Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Computer Science
2022 | LIU-IDA/LITH-EX-A--2022/002--SE

Integrating SkePU's algorithmic skeletons with GPI on a cluster

Integrering av SkePUs algoritmiska skelett med GPI på ett cluster

Joel Almqvist

Supervisor : August Ernstsson
Examiner : Christoph Kessler

External supervisor : Bernd Lörwald of Fraunhofer ITWM

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

As processors' clock-speed flattened out in the early 2000s, multi-core processors became more prevalent and so did parallel programming. However this programming paradigm introduces additional complexities, and to combat this, the SkePU framework was created. SkePU does this by offering a single-threaded interface which executes the user's code in parallel in accordance to a chosen computational pattern. Furthermore it allows the user themselves to decide which parallel backend should perform the execution, be it OpenMP, CUDA or OpenCL. This modular approach of SkePU thus allows for different hardware to be used without changing the code, and it currently supports CPUs, GPUs and clusters. This thesis presents a new so-called *SkePU-backend* made for clusters, using the communication library GPI. It demonstrates that the new backend is able to scale better and handle workload imbalances better than the existing SkePU-cluster-backend. This is achieved despite it performing worse at low node amounts, indicating that it requires less scaling overhead. Its weaknesses are also analyzed, partially from a design point of view, and clear solutions are presented, combined with a discussion as to why they arose in the first place.

Acknowledgments

This thesis was made possible through the hard work of the many previous contributors to the SkePU framework. Without them there would be no framework to extend in this thesis, and as such, I would like to extend a thank you to all them. Furthermore, the help, assistance and many SkePU contributions of my supervisor August Ernstsson and examiner Christoph Kessler has also been of particular help.

Outside of LiU, the Fraunhofer ITWM's GASPI-team has also helped greatly in this thesis. Both in the sense of being part in creating the used GPI framework, but also in providing assistance with it. In particular, the assistance of my external supervisor Bernd Lörwald has been of great help when using the GPI framework.

Contents

1 Introduction

This section presents the scope of this paper, that is its aim, motivation as well as structure.

1.1 Motivation

With clock speeds flattening out, the main way processor chip manufacturers attempt to achieve speedup is through more parallel cores [2]. This in turn changes how code has to be written in order to properly use the new parallel architectures, typically resulting in more complex code. This new programming landscape has created a need for tools which can use the underlying parallel architecture with a minimal increase in code complexity. The time it takes for the developers to write a program is an important part in any software development process. As such, if improved performance through parallelization comes at the cost of increased development time, it must be viewed as a trade-off rather than a net gain. Work has been done to highlight this issue [26, 2] and it further motivates high-usability tools which may reduce the development time while still using the underlying parallel architecture. Of course this is yet another trade-off as such a tool likely would sacrifice some performance for its usability, which is analogous to using higher abstraction-level languages rather than for example assembly.

One such usability-tool is SkePU, which through its algorithmic skeletons allows for code to be executed using different underlying models such as OpenMP, OpenCL and CUDA in order to best match the target architecture [16]. Thus the algorithmic skeletons may be viewed as an interface for the different execution models, which may themselves behave differently depending on their implementation. This modular approach is easy to extend, and one communication library which has shown promise is GPI [23, 37]. By combining these two it might let SkePU improve its performance on cluster architectures, which in turn might help the code complexity and performance dichotomy mentioned previously.

1.2 Aim

The aim of this thesis is to integrate SkePU and GPI in order to help determine whether this integration is likely to work and if it would be beneficial for SkePU. Which in turn would help make complex parallel architectures easier to use. In SkePU, the different models executing its algorithmic-skeleton-interface are called *backends*, and in this thesis a new prototype backend based on GPI was created. It was then compared to the already existing cluster backend [16], in order to analyze its performance and thus elicit strengths and weaknesses with its design. One aspect which is of particular interest is whether the new prototype overlaps its computations and communications, as previous evaluation of GPI required this for its strong performance [23].

1.3 Research questions

1. How does the prototype GPI backend compare to the currently existing StarPU-MPI one with regards to execution time?
2. Does the prototype's design allow for overlap in its computations and communications?
3. How well does the GPI backend scale compared to the existing StarPU-MPI backend?

1.4 Delimitations

SkePU is a large project with many more features than is feasible to include in a proof of concept prototype. As such, only three algorithmic skeletons are implemented, and only a subset of all their features are included. The precise delimitations of the prototype itself is available in Section 4.1. Furthermore, to limit the scope of the project, the prototype is only compared with the existing SkePU cluster backend. This means that no comparisons are done with pure MPI and also that no comparisons are done with a non-cluster version of SkePU. The MPI comparisons are of interest due to the dominating popularity of MPI as well as being the benchmark for previous GPI evaluations. Apart from this, the traffic and memory usage is not measured but rather only execution time.

Limitations of the approach

The approach taken in this thesis project, which is to evaluate GPI by comparing the backend based on it with another backend, comes with a few limitations. Firstly the comparisons are done with one layer of indirection through the backends themselves. As such the comparisons do not necessarily reflect the performance of the underlying tools, but rather this combined with the quality of the implementation. As the GPI backend is meant to serve as a proof of concept it lacks the maturity of the StarPU-MPI backend which has been under development for a longer time. Hence any comparison done between the two need to take this into consideration and attempt to distinguish factors which may arise because of it.

1.5 Structure of this thesis

Chapter 1 contains motivation, aim, delimitations, research questions as well as this section. Chapter 2 includes relevant background for the subject at hand, parts of which are taken from multiple disciplines. Chapter 3 lays out both the SkePU API in more detail and the purpose of SkePU. This chapter can be viewed as a stand-alone extension of Chapter 2 as its aim also is to provide background knowledge of the subject at hand.

Chapter 4 describes in detail how the prototype backend is implemented and highlights some important design decisions taken. It also covers how the benchmark programs were implemented and which part of their execution is likely to become their bottle-neck. This is discussed in Section 4.7 and is complemented by a high-level description of the programs given in Section 2.14. Afterwards comes Chapter 5 which covers technical details such as compilers versions, hardware as well as some aspects of how the measurements were made.

Chapter 6 contains the results of these benchmark programs. Chapter 7 discusses the results of Chapter 6 using the knowledge of the prototype's implementation from Chapter 5. From this the effects of the design on the generated results is discussed. Within this chapter the project and results are also discussed from a societal perspective. Chapter 8 links the analysis of the previous chapter with the research questions and proposes possible extensions of the thesis.

2 Background

This chapter outlines important concepts which are used frequently in this thesis. They may be divided into abstract concepts such as in Sections 2.1 to 2.9 and 2.14, whereas Sections 2.10 to 2.13 primarily covers programming tools. These concepts are vastly different and their only unifying feature is their usefulness for understanding this thesis. Notable with its absence from this chapter is any mention of SkePU, which has its own dedicated chapter following this one.

2.1 Flynn's taxonomy

There are multiple ways to structure a parallel architecture and as such there needs to be a way to distinguish between them. One way to do this is through their different control structure at the level of computer architecture. This is the purpose of *Flynn's taxonomy*[1], in which there are four categories: single instruction stream and single data stream (SISD), single instruction stream and multiple data stream (SIMD), multiple instruction stream and single data stream (MISD) and lastly multiple data stream and multiple data stream (MIMD). SISD corresponds to a single list of instructions applied with a single input, in other words the typical sequential execution model. MISD however uses multiple instructions on the same data set, an uncommon model which allows multiple processors to generate the same output. By comparing these outputs the system is able to detect errors and potentially recover from them, which is why the MISD model is most commonly used for fault tolerance.

SIMD is a vector-style execution model where a single, usually simple, operation is applied with multiple inputs generating multiple outputs. This pattern is common in computing where the same simple operation is used a large number of times, such as in graphics and machine learning. An important aspect of SIMD is its lockstep feature which mandates that all operations are done simultaneously, resulting in an atomic operation. Hence there may not be any data races in a SIMD-style execution.

The MIMD model is the most general one and in it multiple processing units receive both their own instruction and data. Modern multi-core computers are examples of MIMD-style execution whereas the older singe-core computers use SISD. It is possible to combine MIMD

[1]Further reading on Flynn's taxonomy:
`https://hpc.llnl.gov/tutorials/introduction-parallel-computing/flynns-classical-taxonomy`

with SIMD by letting the processors execute SIMD-style instructions in a MIMD fashion. This setup allows for multiple and parallel vector-style execution, resulting in a very large amount of simple instructions being executed quickly.

2.2 Memory models, fork-join and SPMD

Flynn's taxonomy offers one way to divide the structure of parallel code centered around the processors architecture, however this division fails to encapsulate other important aspects of a parallel computer. In particular the processors in a MIMD-model may share a common memory or they may have their own, resulting in two different system architectures. A *shared memory* model requires that all processors have direct accesses to the memory through for example a bus, which limits the physical distance between the two [35, pp. 34-37]. This makes it difficult to scale such a system to a large number of processors, which is why other models are typically used when doing this [35, p. 49]. Using *distributed memory* the processors are only able to access their own memory directly whereas the others are accessible through a network connecting all the processors. In this model accesses to unowned memory is more cumbersome and often limited by the speed of the network rather than the memory or its bus. As direct access is impossible, communication must be done through so-called *message passing* where messages with the desired information are sent between processors. Compared with the shared memory this process may be slow, but it scales better with increased numbers of nodes [35, p. 49].

A common programming model using the shared memory architecture is the fork-join style of programming. In it, programs start with a single master thread which then spawns new threads to complete some subtask, after which they terminate [6, pp. 23-25]. As such, the degree of parallelization is increased in the middle of the program while starting and terminating at a single-threaded section. Using the shared memory the threads are able to manipulate the same data structures and instantly read data created by another thread. The downside of this style of programming as previously mentioned, is the need to synchronize the thread's memory accesses to avoid concurrency issues. Yet another one is the prerequisite of a shared memory architecture, which may prove difficult to scale to a large number of processors. However despite these limitations the fork-join style of programming is common and the model of choice for *POSIX threads* (pthreads), which in turn is used by UNIX systems[2].

