SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Malleability for the HPCG Benchmark Using LAIK

Ibrahim Erdurucan

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Malleability for the HPCG Benchmark Using LAIK

Unterstützung dynamischer Ressourcenanpassung für den HPCG-Benchmark mit Hilfe von LAIK

Author:	Ibrahim Erdurucan
Supervisor:	PD Dr. rer. nat. Josef Weidendorfer
Advisor:	M. Sc. Amir Raoofy
Submission Date:	15th of December 2023

Acknowledgments

I extend my sincerest gratitude to my supervisor PD Dr. rer. nat. Josef Weidendorfer and my advisor M. Sc. Amir Raoofy for their valuable input and expertise. Without their input and expertise, this work would have been quite difficult. I also express my appreciation to Yassin El Gezawi for delivering constructive feedback. I would like to thank Eduard Ugulava for printing this work. We have been waiting for this moment for three years. Lastly, I would like to thank my family for their unwavering emotional and practical support throughout.

Abstract

Adjusting global data dynamically to available resources remains a frequent issue in HPC applications. This task is particularly challenging with legacy MPI code. Developers require an in-depth understanding of the code base and must write application-specific code to enable dynamic adaptation. To tackle this challenge, the authors of LAIK, a library for fault-tolerant distribution of global data for parallel applications, have worked on developing an efficient solution. LAIK streamlines the process by offering a solitary function to adjust the number of participating processes. In this work, we ported the High Performance Conjugate Gradient Benchmark to LAIK to enable dynamic global data adaptation. Furthermore, we used the abstract concepts of LAIK for interprocess communication. This was accomplished incrementally. There were many factors to consider, but ultimately achievable. Our final version yielded comparable results to the original application. To complete this work, we evaluated computing performance, memory consumption and the effectiveness of dynamically adapting resources.

Contents

1 Motivation

Due to factors such as heat production and the physical limitation on processor speed, computer systems are now designed with multiple processors. As a result, the idea of parallel systems has emerged, where multiple processors can collaborate towards a common aim with a given overhead in terms of communication. Furthermore, the workload can be dispersed by utilising various computer systems such as distributed systems, in addition to multiple processors. At best, this allows developers to enhance application performance. However, it requires developers to consider the principles of parallel programming. This is where the Message Passing Interface (MPI) becomes essential. Carleton University explains: "The Message Passing Interface (MPI) is an Application Program Interface that defines a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes. Using MPI allows programs to scale beyond the processors and shared memory of a single compute server, to the distributed memory and processors of multiple compute servers combined together" [17]. The interface lessens the burden on developers by eradicating the requirement for low-level communication code implementation. High Performance Computing (HPC) applications often employ MPI, which is also used by the High Performance Conjugate Gradient Benchmark (HPCG Benchmark) software. Essentially, the HPCG Benchmark introduces a novel ranking metric for HPC systems. One recurring challenge encountered in HPC applications pertains to the fluctuating adaptation of global data to available resources, such as requiring an additional node for computation. The HPCG Benchmark presents the same issue since it can only accommodate a predetermined quantity of compute nodes. The matter to address is whether it is viable to sustain a feature that permits the count of nodes to fluctuate based on resource accessibility, without requiring explicit implementation. How about simplifying communication by using abstract concepts to avoid explicit communication code, such as setting up send or receive buffers? Bearing these questions in mind, Josef Weidendorfer and his team have introduced *A Library for Fault Tolerant Distribution of Global Data for Parallel Applications (LAIK)*, which is, in summary, an abstraction of MPI with added features, including resource adaptation and code abstraction for inter-process communication. The team behind LAIK, comprising Josef Weidendorfer, Dai Yang, and Carsten Trinitis, outline their approach: "HPC applications are typically not written to handle dynamic changes in the execution environment, such as the removal or addition of new nodes or node components. However, for greater flexibility in scheduling and fault-tolerance strategies, an application-integrated response would be worthwhile. This is difficult to achieve with legacy MPI code" [Jos17]. To achieve malleability in the HPCG Benchmark with ease, the code should be migrated to the LAIK library. *Malleability for the HPCG Benchmark Using LAIK* presents the process of migration and the corresponding challenges. Firstly, an overview of LAIK and the HPCG Benchmark will be presented. Following this, we will give a detailed explanation of how the HPCG Benchmark was adapted for LAIK. Subsequently, we will assess the performance outcomes and draw comparisons with the original HPCG Benchmark. Lastly, the dissertation will culminate in a summary and potential avenues of future exploration.

1.1 Related work

The LAIK team has managed to port the *LULESH* program to their platform with great success. Nevertheless, the duty at hand comprises more than only the HPCG Benchmark's migration to LAIK. Additionally, it demands exploring the LAIK library and integrating a new custom layout as described in Chapter 3.3. By undertaking this, we avoid having to make major code changes to the application being ported.

2 Background

Before examining the port's particulars, we will provide a summary of LAIK and the HPCG Benchmark.

2.1 LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications

LAIK is a C-based library that serves as a wrapper for MPI, as well as supporting TCP2 and TCP backends. This allows for the simultaneous use of multiple communication libraries in an application. For example, it is plausible to partially use LAIK in the HPCG benchmark while still explicitly using MPI. Moreover, LAIK enables the transfer of current applications unrestrictedly. LAIK facilitates the distinctive ability to resize the *world* dynamically, as expounded in the motivation section (refer to Chapter 1). The term *world* in this thesis and LAIK refers to a cluster of computing nodes that is known as a *communicator* within the context of MPI. Each process is linked to a specific *world*. A useful analogy is to envision a population residing in the same "world" (refer to Figure 2.1). Processes are capable of exchanging data with other processes within the same *world*. This communication can occur in either a point-to-point or collective manner. In MPI, the default communicator is referred to as *MPI_COMM_WORLD*. Separation of this group of processes is possible at a later stage. Upon launching an application, each process initially adheres to the default communicator. LAIK presents a similar concept called *Laik_Group*. There are additional LAIK-specific features beyond the segmentation of process groups, specifically the removal of existing processes or incorporation of new processes into the system. The latter is particularly helpful in instances where the application requires additional resources. LAIK demonstrates that this can be achieved with just a single function call. As a result, LAIK can integrate computing nodes from the *world* without requiring the application to be re-executed with the updated set of computing nodes. Further information regarding this feature will be discussed in Chapter 3.2. LAIK integrates the communication code that programmers usually need to write explicitly in their applications. When communication functions are invoked, LAIK performs a consistent sequence of MPI calls. It is worth noting that communication is generated from the developer's perspective, achieved by partitioning

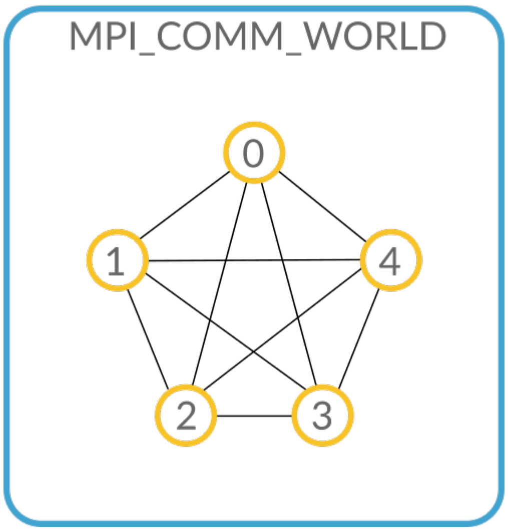

Figure 2.1: An illustration of how processes can co-exist and communicate with each other.

abstract index spaces, which requires the developer to think about the data in this fashion. This means that the programmer possesses an abstract and straightforward concept for designating communication, which we will examine in greater depth.

2.1.1 The Concept of Partitioning: A Comprehensive Analysis

As previously stated, interprocess communication will be specified at an abstract level by developers. The LAIK technology incorporates the concept of partitioning global index spaces, which enables HPC programmers to define communication requirements by transitioning between access phases to data structures using partitioning. Currently, LAIK exclusively offers support for 1/2/3D arrays acting as a data structure or container. A diagram of a global index space can be viewed in Figure 2.2. The index space has a

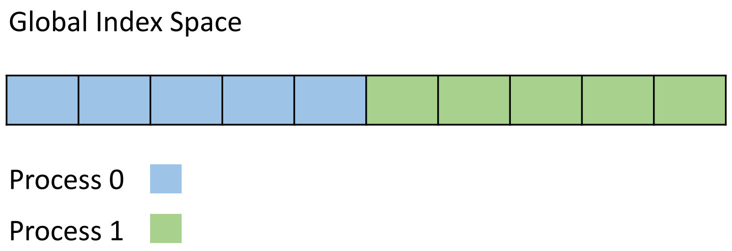

Figure 2.2: A global index space with an exemplary partitioning.

total size of ten. We then determine a partitioning for assigning indexes to processes in the abstract index space. In this instance, we assigned the initial five indexes to *process 0* and the last five indexes to *process 1*. Other extant processes may have no assigned indexes. Therefore, this partitioning is the current access phase to the data structure. Now, it is assumed that *process 1* should access the initial five elements, and *process 0* should access the final five elements (refer to Figure 2.3). The second phase

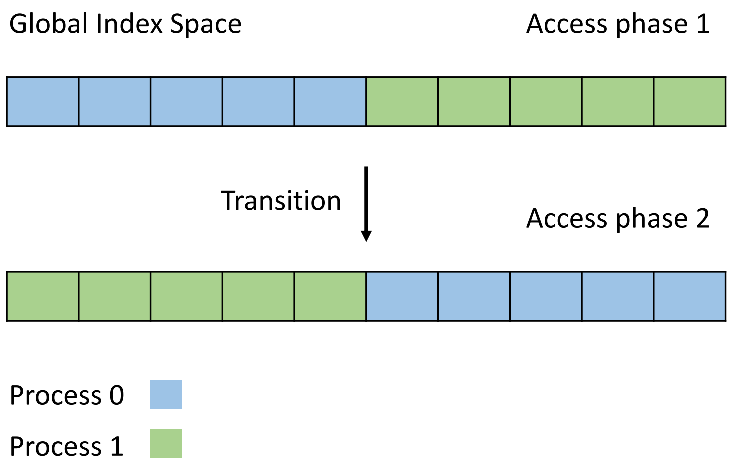

Figure 2.3: Transitioning from access phase 1 to access phase 2

of the data access would be a different partitioning of the global index space from the first phase. Transitioning to this phase triggers inter-process communication, which is automatically detected by LAIK. In our particular case, LAIK recognises that the first five elements pertain to process 0, but switching to the new partitioning assigns the indices to process 1. For the final five elements, process 1 will transmit their values to process 0 in relation to the new access phase (partitioning). LAIK retrieves the information from the partitioning and readies all elements for communication. At the same time, it's possible to retain items locally, transmit an arbitrary group of values to *process i* and other values to *process j*, or receive values from one or more processes through a single function call. As previously mentioned, LAIK furnishes the application with several backends to utilize. In the case that MPI is enabled, LAIK automatically executes the sequence of MPI calls. This implies that the programmer does not have to worry about send or receive buffers or low-level code. LAIK is not restricted to exclusive access to indices in space. Multiple processes can access the same global indices, leading to conflict resolution by LAIK through a *Reduction Operation*. In summary, the programmer simply needs to specify which processes should have access to certain global indexes to instruct LAIK how to communicate (refer to Figure 2.4).

