high-performance, portable communication library for exascale." In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2018, pp. 138–158.

[Cha+10] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. "Introducing OpenSHMEM: SHMEM for the PGAS community." In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 2010, pp. 1–3.

[Dev+20] H. Devarajan, A. Kougkas, K. Bateman, and X.-H. Sun. "HCL: Distributing Parallel Data Structures in Extreme Scales." In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2020, pp. 248–258.

[Din+09] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. "Scalable work stealing." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–11. DOI: 10.1145/1654059.1654113.

[HL08] T. Hoefler and A. Lumsdaine. "Message progression in parallel computing - to thread or not to thread?" In: *2008 IEEE International Conference on Cluster Computing*. 2008, pp. 213–222. DOI: 10.1109/CLUSTR.2008.4663774.

[Jan] P. S. Jannis Klinkenberg Minh Thanh Chung. *Chameleon Apps GitHub*. URL: https://github.com/chameleon-hpc/chameleon-apps (visited on 07/05/2021).

[Kli+20] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller. "Chameleon: reactive load balancing for hybrid MPI+ OpenMP task-parallel applications." In: *Journal of Parallel and Distributed Computing* 138 (2020), pp. 55–64.

[NC99] J. Nieplocha and B. Carpenter. "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems." In: *International Parallel Processing Symposium*. Springer. 1999, pp. 533–546.

[Sha+12] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. "A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI." In: *SIGMETRICS Perform. Eval. Rev.* 40.2 (Oct. 2012), pp. 92–98. ISSN: 0163-5999. DOI: 10.1145/2381056.2381077.

[Zhe+14] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. "UPC++: a PGAS extension for C++." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1105–1114.