Another programming model is *single program, multiple data* (SPMD) in which multiple instances of the same program are started simultaneously but they all work on their own data partition. In such a system there is no ”main instance” responsible for the progress of all the others, instead the program terminates once all instances are finished. During the execution of the program communication may be done between the instances either through a shared memory or message passing. However as every instance is meant to work on their own separate partition of the data during SPMD it more closely aligns with the message passing architecture and is more commonly used with it. A popular tool using the SPMD model is *Message Passing Interface* (MPI) which is commonly used on cluster computers, which lack a shared memory and use the message passing architectures. For a further explanation as well as evaluation of the SPMD model see the paper by Kamil and Yelick [28].

2.3 Computer clusters broadly

Computer clusters are a large collection of computers with a single unifying interface allowing the programmer to use the capabilities of multiple computers. There are numerous reasons to use a computer cluster and depending on what problem they aim to solve their structure may vary quite a bit. A common cluster type is the *capacity cluster* which allows for a number of independent small scale programs to be run. Such a cluster is typically used to host servers

[2]The pthreads API by IEEE: `https://pubs.opengroup.org/onlinepubs/9699919799/`

of different types and need not have fast communication between its nodes, the Ethernet may suffice. The cluster type which is the most relevant for this thesis however is the *capability cluster* which is intended to handle a single very large problem. Unlike the capacity cluster its different parts need to communicate to a much larger degree as they all work on the same problem. Hence minimizing the latency is crucial, meaning both that the cluster may not be spread out geographically and that e.g. Ethernet is insufficiently fast. Capability clusters are commonly used in scientific computing and in the field of High Performance Computing (HPC) generally and is what the word *cluster* will refer to in this thesis here forth.

The principle of clusters is to scale horizontally, that is to use more computers rather than better computers. This is analogous to the on-chip situation where recent speedup primarily has been achieved by adding more processors to a single CPU [2]. Common among both of these examples is the fact that they achieve their increased capabilities by executing more code in parallel. As such the strength of a cluster is its capability of executing an enormous amount of subtasks in parallel across multiple different computers. Using this technique the top cluster computer of the ”Top 500” June 2021 list is able to achieve up to $4.4 \cdot 10^{17}$ floating point operations per second with about $7.6 \cdot 10^{6}$ processors [3].

2.4 Cluster architecture

The structure of a cluster is a set of interconnected stand-alone computers called *nodes*, which are accessible through a unifying interface. The nodes contain their own memory, cache and CPU, the last of which contains multiple processors called *cores*. Cores within the same node share the same main memory and as such are able to use techniques such as fork-join. This is however not possible between nodes, meaning that clusters are a sort of hybrid system with shared memory within in a node and distributed memory from the viewpoint of the whole system. Typically, communication between nodes is done through message passing.

In order for messages to be sent between nodes they need to be connected, the naive way to do this is shown in Figure 2.1 where every node has a direct edge to all the others. However as the number of nodes grows this becomes unsustainable and a more complex topology is required [35, pp. 37-43]. One such topology is the tree based one shown in Figure 2.2 where every leaf corresponds to a cluster-node and the other tree-nodes are switches. If a message is sent in this network it always starts and ends at a leaf, meaning that in the worst case a message must traverse the height of the tree twice. As the tree's height grows logarithmically with the amount of nodes, this path is guaranteed to be fairly short. However this graph has a fatal flaw which is the amount of messages which must pass through the upper nodes of the tree, and the root node in particular. If the messages' origin and destination are uniformly distributed then half of them have to pass through the root node, which is likely to overload it. To improve on the tree design many clusters use the so-called *fat tree* which has increased capabilities at the higher nodes in the graph [32]. This is achieved by having multiple root nodes and more edges, which together splits the traffic among more nodes to avoid overloading the ones high up the tree. Yet another extension made in some fat trees are to use higher bandwidth cables near the roots as more traffic is expected along these edges. An illustration of a fat tree is provided in Figure 2.3 and in it the colored edges could be interpreted as higher bandwidth cables, if such an extension was made. The downside of the fat tree compared to the regular tree is that it uses both more and potentially more expensive hardware. Despite this topologies based on the fat tree are common among modern clusters.

Lastly all nodes in a cluster need not be equal in terms of hardware, in such a case the cluster is said to be a *heterogeneous cluster*. Accelerators such as a GPU may be given to a few nodes to specialize them for a certain kind of computations. In such a system it is important

[3]List of top performing supercomputers in June 2021 by Top 500:
`https://www.top500.org/lists/top500/2021/06/`

to make sure that the appropriate node type receives the correct subtask, a consideration not needed in a homogeneous cluster.

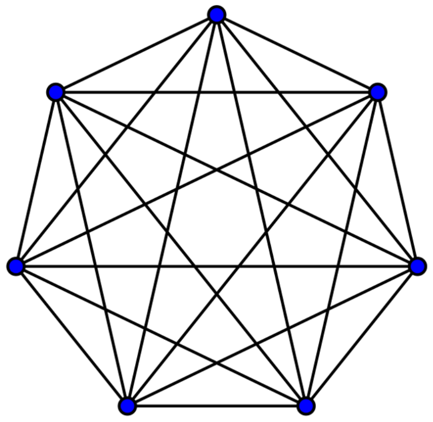


Figure 2.1: A complete graph where every node has a direct edge every other node.

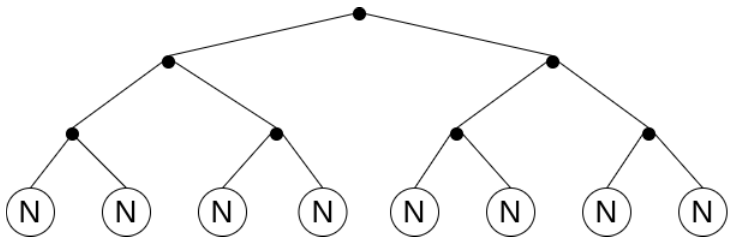


Figure 2.2: An example of a cluster using a tree topology. Every leaf in the graph corresponds to a node in the cluster and the black dots to switches.

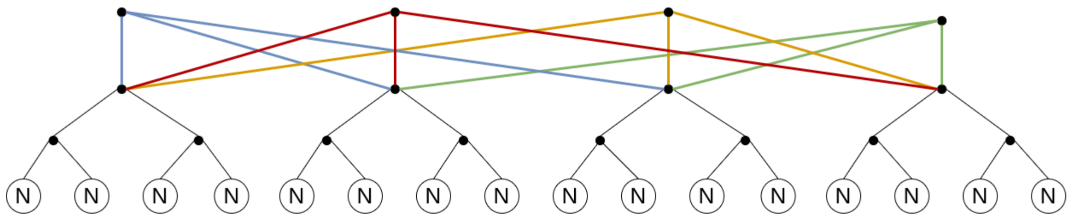


Figure 2.3: A fat-tree topology of a cluster where the leaves correspond to cluster-nodes and the black dots to switches. Some edges are colored primarily for clarity sake, although they could be viewed as higher bandwidth cables.

2.5 Speedup and linear scaling

Speedup in parallel programming is a concept for measuring how the execution time differs when additional processors are used to execute the program. There are two kinds of speedups discussed in this thesis and they mainly differ in what they consider to be the baseline value of comparison. In *relative speedup* a time is measured against how well the parallel implementation does with a single processor whereas *absolute speedup* uses a different solution which is expected to run on a single processor [40]. The difference between the two is thus that the baseline in relative speedup is the same parallel program using a single processor whereas in absolute speedup it is a properly implemented single threaded solution. Furthermore this single-threaded solution must use the best currently known algorithm for the problem. These base-points are then compared to the parallel program's execution time using p processors, which is denoted $T(p)$. Then let T_{seq} denote the sequential solution's execution time and S the speedup. Then we get the following expressions:

$$S_{rel} = \frac{T(1)}{T(p)}$$

$$S_{abs} = \frac{T_{seq}}{T(p)}$$

Furthermore $T_{seq} \leq T(1)$ as the best sequential solution may never be slower than a parallel solution using a single processor. If this would be the case then the parallel solution would become the optimal sequential solution, contradicting its definition. Which means that the following must also be true:

$$S_{abs} <= S_{rel}$$

Both of these measures are useful but what they highlight is different and as such it is important to distinguish between them. Relative speedup may be used to demonstrate improvements within a program whereas absolute speedup could be used to clearly demonstrate that a given problem is effectively solvable using a parallel implementation. Naturally their use-cases are not limited to this, but it illustrates how they may be used differently. As absolute speedup requires the creation of another solution it is typically easier to use relative speedup.

Regardless of which kind of speedup is being measured the optimal scaling it can detect is *linear scaling*, which is when doubling the amount of processors halves the execution time. Any scaling higher than this would imply that halving a processor's workload would reduce its execution time by more than half. This is possible in fringe cases where the changed workload improves the cache's hit-rate. But as the hit-rate may only improve to a certain point this super linear speedup is limited in duration. Due to this super linear speedup is more of a curiosity than something actively striven for and linear speedup acts as the gold standard. But even this is not truly achievable in practice, as it may only occur when adding more processors does not entail a larger overhead. This is not possible as the nodes necessarily need to communicate with each other, incurring at minimum some overhead. The highest aspiration of any achieved speedup is hence to be as close to linear speedup as possible.