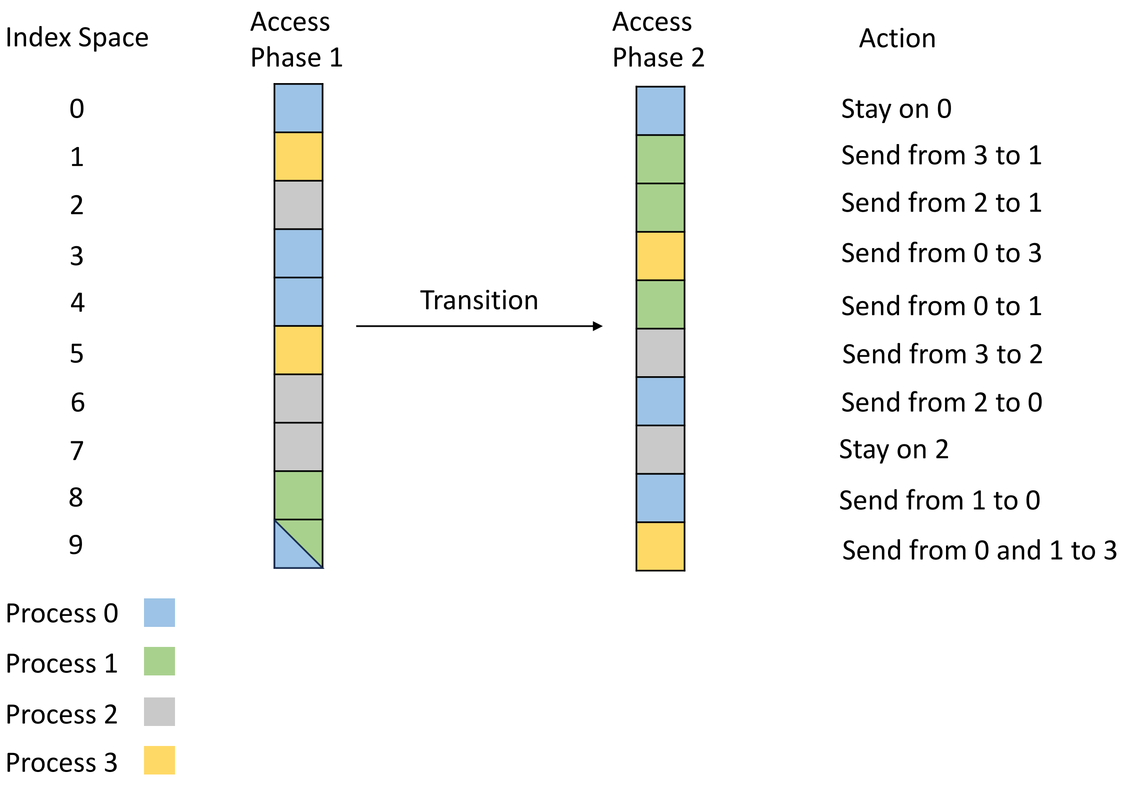

Figure 2.4: After switching to a new access phase, LAIK automatically calculates all actions.

Utilizing LAIK for Collective Communication: An Exploration through Sample Code

Let us now explore how to apply the concepts mentioned previously. As a reminder, programmers should approach interprocess communication from an abstract standpoint. In this scenario, our goal is to perform an All-to-All reduction where every process sends its values to all others and undergoes a reduction operation using the *LAIK_RO_Sum* reduction operation. Thus, each process will gather the received values, including its own, and store the value after applying the reduction operation, as demonstrated in Figure 2.6. As we mentioned, LAIK exclusively offers support for 1/2/3D arrays acting as a data structure or container. In C, it is introduced as *Laik_Data*. Therefore, we assume that the data consists of values belonging to any type. LAIK facilitates several types of values that are stored in the data container. It is also feasible for the user to craft their custom types. To maintain simplicity, we shall employ *laik_Int32*. A sample code is provided to demonstrate the implementation of LAIK for the All-to-All reduction process.

```c
#include <laik.h>

int main(int argc, char* argv[])
{
    // Initialization functions
    Laik_Instance * instance = laik_init(&argc, &argv);
    Laik_Group * world = laik_world(instance);

    // Define abstract index space of global size 1
    Laik_Space * space = laik_new_space_1d(instance, 1);
    // Create partitioning to specify, which processes should have access to global index 0
    // laik_All is the partitioner algorithm, which creates a partitioning, where every process has access
      to every index
    Laik_Partitioning * allParitioning = laik_new_partitioning(laik_All, world, space, NULL);
    // Create data structure/container containing elements of type int and associated with <space>
    Laik_Data * sum = laik_new_data(space, laik_Int32);
    // Switch to the access phase/partitioning <allParitioning>
    laik_switchto_partitioning(sum, allParitioning, LAIK_DF_None, LAIK_RO_None);

    // assign arbitrary value [...]

    // Switch again to <allParitioning> to do an All-to-All reduction
    laik_switchto_partitioning(sum, allParitioning, LAIK_DF_Preserve, LAIK_RO_Sum);

    // ...
}
```

Figure 2.5: An example code snippet demonstrating how to perform an Allreduce operation using LAIK.

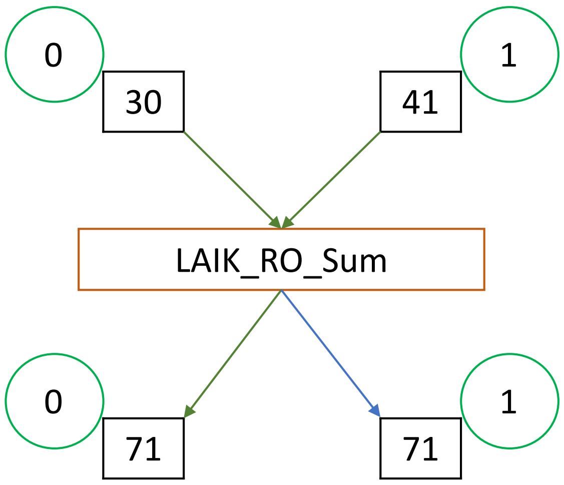

Figure 2.6: All-to-All reduction visualisation.

2.1.2 The Resizing Capability of LAIK

As outlined in the motivation, a common issue in HPC applications is the need to dynamically adjust resources according to availability. This could have various reasons: If the HPC system has no further need for certain processes or requires additional nodes to carry out the computation. But it could also be that a process will fail. For this scenario, an application could determine the possibility of a process failing and resize the world with LAIK to prevent the application from crashing and having to run the application again. This may prevent data loss. In the context of LAIK, the resize function refers to the dynamic alteration of the world size. Resizing in LAIK involves a single function call from a programmer's perspective, making it a straightforward process:

```
Laik_Group* laik_allow_world_resize(Laik_Instance* instance, int phase);
```

The *Laik_Instance* specifies the instance of the world in which processes are operating, while the *Laik_Group* represents that world itself. LAIK also defines a parameter named phase which we will explain later. Let us first analyse how LAIK offers this functionality. This begins with the use of a dedicated launcher, named *tcp2run*. Let us assume that we intend to initiate computation with 50 processes and subsequently incorporate 21 more processes:

```
tcp2run −n 50 −s 21 <path/to/executable>
```

The launcher exports an environment variable known as *LAIK_SIZE* and sets its value to the number specified in the -n option. Please note that this currently only works with a TCP2 backend. Following this, the launcher commences 71 processes. Subsequently, a *Laik_Instance* must be initialized by a process to utilise LAIK, which can be achieved via the function *laik_init(&argc, &argv)*. Currently, 71 processes invoke the initialization function. However, LAIK is aware via the environment variable *LAIK_SIZE* that only 50 processes should proceed and join the world. Thus, LAIK permits solely 50 processes to exit the initialization function. Upon running *laik_size(world)*, the output would be 50 rather than 71. Consequently, only 50 processes participate in exchanging data within this group/world. If the application requires additional processes to accelerate computation, the already running processes should execute the function *laik_allow_world_resize()*. Subsequently, the function *laik_init()* will release the remaining 21 processes. If a process invokes *laik_size()* at this point, the result will be 71. Consequently, all 71 processes can now share the resources available. Let us return to the parameter phase. During the computation, processes need to set the phase value to a number greater than zero. This permits new joining processes to obtain the phase, and developers can distinguish between the current process's type - new joining or old - using a simple if-statement. The redistribution of information represents the final essential step, which must be carried out manually by the application developer, as it is dependent on the particular application. We will see how we did that concerning the HPCG Benchmark.

2.1.3 Understanding the Concept of Layouts

Another vital aspect of LAIK is how it manages data storage in memory. The developers have introduced the concept of layouts for this purpose. A layout defines how process-local partitions of a data container are stored in memory, which is crucial since LAIK offers automatic data management. Understanding this concept is necessary for mapping global indexes to their corresponding memory offsets. It is crucial because if values are to be exchanged or copied, LAIK needs to know the correct offset in the buffer. The pre-existing layout offered by LAIK is named the *Lex Layout* and will be introduced in greater detail at a later point. LAIK's generic layout version is implemented through a provided interface that layouts are required to adhere to, granting developers the ability to implement their custom layouts. Later on, we will explore how to create our custom layout. Generally, a layout facilitates operations such as copying values between two buffers through the *copy* function, indicating to LAIK whether it can reuse a memory/buffer for optimisation purposes with the *reuse* function, or direct LAIK where to copy incoming or outgoing values with the *(un-)pack* function. In summary, the implementation of a layout is crucial to automate the process of (de)allocating buffers, communication, etc. Developers can design their layout to meet the specific requirements of each application, such as access patterns or optimization purposes.

2.1.4 Benefits of using LAIK

There are many benefits os using LAIK:

- Different communication libraries can be used simultaneously.
- Dynamic adaptation of resources (see in Chapter 3.2)
- Increase of maintainability
- Easy-to-understand concept of specifying communication between processes
- Customised application-specific layouts are supported
- Open source library
- Support by the developers

Now that background information on LAIK has been presented, we shall continue with the HCPG Benchmark.

2.2 High Performance Conjugate Gradient Benchmark (HPCG)

The HPCG Benchmark, as mentioned in the motivation, is software for ranking HPC systems. The authors Jack Dongarra, Michael Heroux and Piotr Luszczek explain that the HPCG Benchmark "performs a fixed number of multigrid preconditioned (using a symmetric Gauss-Seidel smoother) conjugate gradient (PCG) iterations using double precision (64-bit) floating point values" [DL19]. Jonathan Richard Shewchuk outlines, that "CG is the most popular iterative method for solving large systems of linear equations. CG is effective for systems of the form $Ax = b$ where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite (or positive-indefinite) matrix" [She94].
The HPCG Benchmark assesses the effectiveness of HPC systems by measuring the performance of preconditioned CG iterations. Jack Dongarra, Michael Heroux and Piotr Luszczek further state, that the "HPCG rating is a weighted GFLOP/s (billion floating operations per second) value that is composed of the operations performed in the PCG iteration phase over the time taken. [...]". After a successful execution, the HPCG Benchmark produces a report containing results and performance measures. The report can subsequently be uploaded to the HPCG Benchmark website where a ranking can be obtained. We will provide more detailed information in Chapter 4.

2.2.1 Work distribution

If the computation involves multiple nodes, the HPCG Benchmark will break down the intricate task into smaller pieces. The workload is then shared among active processes by randomly distributing the rows of the sparse matrix to the processes. As the Conjugate Gradient (CG) method is implemented iteratively, updates to the result vector happen after each iteration. Hence, processes need to retrieve the updated value from the relevant process. This is because if a process owns a global row <i>, it will update the global vector at index <i>. Assuming a sparse matrix-vector multiplication is intended, another process may require the global vector value at index <i>. However, each process can only access its local portion of the global vector. As a result, the HPCG Benchmark identifies the specific values that each process requires from other processes. These values are then transmitted using MPI. This results in a unique access pattern for the HPCG Benchmark. If external values from other processes are needed, the HPCG Benchmark will store them at the end of the local values in the local vector. We will observe that our bespoke design was intended specifically for this function. Another reason for a unique access pattern is the use of a sparse matrix where only non-zero values, their corresponding indices and the number of non-zero values per row are stored (refer to Figure 2.7). This is the technique utilised by the HPCG Benchmark to perform mathematical operations, including sparse matrix-vector multiplication.

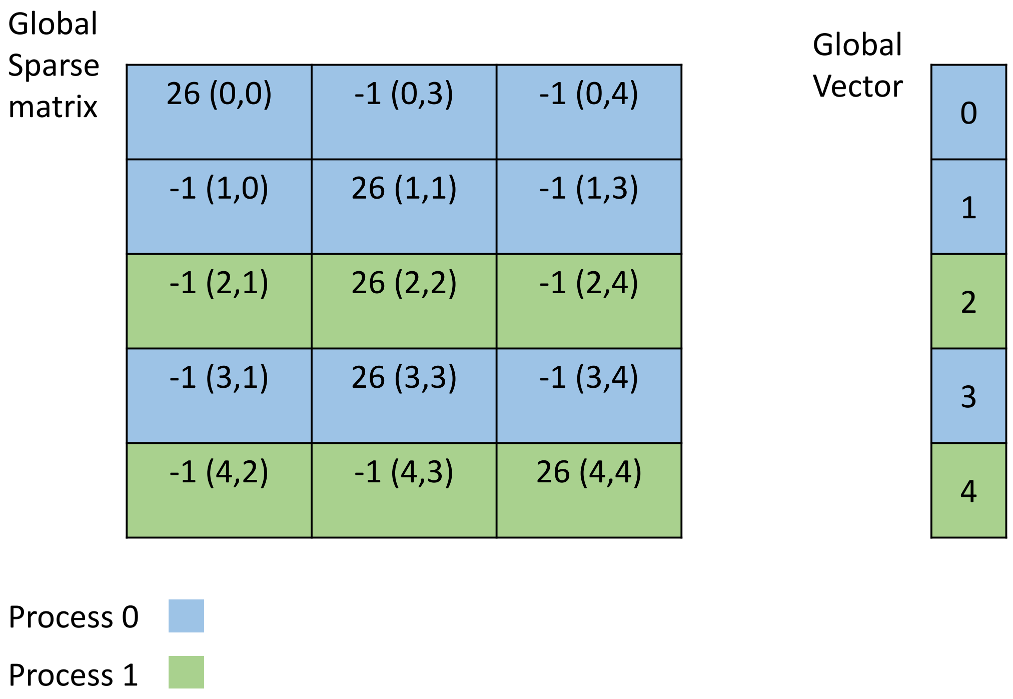

Figure 2.7: An example of a pseudo-random distribution of rows between two processes.