2.6 Amdahl's law

A program may be divided into multiple parallel and sequential sections, and by accumulating all the time spent in either one of them we can describe their relationships thus: $T_{tot} = T_S + T_P$. In a sequential program $T_P = 0$ whereas for a parallel one $T_P > 0$. Typically $T_S > 0$ holds for parallel programs, certain models such as fork-join even mandates it. Adding more cores to handle a problem will serve to speedup the parallel section but not the sequential one, assuming that we have n cores and a linear speedup we then get the following expression:

$$T_{tot} = T_S + \frac{T_P}{n}$$

Even in this toy example which assumes perfect parallel scaling with no overhead cost, the execution time is still limited by the sequential section. To put this in more formal terms let $p = \frac{T_P}{T_{tot}}$, $1 - p = \frac{T_S}{T_{tot}}$ and $S(n)$ denote the speedup when using n cores. Thus $p \in (0, 1]$ and the final expression becomes:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Which implies:

$$S(n) < \frac{1}{(1-p)}$$

This is known as *Amdahl's law* and it illustrates how the speedup of a program is limited by its sequential section. Assuming that the sequential code takes 40% of the execution time any speedup due to parallelization may never reach 60% and thus the execution time will never go below 40% of what it is at $n = 1$. As such an important idea implied by Amdahl's law is the diminishing effects of adding additional cores to a program [6, pp. 33-34].

2.7 Data containers

In this thesis the term *data container* refers to a structured encapsulation of a data set. Such a container may be of a certain dimensionality such the one-dimensional vector and two-dimensional matrix. But it could also have a varying dimensionality such as the N-dimensional tensor. Apart from this factor a container may structure the data in a specific manner as to make it faster to use. Memory accesses which are increasing and sequential are significantly quicker than those with large gaps between them. This is due to the cache reading the chosen address and some subsequent addresses. Hence a data container may try to structure itself in such a way as to make sure that its elements are accessed sequentially, even if the elements are not necessarily sequential from a logical point of view. Furthermore they may have different structures as to allow for quicker insertions and size increases. For example a container which puts all elements sequential in the memory would have quick sequential access. But it can not increase in size while keeping this feature without moving all of its elements into a new larger contiguous memory space. It is of particular note to understand whether a matrix implementation stores its rows or columns contiguously as the choice is arbitrary, but it significantly impacts traversal speed. This illustrates the importance of knowing how a data container is implemented and how it stores its elements,s in order to ensure good performance.

2.8 Computational patterns

In computer science there are many computational patterns, which are a structured way to perform computations. The pattern itself determines how the computations are applied and how the output is chosen, but the actual computations are left unspecified. In order to instantiate a computational pattern the computations must be defined, which is done by the so called *user-function*. The user-function is a function which takes in a certain number of arguments depending on the pattern, and returns a value. Thus it offers a way to specialize the pattern in order to fit the particular issue at hand. Furthermore, implementations of a pattern must ensure that the user-function is executed in accordance to certain pattern-specific dependencies. By adhering to them, an implementation is able to parallelize parts of its execution. This section covers the patterns which are relevant for this thesis, many of which are fairly well known.

Map

The most basic map pattern operates on a data set and transforms it by applying a user-provided univariate function on every element and replacing it with the generated result. A requirement for the map pattern is that there are no data dependencies between the elements, which results in the computations being so called ”embarrassingly” easy to parallelize. Mathematically every element i in the set C is thus transformed the following way.

$$C_i = f(C_i)$$

Historically the map pattern has existed as a concept for decades, for example Lisp has it implemented as a primitive [4].

Reduce

The reduce pattern works on a data set and returns a scalar value. It uses a binary associative and commutative function which is applied to an accumulator and element pair until every element has been included once. Furthermore the accumulator may also have an initial value. If we let A be the accumulator, f the user-function and C a data container with N elements we can put this mathematically as:

$$A_1 = f(A_{init}, C_1)$$
$$A_2 = f(A_1, C_2)$$
$$A_N = f(A_{N-1}, C_N)$$

The generated output of reduce is thus A_N. Note however, that an actual implementation of reduce may calculate the partial states of A in a different order than the definition above. An important aspect of reduce is that its user-function is associative and commutative which allows for an arbitrary accumulation order. Parallel implementations in particular may leverage this by splitting the accumulation into smaller parts which are run in parallel. Then they may accumulate the partial-accumulations to get the final result. Lastly, similarly to map, the reduce pattern has been used for a long time and is also implemented in Lisp [5].

MapReduce

In this thesis, MapReduce refers to a computational pattern which combines both map and reduce. Functionally there is no difference between a map followed by a reduce and a MapReduce call but despite this there are good reasons to use it. The first of which is that it is easier for the programmer to use a single computational pattern rather than two. Anecdotally the use of a map followed by reduce proved to be common enough to merit the creation of this combined pattern [11]. The second reason is that the combination of the two patterns is able to be executed quicker than a map followed by a reduce. This can be achieved by eliminating some of the middle steps, in particular whenever the map function generates a value the reduce function may immediately consume it. This leads to a much higher data locality than the original approach while also removing the need to store the elements generated by map. Hence MapReduce is able to improve performance while using less memory compared to a two step approach of map plus reduce. The pattern requires both a map and reduce function, both of which must fulfill the same data dependencies and use the same arguments as their standalone counterpart. If we let A be an accumulator and C a data container of size N, then MapReduce can be described mathematically as follows:

$$A_1 = f_{red}(A_{init}, f_{map}(C_1))$$
$$A_2 = f_{red}(A_1, f_{map}(C_2))$$
$$A_N = f_{red}(A_{N-1}, f_{map}(C_N))$$

The return value of a MapReduce is a single element just like in reduce and as such in the example above A_N would be the returned value.

[4]The syntax of Lisp’s map implementation: `http://clhs.lisp.se/Body/f_map.html`

[5]Lisp’s implementation of reduce: `http://clhs.lisp.se/Body/f_reduce.html`

Cartesian product

This pattern is based on the definition from set theory where the cartesian product of two sets is every possible combination of their elements. Furthermore the pattern also applies a binary function on every combination. Given the two sets $\{A, B\}$ and $\{C, D, E\}$ and the binary function f the pattern thus calculates:

$$f(A,C), f(A,D), f(A,E)$$
$$f(B,C), f(B,D), f(B,E)$$

Scan

Scan takes a one dimensional data container, a binary associative function and generates a new set where every element is an accumulation of the previous elements. Given the binary associative user-function f, input element E and resulting element R, we get:

$$R_i = f(E_i, f(E_{i-1}, f(E_{i-2}, f(...))))$$

Or, equivalently:

$$R_i = f(E_i, R_{i-1})$$

There are two different versions of this pattern depending on whether the current element is used in the calculations for the output or not. The example below does use the current element and is thus inclusive whereas the example below is of the exclusive version:

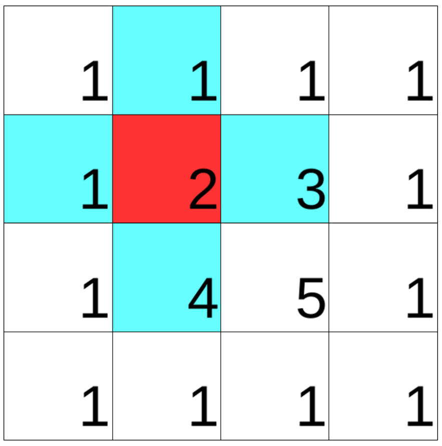
$$R_i = f(E_{i-1}, R_{i-1})$$

A question which arises from the description so far is how the pattern should behave when there is not enough elements to give to the binary function. For example using the inclusive definition above, if the indexing starts at one, it means that E_0 does not exist and thus that R_1 is not properly defined. How the scan pattern should behave in these cases depends on the implementation, one solution would be to simply output the provided argument. However in the exclusive case R_1 does not have even a single existing argument, as such another base case would have to be added. Examples of these might be to set $R_1 = E_1$ or to set it to some predefined value such as 0 or 1. How these base cases are handled is important but how this is done is more closely tied to the implementation rather than the computational pattern itself.

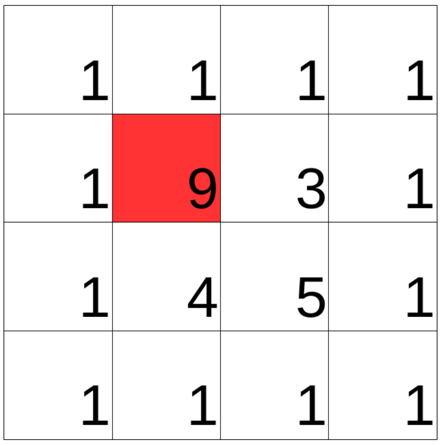
Stencil computations

Stencil computations are a specific type of data-parallel computations applied on a dataset where the output depends on the neighboring elements. What qualifies as a neighbor depends on the implementation as well as the dimensionality of the data. For example, in a two-dimensional grid a neighbor might be any element which is reachable within X non diagonal steps from it. The input to the user-function is thus an elements whole neighborhood while its output is a single value which the element assumes. Doing this for all elements thus transforms the whole set. An issue with stencil computations are the complex data dependencies which arise due to an element potentially belonging to multiple neighborhoods while simultaneously needing to be transformed itself. Figure 2.4 illustrates how disregard to the data dependencies would generate incorrect results. One way to solve this issue is to use an extra buffer to store the elements, resulting in increased memory usage but making the problem trivial to parallelize. Another way is to determine specific execution patterns which allow for high parallelism at the cost of relaxing the constraints somewhat. A specific example of this is the Gauss-Seidel update schema which works at a two dimensional grid with every neighbor being defined as an element with a non diagonal distance of one. Every second diagonal may then be executed in parallel and so may all the remaining diagonals, although not the two of them simultaneously. Note however that in this schema one of the two sets of diagonals will use neighborhoods containing only modified elements, but despite this it may still be a useful schema.