3 Porting HPCG to LAIK

Our objective is to utilize LAIK to allow for resource malleability and employ LAIK as a means of communication during mathematical operations within the HPCG Benchmark. Additionally, our objective is to create a novel custom layout that caters to the access pattern of HPCG in mathematical operations. Given our understanding of the library, application, and problem statement, we aim to detail the complete process of utilising the LAIK features.

3.1 Integration of LAIK into the HPCG Benchmark

We will begin by outlining the integration process of LAIK into the HPCG Benchmark.

3.1.1 Replacing MPI with LAIK during Broadcast and Reduction Operations

Firstly, LAIK was required for operations such as Broadcast or Allreduce. So we replaced all the MPI functions that were used to exchange certain values. To avoid redundancy, we developed an auxiliary function for the *laik_broadcast* and *laik_allreduce* operations. The function signatures are easily understandable.

```
// These functions use LAIK for communication
void laik_broadcast(const void * sendBuf, void * recvBuf, uint64_t n, Laik_Type * data_type);
void laik_allreduce(const void * sendBuf, void * recvBuf, uint64_t n, Laik_Type * data_type,
    Laik_ReductionOperation ro_type);
```

As it is evident, the signature omits the *Laik_Data* structure. However, for communication purposes, it remains feasible to implement our solution using LAIK. Our approach involves preparing LAIK-specific data by creating LAIK-specific objects, extracting values from the *sendBuf* to the *Laik_Data* object, performing communication, and transferring the values back to the *recvBuf*. The problem we faced pertained to a rigid limit stipulated by LAIK on the number of *Laik_Data* objects that can be created. Consequently, we resolved the issue by reutilising existing data containers and employing a map-based approach.

```
// LAIK has a hard lmit of Laik_Data objects. Reuse Laik_Data objects with the same size and
    Laik_Type
std::map<std::pair<int, Laik_Type *>, Laik_Data *> data_objects;
```

```

Laik_Data *data;
Laik_Space *space;
if (data_objects.find({n, data_type}) != data_objects.end())
    data = data_objects[{n, data_type}]; // Get data container
else
{
    space = laik_new_space_1d(hpcg_instance, n);    // Define abstract index space of size n
    data = laik_new_data(space, data_type);         // Allocate new data container with type <data_type>
    data_objects.insert({{n, data_type} , data});   // To reuse that containe later
}
```

In this phase, MPI calls have solely been substituted for operations such as Broadcast/Allreduce. In the following stage, the preliminary plan to enable communication through LAIK was implemented whilst executing operations such as sparse matrix-Vector-Multiplication.

3.1.2 Calculation of Partitionings for Vectors

If you refer back to the code example presented in Figure 2.5, we have used the partitioner algorithm named *laik_All* for simplicity. With that, we achieved a partitioning in which each process has access to every global index within the abstract space. Thus, to determine the data structure's required partitioning, the programmer must furnish LAIK with a partitioner algorithm, which each process will then execute offline, leading to LAIK's partitioning of the abstract index space and the outputting of the resulting partitioning. We will demonstrate our partitioner algorithm at a later stage to clarify its purpose.
During the second phase of the porting process, a major mistake was made by developing the partitioner algorithm with a focus on local partitioning. To ensure optimal outcomes, developers must understand the global perspective and adjust the partitioning process accordingly. Therefore, we had to revise our partitioner algorithm. Initially, we conducted a thorough search for all the necessary information to partition the data container. As we were adapting an existing application, the necessary partitioning information had already been calculated within the code. We aimed to employ it within the context of LAIK. In contrast, the HPCG Benchmark developers created (network) buffers or implemented communication directly. To transmit the information to LAIK, we introduced a struct called *partition_data*. As a consequence, the following code was achieved:

```
void partitioner_alg_for_x_vector(Laik_RangeReceiver *r, Laik_PartitionerParams *p)
{
    // Get information needed for our partitioner algorithm
    partition_d * data = (partition_d *) laik_partitioner_data(p->partitioner);
5      // Current abstract index space we are working on: for the vectors used in HPCG
6      Laik_Space * x_space = p->space;
7      // Some error handling
8      assert(data->size == laik_space_size(x_space));
9
10     Laik_Range range;
11     for (long long i = 0; i < data->size; i++)
12     {
13         // assign every process its global index in the x vector
14         int proc = ComputeRankOfMatrixRow(*data->geom, i); // HPCG implements this function to
           calculate the rank/id owning the global row <i> of the sparse matrix
15
16         laik_range_init_1d(&range, x_space, i, i + 1);
17         laik_append_range(r, proc, &range, 1, 0);
18     }
19     if(data->halo)
20     {
21         // Code snippet for Partitioning 2 (accesing external values as well)
22         [...]
23     }
24 }
```

LAIK presents a partitioner algorithm signature that stores the data we provide alongside additional information, thereby constituting an indispensable phase in partitioning a LAIK data container. For our situation, we need two partitionings to allow switching between them and exchanging values. The initial partitioning, referred to as the *local partitioning* henceforth, exclusively permits the process-owned index (row). In contrast, the second partitioning, termed the *external partitioning* throughout this work, grants access to external values as needed. Refer to Figure 3.1. Our partitioning algorithm addresses this scenario in the if-statement on line 19. The HPCG Benchmark demonstrates deterministic behaviour when randomly distributing rows. To illustrate, consider a size 7 vector x and the arbitrary allocation of rows to two processes. We assume that process 1 requires access to external values at global indices 0 and 3, while process 0 only necessitates the value at global index 5. As previously mentioned, when the local partitioning is in operation and the application transitions to the external partitioning, processes will exchange values. Each process assigns new values solely to indexes that align with its rows in the global problem matrix. Hence, resolving overlapping indices is unnecessary upon returning to the local partitioning as LAIK is directed not to communicate values during the transition. Despite implementing the partitioner algorithms, they remained untested. The upcoming step will analyse the persisted problem.

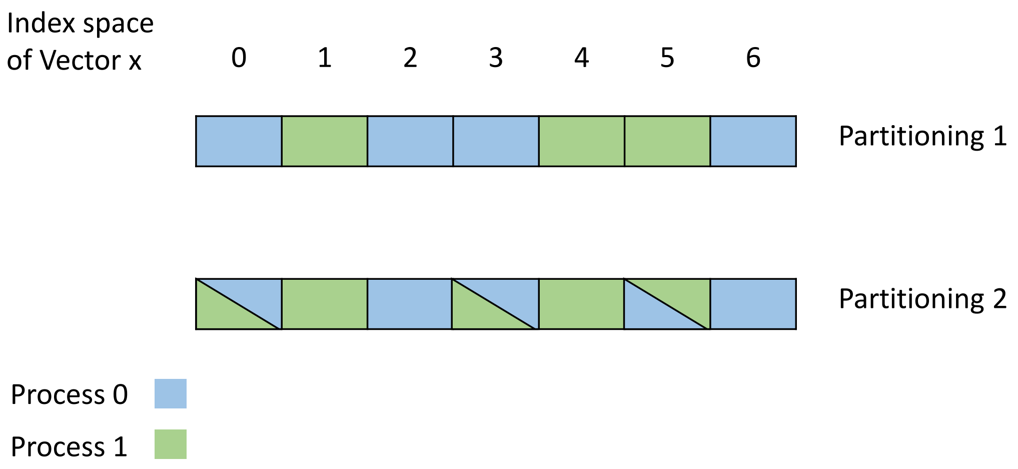

Figure 3.1: The HPCG Benchmark employs two partitionings for communication via LAIK. In this example, partitioning 1 refers to local partitioning, while partitioning 2 refers to external partitioning.

3.1.3 Calculation of the mapping resulting from the Lex Layout

The next step was to implement a mapping. The usage of the *Lex Layout* is the reason for this, as LAIK currently only supports this layout. As detailed earlier in Chapter 2, a layout dictates the storage of local partitions of a data container in memory per process. LAIK considers this a crucial aspect since it offers the developer of applications abstract structs (Laik_Data) whilst automatically allocating the stored data.
The clearest way to elucidate how the *Lex Layout* allocates a buffer through partitioning information is by providing a visual representation. Let us assume that the current active partitioning on the data structure is the local partitioning, as presented in Figure 3.1. Upon further examination of Figure 3.2, the allocation buffer, referred to as the one allocated by LAIK, of process 0 differs from that of process 1. This distinction is due to *Lex Layout* determining the lower and upper limits of process-local partitioning. In this specific instance, the lower limit of process 0 is 0, and the upper limit is 6. For process 1, the lower bound is established at 1 and the upper bound at 5. It should be noted that the allocation buffer of process 0 contains memory for indexes 1, 4, and 5, even though they are not assigned to process 0. Similarly, the allocation buffer of process 1 contains memory for indexes 2 and 3. This configuration implies that accessing, assigning, or reading values from the data structure may lead to inaccurate index access. Assuming the partitioning depicted in Figure 3.1 and the allocation buffer of process 0, we shall proceed to iterate through the data structure and access its elements in the manner outlined in the code snippet below.

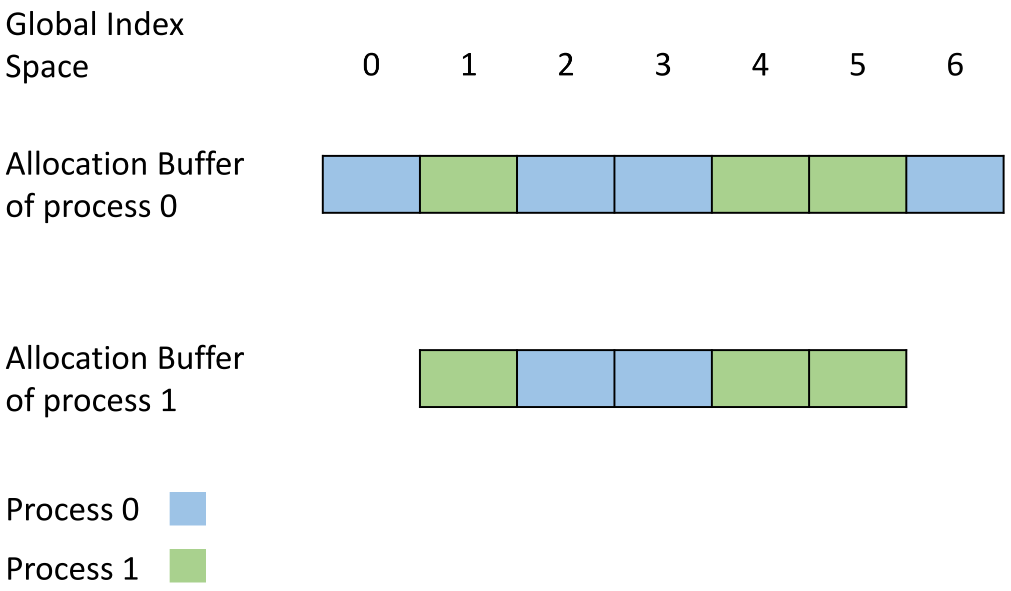
Global Index Space
0
1
2
3
4
5
6
Allocation Buffer of process 0
Allocation Buffer of process 1
Process 0
Process 1

Figure 3.2: Buffers automatically allocated by LAIK using the *Lex Layout*.

```
int localSize = 4; // we have four global indexes assigned to process 0
for(int i = 0; i < localSize; i++)
  base_pointer[i] = 71;
```

If we examine the allocation buffer of process 0 (refer to Figure 3.3), it becomes clear that the initial four elements in the data structure have been assigned a value of 71. This could cause unexpected behaviour in the operation results. The risk of receiving

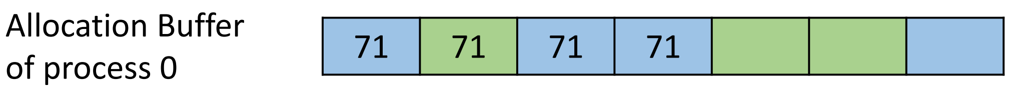

Figure 3.3: Accessing buffer without mapping.

an uninitialized value when process 0 transfers the value indexed at 6 after switching to another partitioning exists. To avoid this, a mapping from local indices to global indices was necessary. We found that the HPCG Benchmark had already completed this task. Using this new understanding, we were able to advance and modify our code, as illustrated next.

```
int localSize = 4; // we have four global indexes assigned to process 0
for(int i = 0; i < localSize; i++)
  base_pointer[map_local_to_global_index(i)] = 71;
```

And now, values have been assigned to the correct offsets in the allocation buffer (see Figure 3.4). However, this was a challenge that we still had to tackle. After analyzing

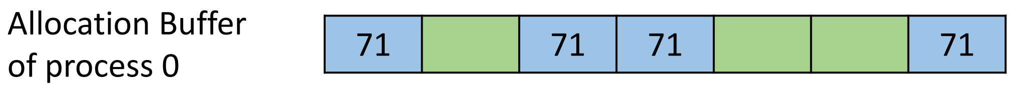

Figure 3.4: Accessing buffer with mapping.

process 1's allocation buffer, we discovered that global indexes commence at 1. This causes an issue as local index 0 being mapped to global index 1 would lead to accessing the second element of the allocation buffer instead of the intended first element. To provide a more precise demonstration of this problem, please consider the following code snippet:

```
int localSize = 3; // we have three global indexes assigned to process 1
for(int i = 0; i < localSize; i++)
  base_pointer[map_local_to_global_index(i)] = 71;
```

Process 1 maps local indices 1 and 2 to global indices 4 and 5, respectively. Subsequently, the corresponding indexes are assigned according to Figure 3.5. The reasoning is

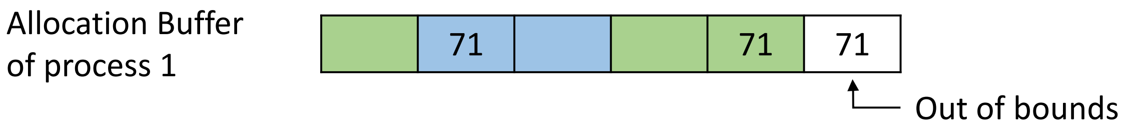

Figure 3.5: Accessing the buffer without an additional mapping step.

simple: indexing buffers start at 0 (e.g. base[0]). However, accessing the first element (global index 1) at base[1] while utilizing the mapping of local to global indices presents a challenge. The value assigned to global index 1 is at offset 0. Consequently, an additional stage is necessary to compute an accurate mapping based on the *Lex Layout*: establish a mapping from local to global index, then from global to allocation index. In our context, the allocation index pertains to the offset or index within the allocation buffer that LAIK generated. A simple method to determine this index involves deducting the smallest global index owned by process i from the global index. In our example, process 0 would subtract 0, while process 1 would subtract 1. We computed the mapping and used it to access the data structures. See the following code snippet for an example:

```
int localSize = 3; // we have three global indexes assigned to process 1
for(int i = 0; i < localSize; i++)
  base_pointer[map_local_to_allocation_index(i)] = 71; // map local to allocation index
```

The final results can be seen in Figure 3.6. In summary, we carried out this step in a timely manner. However, it would have been more desirable if there were no gaps, as we observed. Chapter 3.3 will detail how we removed these gaps by implementing a customised layout.

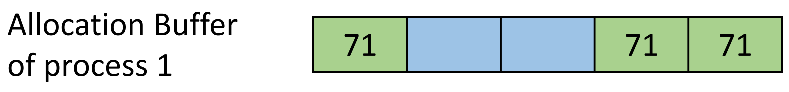

Figure 3.6: Accessing the buffer with an additional mapping step.

3.1.4 Using LAIK Vectors throughout the Benchmark

The fourth stage of our HPCG Benchmark involved the use of LAIK vectors. But it was essential to test everything we had done before proceeding to the next step. To achieve this, we initiated a local *Laik_Data* object in the *ComputeSPMV()* function. Our strategy was to transfer the entries from the original vector into the data container. We then exchanged values via LAIK, performed the sparse matrix-vector multiplication, and restored the resulting values to the original vector. During execution, the HPCG Benchmark generates a file and adds the results of the operations to it. We were then able to compare our results with those of the original application and found no discrepancies between the two files. This indicated the success of the partitioner and the exchange via LAIK.
After conducting tests on our partitioner algorithm, we needed to establish the optimal way to use LAIK vectors across the application. Initially, we tried to include parameters in existing functions where exchange took place. This occurred because we had exclusively transferred the vectors to LAIK, which HPCG employed to exchange values. We intended to retain the original vector parameter for compatibility with earlier versions. However, this method led to several obstacles, resulting in intricate code. Multiple if-statements were utilized to determine the usage of LAIK vectors. Moreover, mapping presented difficulties as it was imperative to use it with LAIK vectors, resulting in more if-statements during operation to distinguish between the LAIK vector and the original HPCG vector. Looking at this code snippet without delving into detail, it is immediately apparent that it does not adhere to good coding practices.

```
ComputeWAXPBY(nrow, 1.0, x, alpha, p, x, A.isWaxpbyOptimized, NULL, p_laik_vector, NULL);
ComputeWAXPBY(nrow, 1.0, r, −alpha, Ap, r, A.isWaxpbyOptimized, NULL, NULL, NULL);
ComputeDotProduct(nrow, r, r, normr, t4, A.isDotProductOptimized, NULL, NULL);
```

Our second attempt consisted of creating prototypes of the operations/functions, whereby we replaced all initial vectors with LAIK vectors. For simplicity, we decided to transfer vectors that do not exchange values to LAIK. This approach allowed for cleaner code whilst maintaining backwards compatibility. This step proved advantageous, as later explained in Chapter 3.2. Additionally, the use of directives simplified handling cases where the application runs with LAIK:

```
#ifndef HPCG_NO_LAIK
  ierr = ComputeSPMV_laik_ref(A, x_overlap, b_computed); // b_computed = A*x_overlap
#else
  ierr = ComputeSPMV_ref(A, x_overlap, b_computed); // b_computed = A*x_overlap
#endif
```

Small Optimization

The fourth stage also involved optimizing the code to resolve a bus error that occurred after the initial call of *ComputeSPMV()*. Our method involved initializing and storing all the required data for the partitioning algorithm, without creating any partitionings. For every LAIK vector, we subsequently produced uniform partitionings during the initialisation process. This technique necessitated allocating a new buffer and replicating data for the partitioner algorithm each time, resulting in poor performance. This resulted in the duplication of bytes and delays in repeatedly creating and storing data for the partitioning algorithm. This approach is impractical for larger problem sizes and even for smaller ones, it may result in bus errors.
Thus, the solution proved uncomplicated. We promptly produced and saved the partitionings once when arranging information for the partitioning algorithm. For each LAIK vector, we initiated the data container and switched to the relevant partitioning. This eliminated the requirement for persistent generation of partitioning or allotment of new buffers, therefore preventing redundant replicas.

3.1.5 HPCG LAIK v1.0

When we tried to compile the application after the previous step, there was a small linking error with a big impact. It took us a couple of hours to fix it:

```
// Function declaration was
int ComputeMG_laik_ref(const SparseMatrix &A, Laik_Vector * r, Laik_Vector * x);
// instead of
int ComputeMG_laik_ref(const SparseMatrix &A, const Laik_Vector * r, Laik_Vector * x);
// => "const" was missing. It resolved the building issues.
```

After resolving the identified problem, we were able to successfully compile our application. Subsequently, we proceeded to the fifth stage: testing to verify whether the output of the result file mirrored that of the original application. Regrettably, the first attempt resulted in segmentation faults. Upon closer examination, we discovered the dilemma outlined in Figure 3.5, wherein we had unintentionally omitted utilising the

mapping from local to allocation indices for each LAIK vector. Despite attempts to correct the discrepancies, inconsistencies remained in the results. In response to the query regarding the source of these inconsistencies, an iterative approach was implemented. The subsequent operation, *ComputeMG_ref()*, was also analysed to resolve the issue. We analysed the output vector of both the original software and the LAIK variant following each operation, including *ComputeSPMV_ref*, which yielded no inconsistencies. After executing this function, we detected discrepancies. We analysed the function and executed its first operation, *ComputeSYMGS_ref*. By comparing the resulting vectors, we identified the root of the problem to be the aforementioned operation. Further evaluation of the code, comparing the initial version with the modified version, did not reveal any instances of copy-paste mistakes. During each iteration, it was found that a variable called *sum* was assigned an entry from a vector. However, the problem highlighted in Figure 3.5 reoccurred due to a mismatch between the local and allocation indices resulting in significant disparities in the output file. Therefore, the last step involved resolving this issue, leading us to the conclusion of the initial HPCG LAIK v1.0 port version.
We now reach an intriguing topic: facilitating the expansion and contraction functionalities. The subsequent Chapter will explore how LAIK can adapt to resizing the world (i.e. adjustments to the number of participating processes) and the encountered challenges during the inclusion of this feature in HPCG LAIK.

3.2 Malleability of ressources in the HPCG Benchmark

After successfully releasing HPCG-LAIK v1.0, our objective was to provide *Malleability for the HPCG Benchmark Using LAIK*. The following paragraphs outline the essential stages required to implement the resize function in the HPCG Benchmark. Initially, we familiarised ourselves with the resizing feature by reviewing sample codes in the LAIK repository. It became evident that the resizing code always had a consistent structure:

1. Calling *laik_allow_world_resize()*
2. Running partitioner algorithms for the new group
3. Redistributing data
4. Releasing old world and old partitionings

Hence, it became apparent that we could integrate it into the HPCG Benchmark with the help of the corresponding boilerplate code that enables world repartitioning. Our main goal was to achieve accurate reduction outcomes.

3.2.1 Enabling shrinking feature

The following phase involved designing partitioner algorithms for the sparse matrix, following the successful demonstration of the LAIK vectors. Our first attempt utilised an abbreviation to reduce size by re-running the setup functions alone. Nevertheless, this approach proved inadequate since only adjusting the *new world size* configuration parameter failed to address the resulting increase in the global problem size. It is necessary to maintain the same overall number of rows in the sparse matrix after repartitioning, given the previous problem size. Consequently, we examined the initialization logic and endeavoured to comprehend how the problem size is determined in the HPCG Benchmark. The HCPG Benchmark calculates the number of processors in the x/y/z-direction (npx/npy/npz) based on the world size:

```
ComputeOptimalShapeXYZ( size, npx, npy, npz );
```

The number of local grid points in the x, y, and z directions (np/ny/nz) can be specified by the user. The default value is 16x16x16. To determine the global grid points, the HPCG Benchmark multiplies the local grid points by the corresponding number of processors:

```
gnx = nx * npx;
gny = ny * npy;
gnz = nz * npz;
```

Thus, the HPCG Benchmark can calculate the local and global problem size:

```
totalNumberOfRows = gnx * gny * gnz;
localNumberOfRows = nx * ny * nz;
```

Therefore, if a new world size is introduced and the setup functions are run again, the function *ComputeOptimalShapeXYZ()* may generate new calculations for npx/npy/npz. A careful examination of the global grid point calculations will reveal that gnx/gny/gnz values will differ from their previous numbers. Thus, the solution to this issue is calculating new quantities of local grid points (nx/ny/nz) in the x/y/z-direction:

```
ComputeOptimalShapeXYZ( new_size, npx, npy, npz );
new_nx = old_gnx / npx;
new_ny = old_gny / npy;
new_nz = old_gnz / npz;
```

One limitation of the program is that the values for npx/npy/npz must be a multiple of old_gnx/old_gny/old_gnz. To address this, we have implemented a solution that involves terminating the program with user-friendly output. After adjusting the local size for each process dynamically, we formulated a technique to attain the identical problem size post-repartitioning. Nevertheless, we encountered assertion errors and segmentation faults due to insufficient attention to detail. In particular, we did not consider the alteration in the size of the local problem subsequent to repartitioning, which necessitated a change in the local length of all LAIK vectors. After resolving this and a few other issues, we proceeded to compare the resulting file. Our initial attempt at repartitioning failed due to a discrepancy in the results. Our test case involved reducing the number of processes in the system from two to one. The non-deterministic behaviour displayed in the result file was intriguing. Despite our failure to identify the problem, we were able to understand the initialisation logic. Therefore, we decided to forego the abbreviation and utilise partitioning algorithms for the sparse matrix.

Partitionings for the Sparse Matrix

Our initial consideration was to determine which data in the sparse matrix required distribution. As a result, we decided to transfer the subsequent elements to LAIK (using *Laik_Data*) following the same approach used for LAIK vectors.