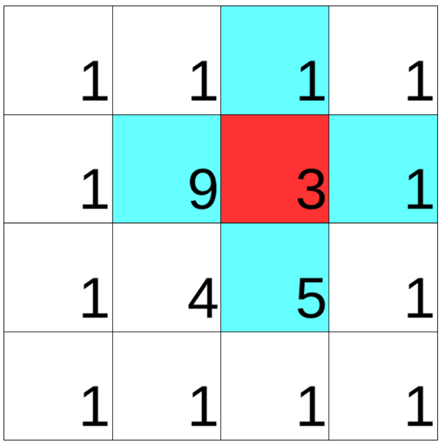
In general, stencil computations are good for emulating systems where changes propagate through neighbors, such as in fluid-systems and image filtering. The execution patterns of stencil computations vary more than for the previously mentioned computational patterns. Not only does it have another parameter in the form of the neighborhood but how the data dependencies are handled depends on this very choice. As such the execution pattern of the user-function itself may vary between different stencil computations, unlike most other computational patterns.



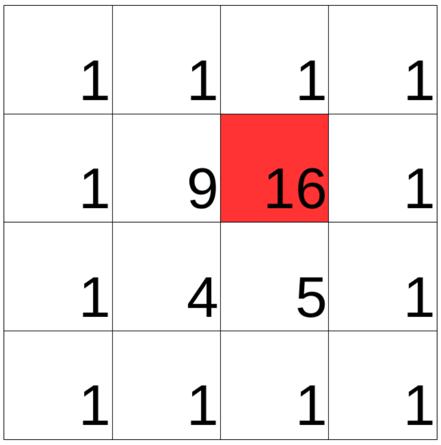
(a) Step 1 - The blue elements are the the red elements neighbor.



(b) Step 2 - The new value is created



(c) Step 3 - The previously created value is now a neighbor.



(d) Step 4 - The generated value is incorrect.

Figure 2.4: An incorrect stencil computation which does not respect its data dependencies.

2.9 Algorithmic skeletons

Algorithmic skeletons, henceforth also referred to as skeletons, are a high-level abstraction aimed at making it easier to use parallel hardware. The concept arose to alleviate the increased complexity of writing parallel software in 1989 [8] and the concept has since been used with multiple different hardware such as multi-core CPUs [33], GPUs or clusters [15]. The idea of a skeleton is that they apply a user-function in accordance with a computational pattern. The benefit of this is that it abstracts away from many aspects of the parallelization-process while simultaneously providing a well tested framework. As such the user does not need to deal with any aspects of the parallelization, which can be an error-prone and time consuming process, especially if the user is not an expert at this task. By using skeletons the user need only to chose the appropriate skeleton and to write the user-function. This is however a trade-off as a hand-optimized implementation without any skeletons is generally expected to execute quicker, but it takes longer to write. In some sense this is analogous to the trade-off between higher- and lower-level languages where one might substantially reduce development time at the cost of performance by using Python rather than Assembly. Apart from abstracting away the parallelization issues a skeleton may also hide other complexities such as the communication aspect within a cluster. As all of these abstractions attempt to hide the hardware from the user they also improve portability as the machine specific code is hidden within the skeleton. Thus the high-level code written to the skeleton may more easily be reused between different hardware. To summarize, a skeleton is a high-level concept which aims to make parallel execution easier by abstracting the hardware away, whether it be CPUs, GPUs or clusters of both.

2.10 OpenMP and scheduling

OpenMP is a framework for parallel execution on a single node using shared memory and the fork-join execution style while being available for FORTRAN, C and C++[6] [6, pp. 23-25]. It uses compile-time directives to generate explicit parallel regions of code, meaning that code outside of them is sequential. Furthermore there are directives to create parallel constructs, the most notable of which is the parallel for-loop. There are multiple ways to customize the behavior of these constructs and regions, such as whether the created variables are shared or not and importantly how the scheduling is done. There are two classes of scheduling, *static scheduling* and *dynamic scheduling*. In the static one every thread is given a set of tasks at compile time, whereas for dynamic scheduling the tasks may be redistributed during runtime. Thus the dynamic scheduling makes it possible to discover load imbalances while the program is executing. For example if a large number of threads are all given N iterations of a task, but every iteration may take either one time unit or a hundred it is likely that some threads will finish much earlier than others. With dynamic scheduling the threads which finish early may offload some of the work by taking some iterations of threads that got more of the time consuming ones. This flexibility does however come at a cost in the form of overhead and depending on how large the work imbalance is this overhead may not be worth the benefit it provides. In such cases static scheduling is superior. The decision between the two thus largely depends on how large the work imbalance is, and if this varies between constructs OpenMP allows for them to be scheduled differently. An example of this and the OpenMP syntax in general is demonstrated in Figure 2.5.

2.11 Message Passing Interface (MPI)

MPI is a standardized message passing interface intended for parallel execution. As it is only an interface there exists multiple implementations of it for the three officially supported languages C, C++ and FORTRAN. Furthermore MPI does not require a shared memory address space and executes in a SPMD fashion, the latter of which makes it suitable for cluster and HPC usage. It also offers intra-node parallelization by running multiple instances of the program on a single node, using the shared memory to communicate when possible.

[6]An OpenMP tutorial given by Ruud van der Pas of Sun Microsystems at Nanyang Technological University: `https://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf`

Figure 2.5: An example of multiple scheduling approaches within the same parallel region using OpenMP and C++.

```
// Do single threaded work here

#pragma omp parallel num_threads(8)
{
        #pragma omp for schedule(dynamic)
        for(i = 0; i < N; i++)
        {
                if(random() )
                {
                        computational_intensive_function();
                }
                else
                {
                        trivial_function();
                }
        }


        #pragma omp for schedule(static)
        for(i = 0; i < N; i++)
        {
                computational_intensive_function();
        }
}
```

Despite this however, combining MPI with another parallel execution model seems to be quite common according to Laguna et al.'s survey of open source MPI HPC projects [30]. In it 74.5% used MPI in combination with other parallel software, many of which offered intra-node parallelization through threading rather than MPI's technique. Part of the explanation for this is the usage of GPU accelerators which are better used with parallelization tools built to support them such as CUDA. But even more generally the paper offers the explanation that internal parallelization using non-MPI tools might be due to them fitting better with their cluster architecture. It is clear however that MPI's internal parallelization model seems less popular than MPI itself, which is widely used in the HPC field.

MPI offers a plethora of different communication methods such as: send-receive (one to one), broadcast (many to many), scatter (one to many), gather (many to one) and group communications. One way to use these tools is in a two-sided manner which requires every type of send call to be matched by a receive call in the remote process as illustrated in Figure 2.6. In it we can also see one of the most common synchronization directives, the *barrier* function call. It takes in a group of MPI processes as an argument and then forces every MPI process in said group to wait until every member has reached the barrier call. This call is used to synchronize the processes at the cost of making some of them wait, which in turn may slow down the execution speed of the program, especially if there is a load imbalance.

Apart from this, MPI also offers one-sided communication directives where the explicit receive is not necessary. First we need to define the concept of *windows*, which are contiguous memory segments owned by a node but visible and usable by remote nodes within the same group. To use the one-sided communication directives the code segments are divided using function calls called *fences*, and the code segments they divide are called *epochs*. Within these the windows are usable by the other nodes without the need of any explicit call by the owner of the window. Conflicts may occur if multiple writes are done within the same epoch or if both writes and reads are done during it. The epochs are bound to a window, which in turn are bound to a group of MPI processes, as such every process need not participate in an epoch. The end of an epoch is marked by a fence which a participating process may not pass until every participant has reached it. Thus a fence works similar to a barrier, and if there is a load imbalance they may also lead to slowdowns just like barriers. An example of the one-sided communications may be seen in Figure 2.7.

Figure 2.6: Example of two-sided communication in MPI.

```
if(node_id == 0){

        MPI_Send(..., node_1, ...);

}
else if(node_id == 1){

        MPI_Recv(..., node_0, ...)
}

MPI_Barrier( ... )

// do work after node 0 has sent the data
```

Figure 2.7: Example of one-sided communication in MPI.

```
MPI_Win_fence( ..., window_1, ... )

if(node_id == 0){

        MPI_Put(...);
}

MPI_Win_fence( ..., window_1, ... )

// do work after node 0 sent the data
```

2.12 GASPI and GPI

Global Address Space Programming Interface (GASPI) is an message passing API for clusters created collaboratively by the following partners: T-Systems SfR, Fraunhofer ITWM, Fraunhofer SCAI, KIT, TU Dresden, Scapos AG, FZ Jülich, DLR and DWD [1]. GASPI offers more fine-grained control of the communication and synchronization process when compared to, for example MPI and its fence-method. Furthermore, it also provides fault tolerance mechanisms and a logical wrapper for different memory types such as the GPU's memory and the main memory, a scheme known as heterogeneous memory [1]. GASPI is similar to MPI in how it is used; it is neither a language nor an extension of one but rather an API which may be used within an existing programming language. An implementation of the GASPI API is Global Address Space Programming Interface (GPI) which was also created at Fraunhofer ITWM. GPI focuses on overlapping computations with communications with its one-sided communications directives in order to reduce the time spent waiting. These directives differ from MPI in that they may execute at any point in the program's lifetime as long as the communication buffers, called *segments*, have been initialized. As such, GPI does not use a concept similar to MPI's fence nor epoch, instead the communications are synchronized through notifications. These may be sent by any node; to any other node, and before modifying its segment a node may wait for such notifications. The difference is subtle but it allows for less tightly linked communications, for example in MPI if a node performs a read it must wait for its request to finish before leaving the epoch. In GPI such a wait would not occur unless explicitly stated as illustrated in Figure 2.8. The effect of this is that GPI allows for many concurrent requests, some of which may span over what would be multiple epochs in MPI. The actual transfers within GPI are based around segments, which are chunks of memory that are globally visible for their group. These segments may then be read to and from by any node and are the basis for all communications within GPI.

The motivation behind the creation of GASPI was MPI's insufficient performance with a large number of nodes [24]. To achieve this in GPI it attempts to overlap computations and communications, while avoiding so called "bulk-synchronous communication patterns"[7]. A comparison between the two has been made with conclusion being that for GPI to outperform MPI it needs to overlap the communication and computations appropriately [37]. Certain concepts within GPI are clearly aimed at this goal, such as notifications which attempt to reduce frequency of points in the code where many nodes wait for each other in favor of spreading them out and involving fewer nodes. Furthermore, the concepts of queues are used to separate different kinds of local requests by assigning them to queues of differing priorities. Their purpose is thus to allow for a more fine-grained control of the communication process, which is important when trying to overlap computations and communications. Hence it is clear that this overlapping is crucial for GPI and that it is designed in such a way as to help achieve this.

Figure 2.8: An example illustrating the different behavior of MPI's epochs and GPI's notifications.

```
if(node_id == 0){

        gaspi_read(..., node_1, ...);
        gaspi_notify(..., node_1, ...);
}
// Node 0 is now able to perform work without needing
// to wait for a respone from the read
```

```
MPI_Win_fence( ..., window_1, ... )

if(node_id == 0){

        MPI_Get(..., node_1, ...);
}


MPI_Win_fence( ..., window_1, ... )

// Any work done by node 0 here has to wait for
// a remote respone from get
```

[7]GASPI further reading: `http://www.gaspi.de/faq/`

2.13 StarPU

StarPU is a high-level runtime system meant to provide a unifying execution model for heterogeneous systems, including both CPUs and GPUs [3]. The goal of StarPU is to create an API which handles mapping and scheduling, requiring only the programmer to create tasks for the runtime system to schedule[8]. As such it abstracts away from the hardware and the additional complexities of properly utilizing a single heterogeneous node. However it has been expanded to handle clusters of nodes using MPI combined with its previous runtime system [3]. This newer cluster version is called StarPU-MPI but in this thesis it is simply referenced as StarPU as it is the only version discussed in any detail. StarPU handles both the communication between nodes and also its internal parallelization unlike GPI and how MPI is frequently used [30]. From this it is clear that StarPU works on a higher abstraction level than both of them and as such carries more responsibilities than either of them. Lastly the runtime system of StarPU schedules the tasks itself and this is done by a scheduler which dynamically divides them among all nodes. To schedule tasks in this way is another example of dynamic scheduling which is the scheduling model of choice for StarPU.

2.14 Problem types used by the benchmark programs

This section outlines some problems from different disciplines which form the basis of this projects benchmark programs.

The n-body problem

The n-body problem is a well studied problem in physics[9] and pertains to how n bodies move under the effect of gravity or electrostatics given a certain mass, initial velocity and charge. The acceleration of every body is dependent on its distance to all the others, and as they move the acceleration changes. While the problem is easy to formulate solving it has proven to be difficult as an analytical solution is impossible for values of n larger than two [22]. The favored approach is instead a computer simulation, a naive implementation of which would grow in $O(N^2)$ as every pair of bodies needs to be evaluated. A parallel implementation would face the issue of intense communication due to the strong data dependencies where every body depends on every other. Despite this much research has been done leading to algorithms which grow in $O(N \log(N)$ [9, 4, 43]. More work has been done with these algorithms when trying to combine them with GPUs as they became more common place [25, 20]. The n-body problem lies in the intersection of HPC and physics and is a well established problem in both domains.

The Mandelbrot set

The Mandelbrot set is a set of complex numbers named after the mathematician Benoit Mandelbrot who was among the first to study the phenomenon and later on wrote a paper about the subject in 1980 [34]. The set is defined as the complex numbers which when repeatedly applied with a specific function converge, and the complement being those number that diverge given the same function. Given an input complex number c it is a part of the Mandelbrot set if the absolute values of the following remains bounded for all n:

$$z_1 = 0$$

$$z_{n+1} = z_n^2 + c$$

When determining whether a number c is part of the set or not it is difficult to tell whether a value will converge or not. If an iterative computational model is used to evaluate different

[8]User guide to StarPU: `https://files.inria.fr/starpu/doc/starpu.pdf`

[9]Additional reading on the n-body problem is available at: `https://www.britannica.com/science/celestial-mechanics-physics/The-n-body-problem`

c values the amount of iterations needed to reach this conclusion may vary greatly. In Table 2.1 this is shown as the c values of $0+0i$ and $1+0i$ are very easy to make a decision about whereas $0.25+0i$ may require a few iterations and $-1.1+0i$ is even more unclear as it seems to oscillate between similar but not equal values. When determining whether a value belongs to the set or not the numerical approach must have a stop criteria and as such may make an incorrect classification. Oscillating values such as $c=-1.1+0i$ typically need to be evaluated for as many iterations as is allowed by the program since it is not clear if the oscillation will ever stop. The result is that the computational complexity to decide whether a given c value exists within the set has a great variance. There are many different ways of visualizing the Mandelbrot set, there is the binary way shown in Figure 2.9. Other ways include, but are not limited to, coloring the pictures in accordance to how many iterations where needed to determine if the pixel would converge or not.

c	$0+0i$	$1+0i$	$-1+0i$	$-1.1+0i$	$0.25+0i$
z_0	0	1	-1	-1.1	0.25
z_1	0	2	0	0.11	0.31
z_2	0	5	-1	-1.09	0.35
z_3	0	26	0	0.08	0.37
z_4	0	677	-1	-1.09	0.39
z_5	0	$4.6\cdot10^5$	0	0.09	0.40

Table 2.1: Example values produced by the Mandelbrot function given differing c values. Note that all the presented c values have zero as their imaginary part, this is done to make the table easier to read and not a feature of the set, which does contain values with a non-zero imaginary part.

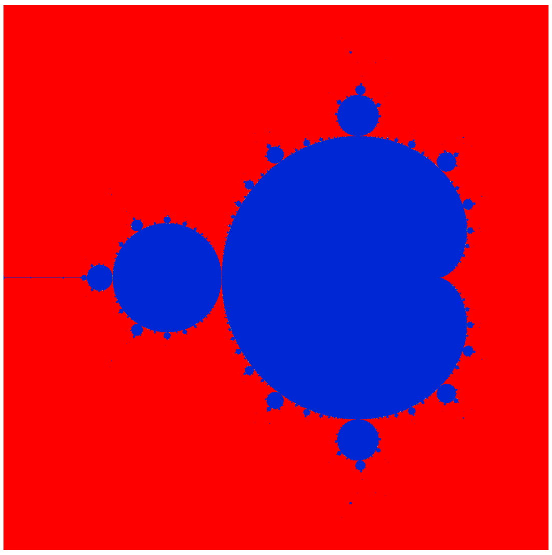


Figure 2.9: The Mandelbrot set where the blue pixels corresponds to points within the set. The image was generated with the SkePU GPI backend and uses a maximal iteration count of 10^4.

Taylor series

A Taylor series is the sum of an infinite polynomial approximation of a function at a certain point [19]. The series is a good approximation of the function at the point and near it, but the quality of the fit worsens as the distance from the point increases. The approximation achieves this by matching the first N derivatives of the original function at the chosen point. As the series is infinite N is made to approaches infinity, a larger N thus corresponds to higher degrees of derivatives being considered and hence a better fit.

As the Taylor series is infinite given an infinitely derivable function solving it numerically will never be perfectly exact. Research has been done in order to generate long series with high precision through parallel multi-node execution by Hristov et al.[27]. Precision is of particular importance of Taylor series as subsequent higher rank derivatives will have less impact on the approximation. In Hristov et al.'s experiment they achieved a precision of about 3374 decimal digits using MPI and OpenMP.

Matrix and vector multiplications

Matrices and vectors are a foundational objects in linear algebra and so are their basic manipulation techniques such as addition and multiplication. The uses of linear algebra computations are innumerable as they so often are a subtask of a larger problem, some examples include solving linear least-squares problems as well as linear equation systems. These smaller problems may themselves use basic functionality such as matrix multiplication and addition multiple times while only being subtasks of a larger one. Thus these basic operations may be called a very large number of times, this was noticed already in 1979 with the creation of the Basic Linear Algebra Subprograms (BLAS) library [31]. BLAS contains highly optimized implementations of common linear algebraic operations with the computational complexity of the operation corresponding to its BLAS rank. Using this Dongarra, Bunch, Moler and Stewart created LINPACK [13] in 1979 as a benchmarking suite where the problem to be solved is a linear equation. LINPACK has since then been updated and expanded to benchmark HPC clusters [12] and is the benchmark of choice for Top 500 [12]. The problem used to benchmark the clusters is still within the domain of linear algebra and still uses the low level operations along the lines of a matrix matrix multiplication. Thus these kinds of operations are of great interest within the HPC community and a corner-stone of a prolific benchmarking suite within the field.

Given a matrix matrix implementation $C = A \times B$ where B is of dimensions $M \times N$ and C of $N \times P$. Then all $M \cdot P$ elements within C needs a row of elements from A and a column of elements from B, corresponding to $2N$ number of elements. As such the computational complexity becomes $MP \cdot 2N = 2NMP$, if we then let $N = M = P$ we get that the computationally complexity for a squared matrix matrix multiplication is $O(N^3)$. The communication complexity however is only on the order of $O(N^2)$ as the transfer of both matrices would require $2 \cdot N^2$ operations.

In matrix vector multiplication given the destination vector D of length N, the square matrix E of dimensions $N \times N$ and argument vector F of length N. Then every element of D corresponds to a single element from vector F and a entire row from E. Thus the computationally complexity becomes $N \cdot (1 + N)$ which is $O(N^2)$. The communication complexity is at most the size of the matrix E and vector D, which corresponds $N + N^2$ number of elements and is $O(N^2)$. Interestingly here we see that the computational and communicational complexity is the same in matrix vector multiplication when using a square-matrix, unlike the previous example of matrix matrix multiplication. This fact has noteworthy consequences for parallelization of the two problems as the matrix matrix multiplication is computationally bound and more likely to benefit from increased computational power at the cost of communication. As matrix vector multiplication is equally bound by communication and computations a parallel solution with an increased communication cost runs a larger risk of creating a new bottleneck.