- nonzerosInRow (1d)
- matrixDiagonal (1d)
- f2cOperator (1d)
- matrixValues (2d)
- mtxIndG (2d)

We will later see that transferring the f2cOperator member was superfluous. The partitioning algorithm was mostly identical to that used for the LAIK vectors:

```
void partitioner_1d_members_of_A(Laik_RangeReceiver *r, Laik_PartitionerParams *p)
{
  SparseMatrix * A = (SparseMatrix *)laik_partitioner_data(p->partitioner);
  Laik_Space * A_space = p->space;

  assert(A->totalNumberOfRows == laik_space_size(A_space));

  Laik_Range range;
  for (long long i = 0; i < A->totalNumberOfRows; i++)
  {
    // assign every process its global part
    int proc = ComputeRankOfMatrixRow(*(A->geom), i);

    laik_range_init_1d(&range, A_space, i, i + 1);
    laik_append_range(r, proc, &range, 1, 0);
  }
  return;
}
```

Each process was assigned its global rows, as we did in the algorithm for LAIK vectors. But we had to consider the implementation of the partitioner algorithm for the 2D members of the sparse matrix, such as *matrixValues*. One potential approach was to adopt LAIK's 2D space and data structure. However, this would have required major modifications to the code's access pattern. Consequently, we decided to flatten the 2D members. The HPCG Benchmark sets a fixed limit on the maximum size of non-zero elements per row. Therefore, the 2D arrays in the original code are constructed using arrays within the main array that have a size of 27. To account for this, a slightly adapted partitioning algorithm has been created for the initial 2D components of the sparse matrix:

```
void partitioner_2d_members_of_A(Laik_RangeReceiver *r, Laik_PartitionerParams *p)
{
  SparseMatrix *A = (SparseMatrix *)laik_partitioner_data(p->partitioner);
  Laik_Space * space2d = p->space;

  Laik_Range range;
  for (long long i = 0; i < A->totalNumberOfRows; i++)
  {
     // assign every process its global part
     int proc = ComputeRankOfMatrixRow(*(A->geom), i);
     int from = i * numberOfNonzerosPerRow; // including
     int to =  i * numberOfNonzerosPerRow + numberOfNonzerosPerRow; // excluding

     laik_range_init_1d(&range, space2d, from, to); // we are flattening the 2D array
     laik_append_range(r, proc, &range, 1, 0);
  }
  return;
}
```

Again, each process is assigned its corresponding global rows. However, as the 2D array is flattened, the entire row (which comprises 27 indexes) must be assigned to the process.
After incorporating the partitioning algorithms, we had to modify the code to allow repartitioning, setup functions, and all operations on the sparse matrix data. It was crucial to implement mapping, just like we did for LAIK vectors, because of the *Lex Layout*. Testing all of the new code resulted in discrepancies in the outcome. We will now explore the reasons behind this occurrence and how we resolved it.

Disceprancies in the result

The task at hand was to locate the source of the bug. After much deliberation, we opted to print out each value used during the calculation. We repeated this process with the original application, comparing the output and discovering a discrepancy in the values accessed by *matrixValues*, *matrixDiagonal* and *mtxIndG*. It transpired that there was an initialization error, whereby the off-by-one mistake occurred. However, despite rectifying the off-by-one error, discrepancies remained. We employed the aforementioned method to determine the source of the errors, which were discovered after invoking the function known as *TestCG()*. Upon comparing the output, it became apparent that the values stored in *matrixValues* were not the same. This was attributable to the implementation of the *TestCG()* function. The HPCG Benchmark replaces the *matrixDiagonal* component of the sparse matrix with alternative values. However, the HPCG Benchmark presents an issue as it stores pointers within *matrixDiagonal* which reference the corresponding entry in *matrixValues*. This method avoids storing a value twice yet in LAIK, we did not utilise pointers. When altering data in the *matrixDiagonal*, it is essential to modify the relevant *matrixValues* elements as well. To rectify the issue, correct values were assigned in the *matrixValues*. As a result, there were no further discrepancies observed when using LAIK containers in the sparse matrix during computation. The tests were conducted without modifying the world's dimensions. Then we tested the same with shrinking. Our test case was to start the computation with two processes and shrink it to one. But as we will see, there were still some bugs.

Disceprancies in the result with shrinking

The initial issue encountered after carrying out repartitioning was a segmentation error that occurred when attempting to access the *f2cOperator*. Rectifying the problem was achieved by altering the data type to *laik_Int32*. Upon investigation of the indices used to access the *f2cOperator*, we observed peculiar numbers such as -2302392 and 139231093193. Subsequently, we determined that the segmentation error was caused by incorrect usage of the *laik_UInt64* data type. If you recall the list of members requiring porting to LAIK (see 3.2.1), we indicated that the *f2cOperator* member did not require porting. The rationale was straightforward. The *f2cOperator* constitutes solely of local data needed by each process and redistributing it would be nonsensical, given its lack of global data. Thus, we reverted to the original state after repartitioning. We needed to reinitialize the *f2cOperator* array according to the new settings. Finally, we produced an output file identical to running the original application. Our subsequent task was to develop code for the new joining processes. We emphasized the importance of these processes being aware that they have joined the computation to synchronize with already-running processes.

3.2.2 Enabling expansion feature

Processes involved in computation can be specified from the outset, as well as those added after invoking *laik_allow_world_resize()*. This topic is comprehensively covered in Chapter 2.1.2. This enables new processes to return from the *laik_init()* function and continue execution. However, it should be noted that additional processes would also need to call setup functions, just as the initial processes did. LAIK tackles this problem by introducing a variable known as phase. In the course of computation, processes are allowed to adjust the phase counter to any desired value. To distinguish between previous and current processes, an if-statement verifies if the phase is equal to zero. Therefore, we have adapted the phase to ensure that new processes avoid the need to perform the entire setup repeatedly. Instead, they retrieve information from the original processes during data redistribution. One challenge we faced was coordinating the exchange of values between the original and new processes in the correct order. This was due to the presence of multiple LAIK containers. To prevent extended wait times or blockages during data redistribution, we designated the order of the containers involved. The issue was swiftly resolved. For example, imagine expanding from two to three processes and experiencing repartitioning during the 10th iteration of CG. The phase differentiates between the initial and new processes, as highlighted. Once the initialization and repartitioning were complete, the new processes could proceed to the section of the code where the initial processes were executing. It was required for the new processes to ascertain the current computation iteration, with the aim being to set the phase to the subsequent iteration, thus achieving synchronization amongst all processes. However, there were still discrepancies despite the synchronization. Our debugging method entailed printing the outcome vectors. Upon executing a particular function, we discovered entries with *inf*. Once investigated, we established that a variable named *rtz* was valued at zero. Division of any double by zero led to *inf*. This difficulty exclusively arose with just-added processes since they did not calculate starting functions that set essential variables for the CG. Therefore, it was necessary to implement the latest processes to take values for certain variables from the initial processes. It was discovered that the variables of the recently added processes had not been updated, leading to the discrepancy. As this issue was resolved, we were pleased to announce the release of the second version of the port: HPCG LAIK v1.1.

3.3 Custom Layout: Sparse Vector

As discussed in section 3.1.3, we have removed the task of calculating maps from local to allocation indices. In the following section, we will provide detailed information on our approach. To summarize, a layout outlines the organization of process-local segments of a data container stored in memory through LAIK's automated data management. The abstract index spaces are employed by software developers. If a programmer is seeking to acquire the pointer to the allocation buffer, they may evoke the associated function within LAIK, which will then return the pointer. About the default layout, known as the *Lex Layout*, it is imperative to acknowledge that this configuration designates memory for all global indexes within a given process's low and high bounds, which may result in the allocation of memory for global indexes that do not correspond to that particular process. Therefore, this arrangement requires extra effort, such as mapping calculations and slight code modifications. Once the HPCG Benchmark was ported successfully and the global data was made adaptable, we progressed towards implementing our bespoke layout according to our needs. Our bespoke layout encompasses the *Sparse Vector layout*. As the name implies, we aimed to establish a customised format that would allocate memory similarly to the HPCG Benchmark. This format consists of an array with local indices and memory for external values as mentioned in Chapter 2.2.1. A concise summary: The HPCG Benchmark allocates a buffer to accommodate all local values, and if the vector values are exchanged, the HPCG benchmark allocates additional memory for them and stores them after the local values. Consequently, the objective of LAIK was to create an allocation buffer that comprises global indices unique to the process and has the same access pattern as that of the HPCG Benchmark. Concerning the *Lex Layout*, we intended to eradicate the gaps within the allocation buffer (refer to Figure 3.7). The first step was to understand the LAIK interface objectively. Through a

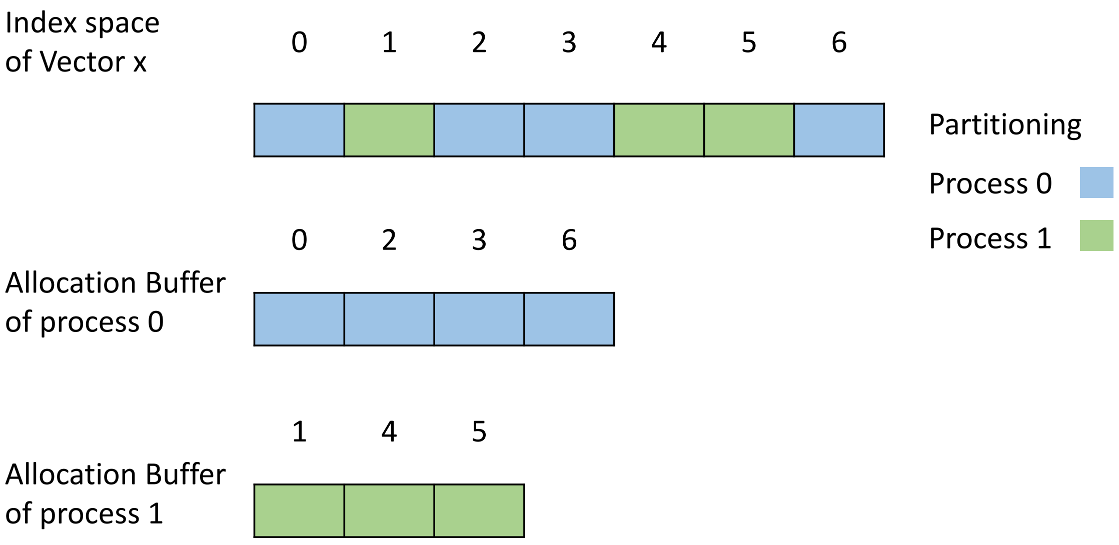

Figure 3.7: Buffers automatically allocated by LAIK using the *Sparse Vector Layout*.

comprehensive review of the documentation and consulting with developers, a clear understanding was achieved. Furthermore, the generic and *Lex Layout* implementations were analysed, indicating a strong structural similarity. In light of this, a new file was produced to address the issue. We created our initial version by adjusting the names after copying and pasting the content of the *Lex Layout* implementation. Specific code was removed during this process. However, there was still a significant amount of work to be accomplished.

3.3.1 Implementing the Sparse Vector Layout

After gaining a clear understanding of layout concepts, we delved deeper into the library. One issue that was immediately apparent was that the existing code for generating *Laik_Layout* objects was only modified for creating a *Lex Layout*. This presented the challenge of discerning the current layout in use. The developer's implementation of a generic function signature facilitated the creation of a new specific layout object. However, there remained a need for a static helper function in relation to the existing layout. We could have opted to implement a generic signature, but we chose to introduce a new flag to signify the layout currently in use. Thus, we implemented our bespoke helper function within the same file as it was done for the *Lex Layout*. And there was no longer a requirement for the generic layout factory of LAIK, as shown in the following code snippet.

```
if (layout == LAIK_Lex_Layout)
{
  Laik_Range *ranges = coveringRanges_lex_l(n, list, myid); // helper function in regard to the lex
     layout
  Laik_Layout *layout = laik_new_layout_lex(n, ranges, 0);
}
else if (layout == LAIK_Vector_Layout)
{
  Laik_Range *ranges = coveringRanges_vector_l(list, myid, &map_size);  // helper function in regard
     to the sparse vector layout
  Laik_Layout *layout = laik_new_layout_vector(n, ranges, layout_data);
}
```

The helper functions known as *coveringRanges()* determine the required memory amount. The parameter "list" includes all global indexes pertaining to the process. The fundamental problem of gaps in the *Lex Layout* is located within the auxiliary function. The memory gaps occur because the helper function traverses locally owned ranges (interval of global indexes) and expands the initial range by the other ranges. Please comply with the following pseudocode.

```
result_range = list[0] // get first range
for range in list // iterate over the other ranges
  laik_range_expand(result_range, range);
```

We aim to illustrate expansion in this context with an example. We have two ranges: The first range is $[10;20[$ and the second range is $[8;17[$. If we expand the first range by the second in both directions, the resulting range is $[8;20[$. If we now expand it by a third range, say $[70;71[$, the resulting range will be $[8;71[$. Expanding in this way encompasses all global indices that are owned both locally and non-locally, provided they fall within locally owned ranges, as evidenced. To rectify this issue, a custom helper function was implemented. Initially, we considered avoiding further expansion of the function and instead adding the size of the ranges to the upper bound. Please refer to the pseudocode for more details.

```
result_range = list[0] // get first range
for range in list // iterate over the other ranges
    laik_range_add(result_range, range);

laik_range_add(result_range, range):
    result_range.to += range.to − range.from
```