These two examples demonstrates the different properties of the operations using a naive implementation, but as these operations are of crucial importance there exists many different algorithms which outperform the naive ones. An example of non-naive implementation is the Strassen algorithm which can perform a matrix matrix multiplication with $O(N^{log_2(7)})$, where $\log_2(7) \approx 2.8$ [39]. Another approach is the systolic one first presented by Kung in 1982 [29] and followed up by Wan and Evans presenting 19 systolic approaches for matrix matrix multiplication [42]. Clearly these operations are of crucial importance to the field of computation as they are continuously being researched with there existing many different implementations of them.

2.15 Related work

Work similar to this thesis has already been done, for example the Muesli framework provides parallel algorithmic skeletons for clusters using C++ [7]. The goal of this framework is very similar to the GPI backend's and hence the differences lie primarily with how they are used and implemented. For example the syntax between the two is fairly different with Muesli using its skeletons as a field within a container unlike SkePU where they are their own object. Perhaps the most notable difference in this category is the fact that Muesli is only able to use at most two containers in an algorithmic skeleton unlike SkePU which has no such limit at all [7]. Furthermore the map skeleton provided by Muesli is not able to access other elements, limiting the problems it can handle. However, unlike the GPI backend Muesli is able to make use of GPUs and is hence better adapted towards heterogeneous clusters. Lastly the communication of Muesli is done with MPI whereas an important aspect of this thesis is the prototype's GPI usage [7]. Hence it should be clear that while Muesli and this thesis have a very similar goal from a high level perspective, they are different in both implementation and use-case.

Another related framework is JaSkel, which is Java-based and offers algorithmic skeletons able to execute on a cluster [18]. Unlike SkePU this framework works through polymorphism and all domain-specific data is provided by creating a subclass and implementing its abstract methods. A focus point for JaSkel is chaining different skeletons together by combining them into for example a pipeline and then executing multiple of these chains in parallel [38]. How to combine the different steps in the chain needs to be defined by the user. Thus JaSkel focuses quite heavily on task parallelization whereas the GPI backend handles tasks sequentially but with drift.

DatTel is a framework similar to SkePU which aims to provide data-parallel skeletons, but it does this by attempting to extend the C++ standard library (STL) rather than creating its own interface [5]. The STL already includes many functions which are similar to algorithmic skeletons such as: ”for_every” ≈ map, and ”accumulate” ≈ reduce. As such extending it with algorithmic skeletons would fit its stylistic choice. Much like the GPI backend DatTel assumes a single control flow which enables it to weave in computations outside of the framework more easily. Similar to SkePU it favors portability by letting the same code be run on different hardware in either a single node fashion or by using a cluster. Unlike SkePU however it only offers a single backend for its non-cluster execution which is pThread based. Also it diverges further from this thesis through its cluster communications which are MPI based.

3 SkePU

This chapter is a complement to the background chapter and explains the purpose and usage of SkePU as well as its features and syntax. The focus is on aspects of SkePU which are relevant for the thesis hence some features of SkePU are omitted in this chapter. For an explanation of such aspects and a more thorough discussion around SkePU in general please see Ernstsson's licentiate thesis [16].

3.1 Purpose and usage

SkePU is a C++ template library for parallel execution originally created by Enmyren and Kessler [14] and later extended to SkePU 3 by Ernstsson, Ahlqvist, Zouzoula and Kessler [17]. SkePU offers parallel algorithmic skeletons which makes it easier and less error-prone for a user to parallelize his or her code. SkePU has multiple skeletons which in turn may be executed using multiple different parallel tools such as OpenCL, OpenMP and CUDA among others. These are known as *backends* and by adding flags during the compilation to signal which backend is desired the user may choose dynamically which one to use. This allows for the same program to be executed on different hardware without any changes in the code. For example when running a program on a machine with a GPU selecting the CUDA backend for execution might be appropriate, whereas on a machine without a GPU, the OpenMP backend would be a better fit. The compilation of a SkePU program is done in two phases, a pre-compilation phase followed by a regular compilation. This pre-compilation is done through a tool provided by SkePU and generates appropriate code for the selected backend as not all of them use C++. The second compilation step is done through the appropriate compiler of the chosen backend. By adding multiple backend flags during the compilation steps it is possible to enable multiple backends for the program, which lets it choose which one to use dynamically at runtime. SkePU was at first only available on shared memory, single node systems, but it has been expanded to allow for multi-node execution using StarPU.

The syntax of SkePU is the same regardless of the backend and it primarily uses two constructs, the skeleton objects and the data-container objects. The skeletons takes in typically one user-function at creation and are executed at later on with container objects as their argument. The type of the skeleton determines which computational pattern it uses whereas arguments in its creation and execution may alter its behavior somewhat. A trivial example of how the SkePU syntax looks is illustrated in Figure 3.1. This sequential syntax is an important part of SkePU as it abstracts away from the parallel implementation and allows the user to focus on the problem rather than the implementation of it. By using only SkePU constructs it is possible to write a parallel program without needing to deal with any of the difficulties of parallel programming.

Figure 3.1: A trivial example of how skeletons are used in SkePU.

```
int i, j;
skepu::Matrix<int> m{i, j};

int add_f(int a, int b){
        return a + b;
}

auto add = skepu::Map(add_f);

// Double every value in m and store it in m
add(m, m, m)
```

3.2 Features and algorithmic skeletons

SkePU offers to the programmer a set of parallel algorithmic skeletons. Furthermore utilities such as different kinds of containers exist. The skeletons which exist in SkePU are: Map, Reduce, MapReduce, Scan, MapOverlap, MapPairs and MapPairsReduce. They all work differently, but shared among them is that their inputs are SkePU containers or constants. Their output is given by either modifying one of the argument containers or by returning a scalar or some other object type. SkePU containers come in different dimensionality for different tasks: Vector, Matrix and Tensor. This section is a short introduction to all of SkePU's skeletons, a more comprehensive layout of the syntax has been written by Ernstsson [16].

Map

Map is one of the most flexible skeletons in SkePU and it is based on the well established computational pattern of the same name, but is extended beyond it. It takes in an arbitrary amount of argument-containers and typically a single destination container. In this case let C be a container where its superscript denotes its name and the subscript its index, we can write the map pattern as:

$$C_i^{dest} = f(C_i^1, C_i^2, ...C_i^N)$$

This is how the standard map pattern works, but SkePU allows for a more general execution. Firstly the index is provided to the user-function if its signature indicates that this is desired. Furthermore an arbitrary amount of scalars are also valid arguments as long as they are provided after the containers. If we let S denote a scalar, the execution model then becomes:

$$C_i^{dest} = f(i, C_i^1, C_i^2, ...C_i^N, S^1, S^2, ..., S^M)$$

In this classical Map accessing pattern we are prevented from accessing an element of a different index. In SkePU's skeleton however we may do so by using a random access *proxy container*, yet again extending the previous mathematical description. As it is possible to access any element from any index the pattern would look like the following:

$$C_i^{dest} = f(i, C^1, C^2, ...C^N, S^1, S^2, ..., S^M)$$

In order to differentiate how the arguments are interpreted SkePU analyzes the signature of the user-function. In the Figure 3.2 the function "mult_f" demonstrates the different accessing patterns as *c* is using the random access one whereas *d* is not. The first argument "index" may be omitted if this information is not needed. These utilities mean that SkePU's map is able to handle a wider range of problems than the original computational pattern. Lastly SkePU's map is capable of an arbitrary amount of return values of possibly different types. To achieve this multiple destination containers are required and the signature of the user-function need to match this. This thus allows the same map to calculate multiple values without needing to iterate through a container multiple times.

The random access pattern within map is itself divided into multiple patterns: Vec, Mat, MatCol and MatRow. These indicate how the random elements will be accessed and fulfills the dual purpose of both making it easier for the user and letting SkePU optimize the data access. Vec is used to access an element with a single index and Mat allows for two-dimensional indexing. MatCol and MatRow however lets the user-function access only within a whole row and or column, which are important in certain tasks such as matrix matrix multiplication. Importantly a SkePU backend is able to optimize the memory-access speed for MatCol and MatRow as it can expect further accesses within the same column or row. Contiguous memory access is much faster and both columns and rows may not be contiguous in the memory simultaneously, as such optimizations such as MatCol and MatRow may significantly improve performance. To conclude, SkePU's Map skeleton is more extensive than the traditional map pattern it is based on and is thus able to handle many more problem types.

Figure 3.2: An example of SkePU's Map's different access patterns.

```
int i, j;
skepu::Matrix<int> m1{i,j};
skepu::Matrix<int> m2{i,j};

int add_f(int a, int b){
        return a + b;
}

int mult_f(skepu::Index index, skepu::Vec<int> c, int d){
        return c[index.i] * d;
}


auto add = skepu::Map(add_f);
auto mult = skepu::Map(mult_f);

// store the result in m1
add(m1, m1, m2)
mult(m1, m1, m2)
```

Reduce

Reduce is another well established computational pattern which works by accumulating all the elements in a container with a binary and associative user-function. Unlike SkePU's Map, its Reduce is very similar to the computational pattern, which is described in Section 2.8. Figure 3.3 illustrates the fairly straightforward syntax of a one-dimensional reduction. An extension of the pattern exists in the form of two-dimensional reductions which work by applying the user-function to all elements in a row or column and generating a vector. There are two ways to handle these types of reductions, either the output is a vector or a second user-functions may be provided which is then applied on the generated vector to create a scalar output. Figure 3.4 illustrates the syntax of the latter approach.