In relation to the previously mentioned example, the allocated memory corresponds to the number of global indexes assigned to the process, resulting in a range of $[10;30[$. This achievement eliminates any gaps and fulfils our first goal. Another feature of our layout is that we only permit the allocation of one memory at any given time. Therefore, we have excluded the parameter "n" in the function *coveringRanges_vector_l()*. Conversely, the *Lex Layout* allows for the allocation and utilization of multiple buffers simultaneously.
Regarding our custom layout, additional information was necessary. However, for the *Lex Layout*, the application programmer need not provide any further details. Therefore, we made changes to the generic signature for creating layouts. Through brainstorming, we determined that the simplest solution was to include a pointer to the layout data in the *Laik_Data* object. This enabled the application programmer to set relevant information for the layout. We require two essential pieces of information: the length of the vector locally and the amount of external values the process will receive. After acquiring all pertinent details necessary for constructing the *Sparse Vector Layout*, our subsequent objective was to devise an effective technique for calculating the conversion of possessed global to local indices by LAIK. This method eliminates the user's need to complete this task manually, as was required when using the *Lex Layout*. Within the context of our newly proposed layout, the necessity for this mapping becomes even more apparent. If we refer to the example presented in Figure 3.7, we can observe that *process 1* owns global indexes 1, 4, and 5. Without any mapping, this would result in accessing the allocated buffer at the incorrect offset. In this example, thus, we would need to map the global indexes as following: $1 \rightarrow 0$, $4 \rightarrow 1$ and $5 \rightarrow 2$. Our approach to achieve this mapping is as follows: As previously mentioned, the parameter *list*

contains all ranges of global indexes belonging to the given process. Local partitioning is assumed during the calculation of the mapping. So, the concept is to have a primary object that stores all the intervals or ranges owned by a process. This is done by iterating through the *list* of locally owned ranges. For example, if the process has ownership of the four ranges: $[6;9[$, $[9;10[$, $[15;71[$ and $[75;80[$, then the process will store the following intervals to calculate mapping: $[6;10[$, $[15;71[$, and $[75;80[$. Following this, we combined neighbouring ranges to reduce the total size of information necessary for calculation. We also benefit from improved performance, as we require fewer iterations in the worst case than if we stored each range separately without merging them. Once we had calculated the mapping of all intervals, we proceeded to implement a critical function for calculating the offset within the allocated buffer for a given global index. This approach allowed us to quickly determine the correct offset. Please refer to the following code for more information:

```
int localOffset = 0;
for (uint64_t i = 0; i < map->size; i++)
{
    // if <globalIndex> was in interval <i>, we add the size of <globalIndex> subtracted by the lower
       bound of interval <i> to <localOffset>
    if (globalIndex >= intervals[i].from && idx_val < intervals[i].to)
    {
        localOffset += globalIndex - intervals[i].from;
        break;
    }
    // if <globalIndex> was not in interval <i>, we add the size of the interval to <localOffset>
    localOffset += intervals[i].to - intervals[i].from;
}

assert(localOffset >= 0 && localOffset < layout->localLength);
return localOffset;
```

The objective of this approach was to accumulate interval sizes until identifying the interval that contains the global index. Thus, we calculate the precise quantity of elements stored in memory. If we identify the appropriate interval, we simply add the offset of the global index minus the lower bound of the interval to the local offset. To elaborate, we refer to the example in Figure 3.7. Our objective was to create a mapping with the parameters $1 \rightarrow 0$, $4 \rightarrow 1$ and $5 \rightarrow 2$. For process 1, the intervals saved are $[1;2[$ and $[4;6[$. Let us consider mapping the global index 5. During the first step of the offset calculator, it is evident that 5 is not within $[1;2[$. Therefore, we increase $localOffset$ by $2 - 1 = 1$. In the second step, we check if 5 is within $[4;6[$, which is clearly verified. Thus, we update $localOffset \mathrel{+}= 5 - 4 = 1 + 1$. Therefore, the final value of the local offset is 2. Afterwards, we exit the loop and verify if the resulting offset fits within the allocated memory. Therefore, we have successfully mapped 5 to 2 in this given example. However, there is still more work to be done as we have yet to address global indexes, which are not internal to this process. Specifically, in relation to the HPGCG Benchmark, there are global rows that are not owned by this process. As previously discussed, if a process requires an element of a vector at the <i> index, but does not own the global row at <i>, then another process will be continuously updating the vector's value at <i>. Thus, a process must receive external values from the process that owns the global row <i>. The HPCG Benchmark stores these values at the end of the local entries of the vector. Consequently, we had to take this arrangement into account in our layout. To determine the order in which the HPCG Benchmark copies external values to the end of local entries, we printed out the values and the respective process from which they came. Upon careful observation, we have identified a prevalent pattern in which received values are arranged in ascending order of rank IDs and global indexes. Let us consider an example with three processes and process 0 requiring values located at global index 2 and 5 from process 1, and global indexes 1, 4, and 5 from process 2. The storage including the external values by process 0 is visually shown in Figure 3.8. With that, Process 0 can carry on performing its operations for the

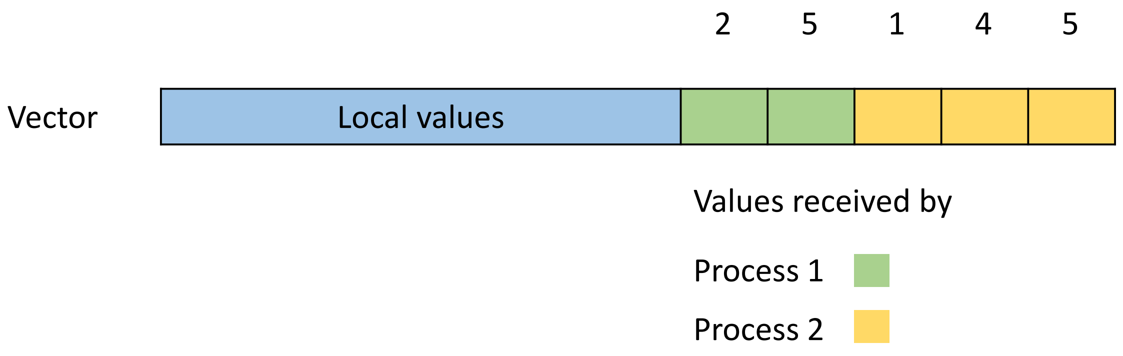

Figure 3.8: An illustration of how external values are stored at the end of the buffer.

current iteration after acquiring the external values at the proper offsets. To return to our implementation, we needed to consider global indexes of external values as well. After analysing the behaviour, we quickly achieved this. We adjusted the code above as follows:

```
int localOffset = 0;
bool wasInLocalRange = false;
for (uint64_t i = 0; i < map->size; i++) {
  if (globalIndex >= intervals[i].from && idx_val < intervals[i].to) {
    localOffset += globalIndex - intervals[i].from;
    wasInLocalRange = true;
    break;
  } else if (globalIndex < intervals[i].from) {
    // we caught an idx which is not locally owned by this proc
    // break so we do not iterate over all intervals
    break;
  }
  localOffset += intervals[i].to − intervals[i].from;
}
// we calculated the offset for locally owned global index
if(wasInLocalRange) {
  assert(localOffset >= 0 && localOffset < layout−>localLength);
  return localOffset;
}
// we have following layout: | Local values | external values |
localOffset = layout−>localLength; // offset starts there
// reset counter, if all values have been received
if(layout−>currentExternalValue == layout−>numberOfExternalValues)
  layout−>currentExternalValue = 0;
// return local offset for the given global index of an external value
return localOffset + layout−>currentExternalValue++;
```

As intervals of the mapping are calculated solely when local partitioning is active, it is possible to immediately determine whether the current global index being handled is external. This verification is executed in the "else-if-statement". It is necessary to check whether the global index is smaller than the lower bound of the current interval. This condition indicates that the global index is not present in the current and other intervals being iterated over nor in the previous interval. The aforementioned condition is implicitly true because if it were not the case, the prior iteration would already have exited the loop and returned the local offset of the global index. Subsequently, upon exiting the for-statement, the localOffset is then set to the local length of the layout object. Notably, calculating the offset for external values posed no complications. This is due to the fact that LAIK facilitates the exchange of values in the same manner as the HPCG Benchmark. We utilised a member of the layout object called "currentExternalValue" to indicate the current offset, incrementing it after a successful mapping of the external value global index. We also stored the total number of external values to exchange, thus enabling us to verify that the communication of values was complete, resetting the counter to 0. Namely, we implemented the *Sparse Vector Layout* following a logical and concise approach. This facilitated the successful implementation of the first version of the *Sparse Vector Layout*.

Subsequently, we had to modify the code for the HPCG Benchmark to incorporate the *Sparse Vector Layout*. Upon testing the updated version, we encountered a "Segmentation Fault" error. We reviewed the code in increments to identify the root cause and determined that certain portions still employed the mapping from the *Lex Layout*. We eliminated those segments and subsequently reran the program, resulting in output files that were identical to those from the original application. So our subsequent objective was to ensure that it is feasible to partition the world using our customized layout.

3.3.2 Shrink feature with the Sparse Vector Layout

After running the HPCG Benchmark with our customised layout and enabling the shrink function, we again encountered a "segmentation fault". Upon careful debugging, we identified the same issue as before: failure to remove the prior mapping due to the *Lex Layout*. Nevertheless, after addressing this problem, we observed that one of the processes had ceased to run. Initially, we suspected a deadlock had arisen. Upon discovering the source code, it became apparent that execution had halted, prompting us to investigate a potential deadlock. However, we determined that this was not the cause and instead identified an overlooked circumstance wherein a new layout object was being generated. A process does not need to allocate a buffer if it has no global indexes assigned to it. So we forgot to handle this case. This omission resulted in peculiar or undefined behaviour. Once the issue was addressed, the program no longer experienced an interruption at this stage. Subsequently, we encountered a similar occurrence during the redistribution of data from LAIK vectors to accommodate the new world size, resulting in a deadlock. To understand this problem, we will briefly explain the little optimisation that takes place when a LAIK vector is created. Consider the pseudo-code below.

```
// First, switch to the external partitioning, then switch to the local one
if(vector->exchangesValues)
   laik_switchto_partitioning(vector->values, externalPartitioning, LAIK_DF_None, LAIK_RO_None);

// Start with partitioning containing only access to local elements
laik_switchto_partitioning(vector->values, localPartitioning, LAIK_DF_None, LAIK_RO_None);
```

We begin by transitioning to external partitioning. This involves LAIK allocating memory for external values. It is worth noting that we use the *LAIK_DF_None* setting, indicating that processes should not exchange values. Once this is complete, we proceed with local partitioning, avoiding value exchange once again. With that, we achieve LAIK to check if an allocated buffer can be reused. If the memory is not reusable, LAIK will need to copy all values from the old memory to the newly allocated memory. It may seem costly, requiring LAIK to calculate the offset for each global index in both the old and new layout, and then copy all values. However, if the memory is reusable, LAIK will only verify whether each global index has the corresponding offset in the allocated buffer. In case it does not, LAIK would need to copy the values to the correct indexes within the reused memory. Therefore, our optimization entails the creation of a memory buffer by LAIK, which incorporates enough space for any external values required. Upon transitioning to the local partitioning method, wherein each process has sole access to its global indexes, LAIK can reuse this buffer. In contrast, suppose we initiate with the local partitioning method and then switch to the external partitioning. In that case, LAIK would have to allocate a new memory buffer and copy all values to it. Having prevented one allocation and copy, it is imperative for a layout to incorporate a function which can determine whether LAIK can reuse the memory of an old layout. Therefore, we had to create our own customized reuse function. With this in mind, we shall revisit the previous issue we encountered; namely, the deadlock that occurred while redistributing the data of LAIK vectors to conform to the new world size. The cause for the deadlock in this case was evident. We applied the previously mentioned optimisation when resizing the world. We specified that any removed processes would transition from the old local partitioning to the new local partitioning due to the world resizing. We also stated that for processes still executing, they would first switch to the new external partitioning and then to the new local partitioning. This was ultimately the root issue. The processes involved in the access phase differ in their initial partitioning, resulting in non-matching send and receive instructions. To rectify the issue, we decided that the processes not being removed would remain in their old local partitioning, even if this resulted in the omission of optimization. This ensured that all processes had the same access phase to the data container, enabling the calculation of matching send and receive operations. We believed that we had taken all necessary measures, but upon executing the program, we encountered an error stating "free() and malloc() unaligned tcache chunk detected". Hence, we conducted an incremental search to pinpoint the source of the error. It became apparent that we had overlooked the implementation of the aforementioned reuse function. Consider the following code snippet:

```
// do not reuse a vector, if the memory of the new layout does not fit into the old
bool new_totalSize_fits = new_layout.allocatedRangeCount <= old_layout.allocatedRangeCount;
// do not reuse a vector, if localLength changed
bool vector_size_changed = new_layout.localLength != old_layout.localLength;
if (!new_totalSize_fits || vector_size_changed) {
    if(!(vector_size_changed))  new_layout.mapping = old_layout.mapping;
    return false; // no, cannot reuse
}
```

As previously indicated, reusing memory is not possible if the required memory for the new layout exceeds that of the old layout. The second boolean value denotes whether the local length of a vector has changed, for instance, due to repartitioning. If this is true, a new buffer must be allocated owing to the diverse offsets and composition of values within the memory. Please keep in mind that the reuse function has been developed to align with the assertions and conditions in the current version of the

LAIK library. If we are unable to reuse a layout, we will check if the local vector length remains the same. If so, this indicates that repartitioning has been executed in our context. As previously explained, in this situation, we cannot begin with external partitioning due to optimization reasons. Therefore, we begin with local partitioning and switch to external partitioning when necessary. A new buffer was generated and all values were copied to it, an unavoidable procedure. However, the primary cause of the error was the lack of mapping assignment to the external partitioning, which resulted in the occurrence of unexpected behaviour. Upon assigning the necessary mapping to the layout for the external partitioning, we carried out testing of the resultant files. This marked the successful implementation of our bespoke layout while the world was being shrunken. The subsequent step involved detecting bugs during the expansion of the world while using our custom layout.

3.3.3 Expansion feature with the Sparse Vector Layout

We commenced this case with the mindset that we would need to invest some more time in debugging. However, upon carrying out a test case of broadening the scope from two to four processes, our HPCG-LAIK Benchmark yielded accurate outcomes. Subsequently, we analysed other test cases and uncovered a latent bug in the calculation of mapping intervals. This error was pinpointed by means of an assertion within the mapping calculation function. Our issue arose from the fact that the number of initialised intervals did not match the map's size. This was a particularly intriguing problem because it did not occur in our previous test case. Faced with this dilemma, we decided to print out the map, including all intervals. Through this exercise, we soon realised that the last interval contained random values, as it had not been initialised. This problem was due to a specific case and did not happen regularly. After resolving this issue, we introduced HPCG LAIK v1.2.
To summarise, we initially ported the HPCG benchmark to take advantage of LAIK's communication concepts. Subsequently, we implemented the part that allows dynamic adaptation of the world. Finally, we created a customised layout to match the composition and access pattern of the vectors in the HPCG benchmark. In the following Chapter, we will assess the effectiveness of our migrated HPCG version to LAIK.

4 Performance Evaluation

Now, we will discuss the performance evaluation. This work will primarily analyze three parts:

- Memory Consumption
- Computing Performance in GFlops/s
- The Effectiveness of Resizing the World

This review aims to compare the results of the original HPCG benchmark, our first version, HPCG LAIK v1.0, and the third version of the port, HPCG LAIK v1.2. During our benchmarks, we utilised an x86_64 architecture system with four sockets, each containing 24 cores. Additionally, each core has two hardware threads. CPU models are Intel(R) Xeon(R) Platinum 8360H CPU @ 3.00GHz. Furthermore, the system's clock frequency ranges from 1200MHz to a maximum of 3001.00MHz. The system has a 3 MiB L1d and L1i cache, a 96 MiB L2 cache, and a 132 MiB L3 cache. Our system employs NUMA (non-uniform memory access) as the memory architecture. There are four NUMA nodes in total, with certain processes sharing the main memory of their respective NUMA node. We will begin analysing the memory consumption of all three versions.

4.1 Memory Consumption

Another important consideration is the memory usage of each version. It is crucial to optimize memory consumption to reduce total memory usage and access to non-local memory, especially for systems with NUMA memory architecture, as problem sizes increase. This optimization can lead to better overall performance. In this evaluation, we consider the necessary memory allocated for the vectors used in the CG. It is important to note that the measurement was performed on a local size of $64,000$. As previously mentioned in Chapter 3.2.1, the user can specify the local dimensions for a process. Therefore, we set nx, ny, and nz (the number of local grid points) to 40. The number of local rows is then calculated by multiplying all three values. This was a crucial aspect, which we will discuss after describing the results. The CG uses a total of nine vectors. Memory usage is evaluated by adding the buffer size of each vector and multiplying it by the size of a double (8 Bytes). The results are visible in Figure 4.1. The

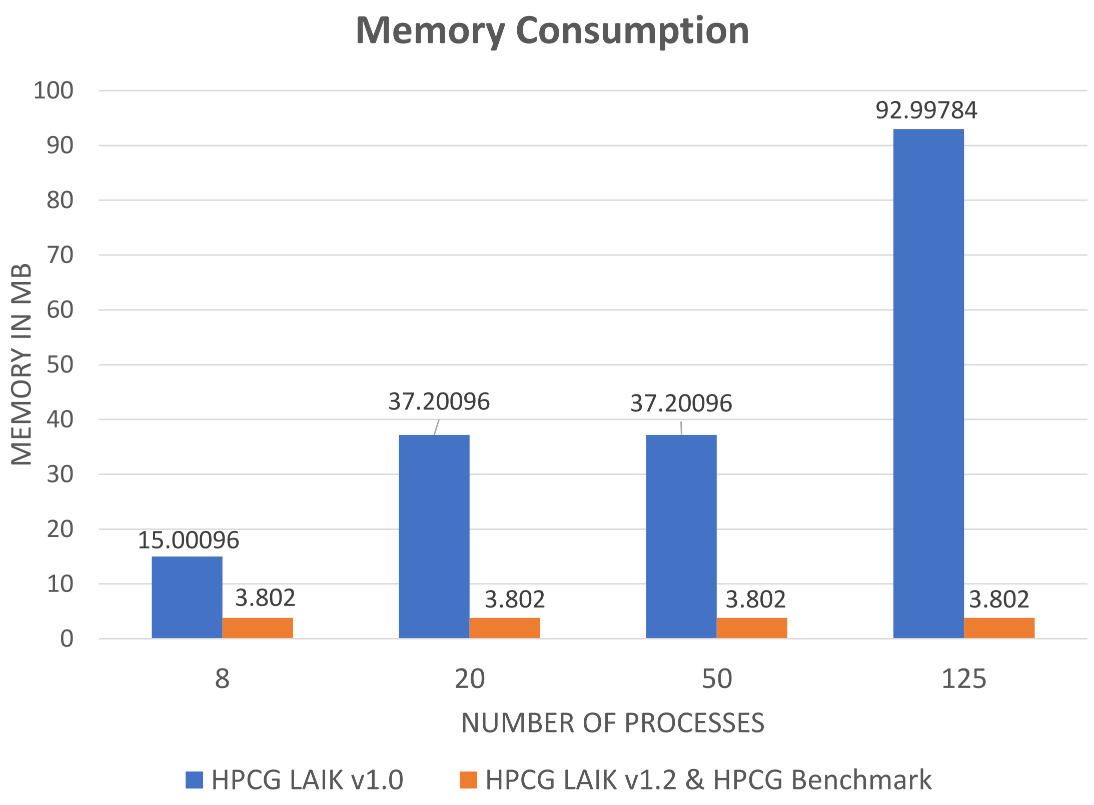

Figure 4.1: Memory consumption benchmark results.

y-axis shows the total amount of memory used in MB by the nine vectors per process. The x-axis shows the number of processes involved in the computation. The blue column represents the results of HPCG LAIK v1.0, while the orange column represents the results of HPCG LAIK v1.2 and HPCG Benchmark. It is worth noting that the latter two have the same results, which is intuitive. In HPCG LAIK v1.2, we implemented a custom layout that corresponds to the memory composition and access pattern of the vectors in the HPCG Benchmark. The initial observation is noteworthy: the HPCG LAIK v1.2 and HPCG Benchmark maintain a constant memory usage per process. This is because each process only allocates memory for its local part. Therefore, regardless of the number of processes, the memory usage per process remains the same. However, the memory usage of v1.0 is significantly higher than that of the other two versions. This is due to the usage of the *Lex Layout*. This layout allocates memory for every global index within the lower and higher bounds, resulting in redundant memory allocation. The amount of memory used by HPCG LAIK v1.0 depends on how the global rows are distributed or how global indexes are assigned to processes. Further benchmarks confirmed this. Running HPCG LAIK v1.0 with 30 processes yielded a memory consumption of 22.40 MB per process. With 40 processes, we observed a memory usage of 29.80 MB per process. We cannot claim that there is exponential growth in memory usage. However, memory usage will likely increase with larger problem sizes and a greater number of processes. For this benchmark, we utilised a relatively small local problem size. Multiplying the memory usage per process by the number of processes reveals that the memory consumption is acceptable. However, as the memory usage per process increases, the overall memory consumption can become excessively high. For example, when the local problem size is set to $1,124,864$, the memory usage for HPCG LAIK v1.2 and HPCG Benchmark is only 66.83 MB per process, whereas HPCG LAIK v1.0 has a memory usage of 265.90 MB per process when the benchmark is run with eight processes. We ran the benchmark with 20 processes and obtained a result of 662.71 MB per process. Multiplying this result by the number of processes, we obtain a total memory consumption of $13,254.2$ MB or 13.2542 GB. This indicates poor memory usage, as the other two versions would have consumed only 1.336 GB in total. Poor memory usage can lead to poor computational performance. If the main memory is full, the operating system will start swapping, which can significantly impact performance. In NUMA architectures, distributing processes across different NUMA nodes can lead to decreased computational performance when processes need to access non-local data. In summary, we optimized memory consumption with HPCG LAIK v1.2.

4.2 Computing Performance

As stated in the motivation, the HPCG Benchmark introduces a new ranking metric for HPC systems. It rates systems based on a weighted GFlops/s (billion floating operations per second) value. The authors of the official website of the HPCG Benchmark explain, that the "HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have an impact on the collective performance of these applications" [22]. Therefore, the HPCG Benchmark measures the performance of fundamental operations such as Sparse Matrix Vector Multiplication, Symmetric Gauss-Seidel smoother, Global Dot Product, Vector Update and Multigrid preconditioner. The authors provide additional information on why they selected those kernels for measurement. However, we will proceed with analysing our benchmark results. In our benchmark, we decided to set the global problem size to

512000. This implies that the sparse matrix has 512000 global rows, which will then be distributed amongst processes. Furthermore, the user can specify the runtime. We set the runtime to 3000 seconds or 50 minutes. The HPCG Benchmark calculates the time for performing one complete CG, then it calculates, how many times it will call CG by dividing the total runtime by the time for one call. With that, it is also very interesting to see, how many calls each version can perform. But we will come to it later. The diagram

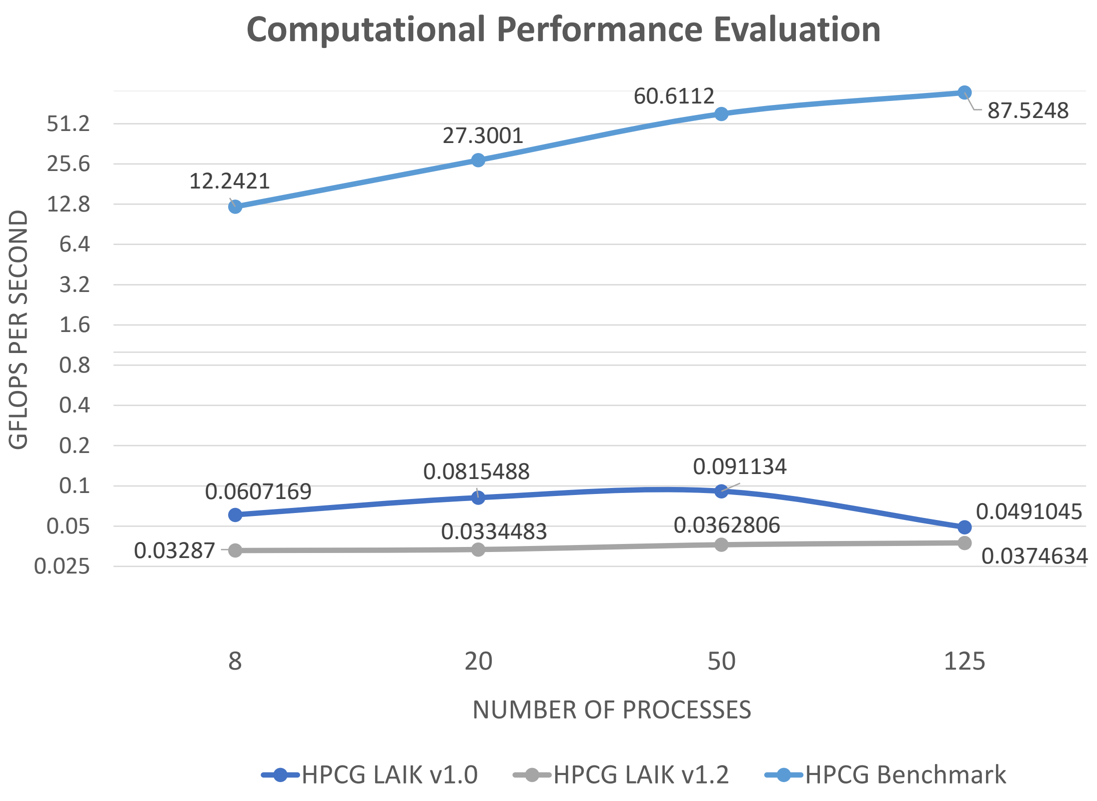

Figure 4.2: Computational performance benchmark results.