Figure 3.3: Example demonstrating the syntax of a one dimensional reduction in SkePU.

```
int i, j;
skepu::Matrix<int> m{i, j};

int add_f(int a, int b){
        return a + b;
}


auto add = skepu::Reduce(add_f);
int res = add(m);
```

Figure 3.4: Example demonstrating the syntax of a 2D reduction in SkePU by Ernstsson [16].

```
float max_f(float a, float b){
        return a > b ? a : b;
}

float max_row_sum(skepu::Matrix<float> &v){
        auto max_sum = skepu::Reduce(plus_f , max_f);
        max_sum.setReduceMode(skepu::ReduceMode::RowWise);
        return max_sum(v);
}
```

MapReduce

Just like the MapReduce pattern is a combination of map and reduce SkePU's skeleton of the same name combines the existing utilities of its map and reduce implementation. As such the intricacies of the Map skeleton with its random access and scalar handling is also present in MapReduce. The signatures of the user-functions supplied to the map and reduction part of MapReduce shares the same meaning as in their own skeletons implementation. One difference between MapReduce and its component skeletons however is that only one-dimensional reductions are allowed. Figure 3.5 illustrates how the syntax of SkePU's MapReduce looks. As discussed more in Section 2.8 the benefit of MapReduce is that combining the two functions reduces execution time and memory usage while reducing the amount of skeletons the user needs to create.

Scan

SkePU's scan is an implementation of the computational pattern with the same name which is described in Section 2.8. The syntax is very similar to the other skeletons, an example of this can be seen in Figure 3.6.

Figure 3.5: Example demonstrating the syntax of SkePU's MapReduce

```
        int i, j;
        skepu::Vector<int> v1{i,j};
        skepu::Vector<int> v2{i,j};

        int add_f(int a, int b){
                return a + b;
        }

        int mult_f(int a, int b){
                return a * b;
        }

        auto dot_product = skepu::MapReduce(mult_f, add_f);
        int res = dot_product(v1, v2);
```

Figure 3.6: Example demonstrating the syntax of SkePU's Scan by Ernstsson[16].

```
float max_f(float a, float b) {
        return (a > b) ? a : b;
}


skepu::Vector<float> partial_max(skepu::Vector<float> &v) {
        auto premax = skepu::Scan(max_f);
        skepu::Vector<float> result(v.size());
        return premax(result, v);
}
```

MapOverlap

MapOverlap is based on a very broad pattern which goes under many different names depending on discipline, such as convolution, stencil, filter and window function [16]. In this thesis the computational pattern is denoted as stencil computations and is described in more detail in Section 2.8. The functionality of MapOverlap is similar to Map in that it applies a user-function at every index of a container, but it follows a very different accessing pattern. While Map allows for multiple containers, MapOverlap does not. SkePU's Map does allow for random access whereas MapOverlap only allows for accesses to elements which are within the same neighborhood. As a stencil operation MapOverlap must define a range and declare that all indexes within this range are neighbors, these are the only elements the user-function may access. This trades off the user-friendliness of the random access for the ability to handle tasks which have too strict data dependencies to be used by Map. An example of the syntax is seen in Figure 3.7.

MapPairs

MapPairs follows the cartesian product style pattern presented in Section 2.8. It takes in two distinct sets of sets of vectors. The size of all vectors within the same set must be the same, and the skeleton's output is a matrix. Every index-pair of the two sets is combined and used as an argument in the user-function to create an element in the output matrix. Let us denote two vector-sets as V_1 and V_2 with the lengths N and M and their vectors as $v_{11}, v_{12}, ..., v_{1N} \in V_1$ and $v_{21}, v_{22}, ..., v_{2M} \in V_2$. Furthermore let the superscript of the vectors denote which index

Figure 3.7: Example demonstrating the syntax of SkePU's MapOverlap by Ernstsson[16].

```
float conv(
        skepu::Region2D<float> r,
        const skepu::Mat<float> stencil
        )
{
        float res = 0;
        for(int i = -r.oi; i <= r.oi; ++i)
          for(int j = -r.oj; j <= r.oj; ++j)
                res += r(i, j) * stencil(i + r.oi , j + r.oj);
        return res;
}

skepu::Vector<float> convolution(skepu::Vector<float> &v){
        auto convol = skepu::MapOverlap(conv);
        Vector<float> stencil {1, 2, 4, 2, 1};
        Vector<float> result(v.size());
        convol.setOverlap (2);
        return convol(result, v, stencil, 10);
}
```

is referenced and let i and j be indexes. Lastly let Mat be the output matrix and f the user-function, then we can express it as:

$$Mat_{ij} = f(v^i_{11}, v^i_{12}, ..., v^i_{1N}, v^j_{21}, v^j_{22}, ..., v^j_{2M})$$

The dimensions of Mat is thus $N \times M$, meaning that more vectors in the vector-sets do not affect its dimensionality. Instead they provide additional arguments to the user-function. Furthermore, like Map, MapPairs allows for a multi-type return value which is then put into multiple destination matrices. An example of MapPairs' syntax is given in Figure 3.8.

Figure 3.8: Example demonstrating the syntax of SkePU's MapPairs provided by Ernstsson[16].

```
int mult_f(int a, int b){
        return a * b;
}

void cartesian(size_t Vsize, size_t Hsize){
        auto pairs = skepu::MapPairs(mult_f);
        skepu::Vector<int> v1(Vsize, 3), h1(Hsize , 7);
        skepu::Matrix<int> res(Vsize, Hsize );
        pairs(res, v1 , h1);
}
```

MapPairsReduce

MapPairsReduce is analogous to MapReduce as it is functionally the same as executing a MapPairs followed by a Reduce. Like in MapReduce combining these operations together makes it is possible to compute them more efficiently while using less memory. The reason for this is that the matrix created by MapPairs can be instantly consumed by the reduce-function.

MapPairsReduce may only perform a 2D reduction, either row-wise or column-wise. As such the output of MapPairsReduce will always be a vector.

Figure 3.9: Example demonstrating the syntax of SkePU's MapPairReduce by Ernstsson [16].

```
int mult_f(int a, int b){
        return a * b;
}

int add(int a, int b){
        return a + b;
}

void mappairsreduce(size_t Vsize , size_t Hsize){
        auto mpr = skepu::MapPairsReduce(mul, sum);
        skepu::Vector<int> v1(Vsize), h1(Hsize);
        skepu::Vector<int> res(Hsize);
        mpr.setReduceMode(skepu::ReduceMode::ColWise);
        mpr(res, v1, h1);
}
```

4 Design and implementation

This chapter goes into depth about the prototype's design and its implementation. Concepts integral to how the prototype operates, such as its consistency model and communication pattern among others, are explained here. As the design is also mentioned so are its consequences, limitations and alternatives in order to provide a wider view of the issues it attempts to handle. Lastly an explanation of how the benchmark programs were implemented and what they aim to measure is provided at the end of this chapter.

4.1 Prototype feature delimitations

SkePU is a large project and as such our prototype only implements a subset of the features of other backends. The chosen ones are three algorithmic skeletons: Map, Reduce and MapReduce, combined with the two container types Matrix and Vector. These were selected as they are among the most commonly used types within SkePU and together they suffice to solve a large number of problems. Furthermore using SkePU's extensive Map pattern it is possible to emulate some other patterns at worse performance. For example Map may perform a cartesian product like MapPairs does by accessing any needed element with the random access proxy container. As such the chosen classes are able to solve enough problems to be able to be compared against the existing StarPU implementation without being limited by problem variety.

However even within the implemented classes not all of their features are included in the prototype. They are excluded as the prototype is able to handle many different problems without them, and as such is able to be measured against the existing StarPU backend without them. By simply not using these features in the benchmark programs a valid relative comparison can still be made, although it does not necessarily represent the backends' performance in absolute terms. It is worth noting thus that the performance of the benchmark problems are not the same as the backend's performance on the underlying problem as certain features are not used. This does not however affect the relative performance comparisons which is the central point of the thesis. The following features are not implemented in the GPI backend: multi-variable returns, proxy container patterns other than the one and two-dimensional ones, Vec and Mat. Notably this excludes MatCol and MatRow, which are important optimizations for any program which accesses entire rows and columns. Lastly, any reduction which is done row-wise or column-wise is not supported by the prototype, only reductions into a single value is. For more details regarding these excluded features as well as the motivation for why they were added to other backends, please see the paper by Ernstsson et al. [17].

4.2 Design

This section highlights the overarching design of the GPI backend and how it operates on a high level.

Usage and program structure

While an important part of a backend is that it implements the SkePU interface, its execution model need not be the same as other backends. This is a natural effect of SkePU's wide applicability, executing a binary file on a single machine is a different process than doing so on a cluster. In line with this variance the GPI backend takes a slightly different approach than the other backends and does not use the pre-compilation step. The main reason for the pre-compilation is to transform the code from C++ to a backend-language which the backend then may compile itself. This is crucial for backends which do not use C++ code internally such as OpenCL. However as the GPI prototype does do this the pre-compilation is not needed. Furthermore as the prototype is a proof of concept adding compatibility features does not help it fulfill its stated purpose. Hence the decision to exclude the pre-compilation step was taken. Instead a SkePU program meant to use the GPI backend is compiled directly with the GPI compiler, and later executed with GPI's execution tool.

As stated above SkePU is an interface to be implemented by the backends, and as such the new GPI backend is essentially a stand-alone library written from scratch. It is used by including the SkePU the library and setting the compilation flags accordingly. Notably there are no dependencies between the GPI backend and any existing SkePU code even if they share the same interface.