in Figure 4.2 shows the relationship between the number of processes participating in the computation and the corresponding GFlops/s value, with the y-axis representing the latter. The points on the line represent the achieved GFlops/s values. The light blue line shows the results of the HPCG Benchmark, the dark blue line displays the results of HPCG LAIK v1.0 and the grey line depicts the results of HPCG LAIK v1.2. It is immediately apparent that there is a significant difference in the results between the original HPCG Benchmark and our ported versions. This outcome was expected since we did not use the optimization called *Reservation API* by LAIK. The Reservation

API is an efficient optimization as it pre-calculates all steps for communication and data management. For example, the API allocates the memory buffer once based on the layout, and the pointer to the buffer remains constant. However, the pointer is returned by calling a specific function in the current state. This step is necessary because the memory buffer may have been automatically reallocated or deallocated by LAIK as we did not make use of the Reservation API. Additionally, the current state calculates a new layout object after each transition, resulting in repeated metadata calculations. This results in poor performance in both versions. There is a noticeable difference between the two ported versions as well. The developers of LAIK stated that the TCP2 Backend used in v1.2 is not optimized for data exchange, whereas the MPI Backend used in v1.0 is optimized. The TCP2 Backend was used because the MPI Backend requires a memory buffer that is allocated according to the *Lex Layout*. When using the MPI Backend with our custom layout, we observed buffer overflows. As the number of nodes involved increased, computational performance improved with each version. However, HPCG LAIK v1.0 did not exhibit this behaviour. When we ran the benchmarks with 125 processes, the computational performance was even worse than when running with only eight processes. We were trying to find out why. One explanation was due to inefficient memory usage, as described in the previous analysis. Initially, we considered oversubscription as a potential cause, given that the system only has 96 cores. However, the other versions did not exhibit similar behaviour, as previously mentioned.

As stated above, the benchmarks consisted of timed runs of 3000 seconds or 50 minutes. Figure 4.3 displays the number of CG sets performed for each version in the benchmark. As expected, the number of CG sets in the HPCG Benchmark is way greater than the number of the ported versions.

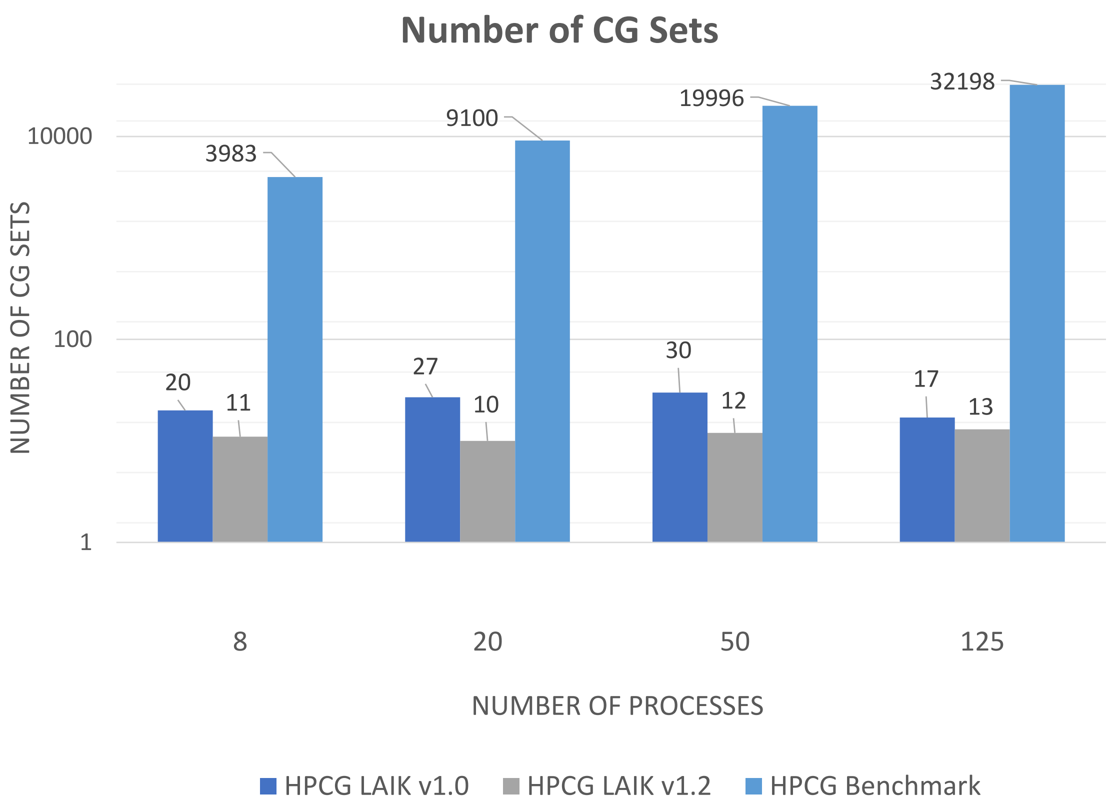

Figure 4.3: Number of performed CG sets.

4.3 The Effectiveness of Adjusting the Number of Computing Processes

The last evaluation aims to answer the question of when it is worthwhile to resize the world by adding new compute nodes. To achieve this, we measured the duration of all iterations before resizing the world and recorded the average time. We determined when it is appropriate to resize the world by calculating the difference in iteration duration before and after resizing. Considering the time required for repartitioning, we can determine the number of iterations needed to compensate for the resizing. If it takes too many iterations to compensate for the resizing, we should question whether the resizing was truly beneficial in this case. It is evident that reducing the number of computing nodes will generally lead to decreased performance. Therefore, we only considered expanding the world. The HPCG LAIK version 1.2 was used in this evaluation. The graph (refer to Figure 4.4) displays the benchmark results with

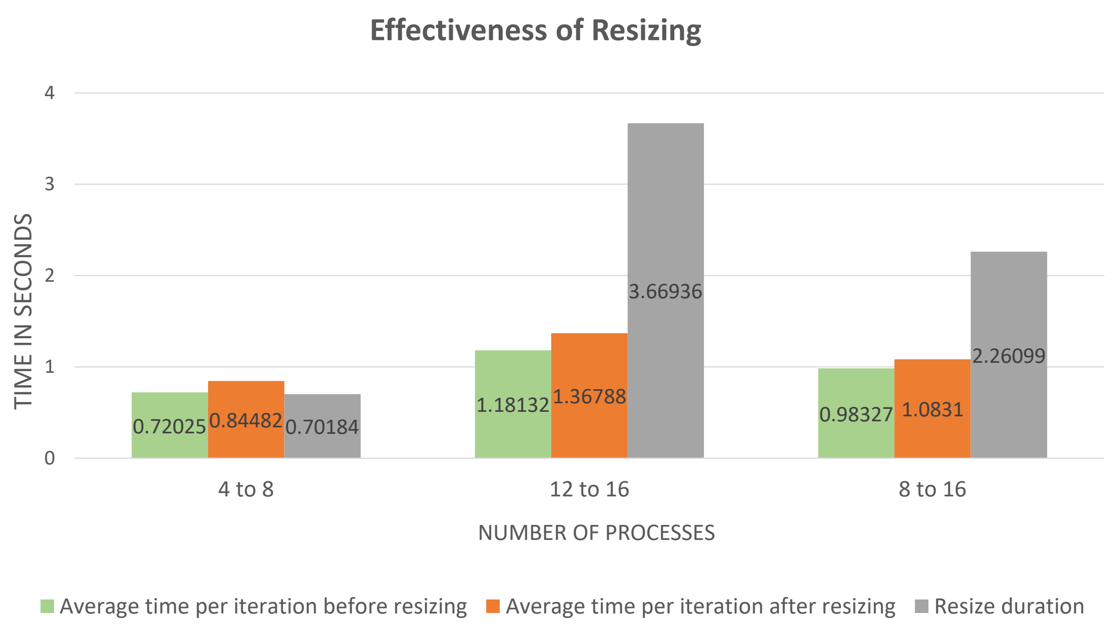

Figure 4.4: A comparison of computational performance before and after resizing, including the time taken for resizing.

the y-axis representing time in seconds and the x-axis representing the number of processes. The green column shows the average time per iteration before resizing, while the orange column shows the average time after resizing. The grey column represents the total time taken to resize the world. Upon comparing the average times, it was observed that the average time per iteration before resizing is smaller than after resizing. In general, this outcome would have been expected if we were shrinking the world. However, since we are expanding the world, we attempted to find out why this might be the case. We increased the number of iterations to obtain more stable results, but they remained unchanged. We were unable to determine the cause, and therefore, we could not determine whether resizing is advantageous in a given scenario. Expanding from 4 to 8 is relatively quick compared to expanding from 12 to 16 or 8 to 16. It could be beneficial. The latter cases may require more iterations to compensate.
In summary, although some of the benchmark results did not meet our expectations, we gained valuable insights into the performance of each version.

5 Conclusion and Future Work

In conclusion, this work has been quite successful. The problem at hand was to determine the feasibility of maintaining a feature that allows the node count to vary based on resource availability without requiring explicit implementation. Furthermore, we aimed to simplify communication by using abstract concepts to avoid explicit low-level communication code. We addressed these questions using LAIK. We comprehended the interface and concepts of LAIK and began integrating LAIK-specific code into the code base of the HPCG Benchmark. For instance, we abstracted communication code by leveraging abstract global index spaces. Enabling resizing was also a critical task, which initially seemed impossible. However, we gained insight into how to resize the HPCG Benchmark through the written examples in the LAIK repository. Furthermore, when we integrated our custom design, we needed to modify the LAIK library. We were thus able to contribute a new custom layout to LAIK. However, as we have analyzed, our ported versions of HPCG LAIK v1.0 and v1.2 showed significantly worse computational performance compared to the original HPCG Benchmark. This issue could be addressed by utilizing the *Reservation API* provided by LAIK. This optimisation would lead to similar results in performance measurements as the HPCG Benchmark. Another potential improvement would be to implement a custom layout for the sparse matrix. This would eliminate the need for the application programmer to implement the sparse matrix. However, an important factor affecting computational performance was the unoptimized version of the TCP2 Backend. This will require improvement in the future. As the average iteration time after resizing is not better than before resizing, it is necessary to investigate the effectiveness of resizing. In summary, although there is still work to be done, the porting of the HPCG Benchmark to LAIK was successful. LAIK's concepts for communication were utilized, and the world was resized.

Abbreviations

HPC	High Performance Computing
HPCG Benchmark	High Performance Conjugate Gradient Benchmark
LAIK	A Library for Fault Tolerant Distribution of Global Data for Parallel Applications
MPI	Message Passing Interface

List of Figures

Bibliography

[17] *Introduction to MPI*. Apr. 2017. URL: `https://carleton.ca/rcs/rcdc/introduction-to-mpi/` (visited on 12/12/2023).

[22] *HPCG Benchmark*. June 2022. URL: `https://www.hpcg-benchmark.org/` (visited on 12/12/2023).

[DL19] M. Dongarra Jack; Heroux and P. Luszczek. *HPCG-benchmark/HPCG: Official HPCG benchmark source code*. Mar. 2019. URL: `https://github.com/hpcg-benchmark/hpcg` (visited on 12/12/2023).

[Jos17] C. T. Josef Weidendorfer Dai Yang. "LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications." en. In: *Konferenzband des PARS'17 Workshops*. Gesellschaft der Informatik. 2017, p. 10.

[She94] J. R. Shewchuk. "An introduction to the conjugate gradient method without the agonizing pain." en. In: School of Computer Science, Carnegie Mellon University. Aug. 1994, p. 64.