The execution model of the GPI backend is SPMD. This means that the code written by the user is executed on all of the nodes separately with only the SkePU constructs performing calls to remote nodes. Thus any work not done using SkePU is run in parallel as many times as there are nodes, hence it should constitute only a small part of the overall work of the program for performance reasons. The SkePU constructs use the arguments provided to them combined with their node ID to divide their workload accordingly. Depending on the construct it may create an OpenMP parallel region to handle the task using the multiple cores it may have available. This program structure assumes homogeneity among the nodes and would lead to load imbalances if some nodes were using better or worse hardware as they would traverse the sequential, and their share of the parallel, parts faster or slower. Which is an issue as a program is only finished after its slowest node is finished. Furthermore the program structure leads to a sort of determinism which is leveraged to help deduce a node's workload asynchronously. If the user does not access the node IDs or uses if-statements with arguments derived from the hardware as shown in Figure 4.1, the control flow of the program is the same for all nodes. Hence, using only its ID and the total node count, a node is able to deduce a large amount of information about a SkePU call since it assumes that every other node will perform the same SkePU call with the exact same arguments at some point in time. Allowing for deduction about the other nodes' states is important as it may be done without any communications, which in turn are slow.

State tracking with operation numbers

Due to the program structure explained above it is possible to create a global order of every SkePU call for all the nodes. These calls are then divided into phases, and every phase is given an incrementally increasing and unique operation number. For example Map is divided into a waiting stage, an execution stage and a finished stage which would correspond to operation

Figure 4.1: An example of syntax which the GPI prototype is unable to handle. The reason is that the value within the if-statement depends on the node's hardware's state and the GPI backend assumes an identical control flow for all the nodes. In other words every creation of a container, skeleton object or execution of a skeleton is assumed to be done by every node, in the same global order.

```
int main(){
        int i, j;
        skepu::Matrix<int> m{i,j};

        int add_f(int a, int b){
                return a + b;
        }
        auto add = skepu::Map(add_f);

        // If the current time is even
        if(std::chrono::system_clock::now() % 2 == 0){
                add(m, m, m);
        }
}
```

numbers: N, $N+1$ and $N+2$. By using these operation numbers, a node is thus able to deduce the state another node is in currently and which operations it has not yet executed. An important caveat is that a node can only deduce which operations another node has not executed if it itself has executed more operation than the other one. In other words, it knows which operations a node that is lagging behind will do, but it knows nothing about what a node that has drifted ahead has done. Furthermore the operation numbers are bound to the nodes themselves, whereas most other parts of the synchronization process are bound to a container object. As such the operation numbers are used to order the other aspects of the synchronization process, which are primarily based around the containers.

The method for propagating the nodes' different operation numbers is based on a synchronization schema known as *vector clocks* [10]. In this schema every node, processor or thread depending on how it is used, has their own counter which is incremented whenever an "event" occurs. An event corresponds to either an independent calculation or a communication event. Furthermore every node stores a vector of the highest occurred value for all the existing nodes, including itself. Whenever it then receives a communication event it updates any values within its vector to the incoming one if they are larger. This thus results in a system where it is possible to clearly see which events occurred before any given state. An illustration of the vector clock schema is shown in Figure 4.2. There are different variations of the vector clock schema depending on its usage, for example in a shared memory systems the size of the vector could correspond to the number of shared objects while its elements are thread IDs [44]. Furthermore there exist more complex vector clock algorithms with better performance than the ones presented so far [44].

The GPI backend's synchronization model is heavily based on the vector clock schema with the crucial difference that every node has the same local events. This does not mean that every node necessarily performs the same computations but rather that every node performs the same SkePU operations and hence passes through the same phases of these operations. What a node does in these phases may vary, for example in the "get" function within Matrix some nodes must fetch the value remotely whereas others need not. As such the values within the prototype's vector clock correspond to certain phases of a SkePU operation. By leveraging this, a node is able to deduce information regarding which transformations have been applied to a container and which have not. Another benefit of the vector clock model is that information may propagate without direct contact, which is particularly beneficial at points which require tight synchronization.

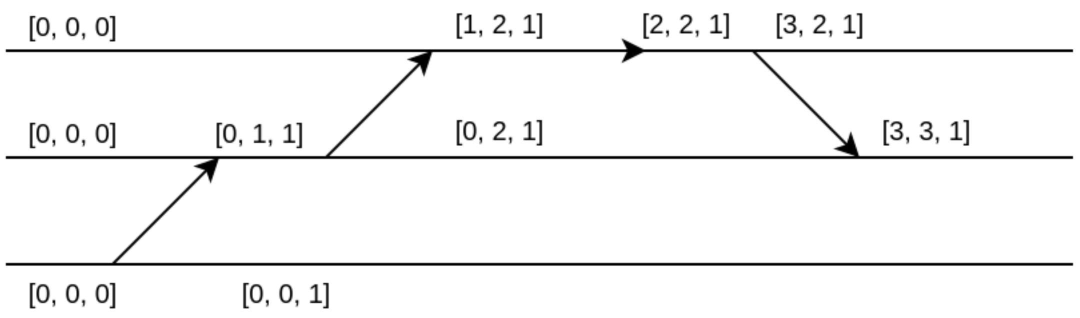


Figure 4.2: An example of how the vector clock schema could be implemented. The three lines are processors and every arrow is an event.

Constraints

The synchronization process is based on operation numbers and constraints. A constraint is a tuple containing a node ID and an operation number. At the end of an operation a node adds constraints to indicate which nodes might read from it during this operation. Then when the container wants to modify its content it has to fulfill all of its constraints before it is safe to modify it. A constraint can be said to be fulfilled when the remote node has reached an operation number larger than the constraint. Due to the SPMD nature a node is able to deduce how the access pattern of an operation is going to look for all the other nodes and as such set the constraints properly. The primary issue however comes in the form of the random access pattern available in Map. That is when elements from other indexes than just the current one is used. Here the user-function determines which nodes communicate with each other. As a constraint indicates that a remote node will read from the current node; this entails that the current node must be able to deduce which nodes a remote node will read from. According to the famously unsolvable Halting problem this can only be done by actually executing the user-function with the same arguments as the remote node [41]. This is clearly an unfeasible amount of work; just the argument transfer would result in a transfer of size $(\#global\_elements - \#local\_elements) \cdot type\_size$ bytes, for every node. As constraints and the loose memory consistency attempts to limit the communication such a cost is unacceptable. Instead every node is assumed to read from every other node if the container is used with a random access pattern. This is a pessimistic assumption only used to avoid the issues mentioned above. If the user-function only uses the current indexed element in the container, then looser constraints can be used. In this case a node can deduce which elements, and hence which remote nodes, all the nodes will access. Which in turn results in constraints which more properly describes the data dependencies.

The constraints are set at the end of an operation and waited for at the start of it, ensuring that data which a remote node might want is never replaced. This scheme guarantees that a node which has drifted ahead does not replace needed data, but it does not guarantee that the data has been produced yet. To complement this, all nodes must validate that the remote node has reached the correct operation number before reading from them. This process is explained in more detail further down.

Consistency model and double buffer

The ultimate purpose of the constraints and operation numbers discussed above is to allow for a weak memory consistency. Every container has a double buffer of its data which other nodes are unable to read from. Whenever an operation modifies the container it always writes the changes to the double buffer. To allow remote nodes to access this new data a container must periodically flush its double buffer and put it into the actual GASPI segment. This is done as sparingly and late as possible by letting nodes deduce whether a remote node might need to read from them during an operation, and only then perform the flush. For example a simple *get(i)* operation will make the node which holds i flush its changes but not any other node. In order to track the changes every node saves the remote state of every other node through the following two fields within the Matrix class: the last operation which modified it and the last operation when it flushed its changes. With these fields every part of the container is able to deduce at which operation number a remote node is safe to read from and when it is not. It also adds fine grained synchronization where parts of a container may be flushed while others are not. Thus the operation number where the flushed data is accessible may differ depending on which node is being read. For example, suppose that node i changes one of its values at operation number j and it got flushed at operation $j + 1$. If another node wants to read from node i it now has to wait for node i to reach operation number $j + 1$. However for any other remote node the required operation number to reach before reading may be lower than $j + 1$ as its value was not changed in operation j. The granularity of the scheme is thus at the level of a node, where all values within a container that belongs to a node are either dirty or not. Lastly note that the field tracking other node's last flush operation is the last time they performed a flush, not the last time they executed an operations where they flushed. To illustrate the difference if a node made local changes at operation i and flushes at $i + 1$ and $i + 2$ then the last flush field would say that this node flushed at $i + 1$.

Flushes

Flushes refer to the process of moving the data from the double buffer into the GASPI segment where it may be read by other nodes. They are done in two cases, firstly if the local node needs to modify its container which already is dirty. Then the existing changes are flushed into the GASPI segment and the new ones applied to the double buffer. The second case is if the local node deduces that another node needs to read data which exists in the double buffer, which is only accessible locally. This architecture thus allows for every node to save two states of its container and makes it the nodes' responsibility to guarantee that the correct state is available at certain operation numbers. The local nodes do however only have knowledge of the operations they have processed so far and as flushes only are done when deemed necessary; a waiting period may have to precede it. For example let operation i modify container A and the subsequent 1000 operations not do so, but operation $i + 1001$ requires reading from the container A. Before a node may read from another it must flush the changes from operation i, but this flush will not be done until $i + 1001$ even though it could have been done earlier. However in this case a node which has drifted ahead needs to wait for a slower one, which is quite acceptable as it is the slowest node that determines the execution time. This case would only lead to a slowdown if a node which is ahead needs to wait and later on becomes the slowest node. This is an improbable scenario and as such this flushing implementation was chosen.

Communication pattern

The benefit of the weak consistency model is that it allows for an infinite drift as long as there are no data dependencies. A local node only needs to wait for a remote one if it either has not produced and flushed the needed data, or if it needs to read a local value and is yet to do so. With the states of remote nodes being propagated indirectly through the vector clock and local deductions the expected communication is kept to a minimum. The communications between nodes fall into two categories, polling for information about the remote nodes state and reading remote elements from one of its containers. In particular, communication across different containers are never done as a container is responsible for all communications with its remote partitions. This makes the design easier as the GASPI segments within every container only need to be able to store remote data from other partitions. The dimensions of this segment is thus adapted at these specific communication patterns and nothing else.

4.3 Matrix

This section describes in detail how the Matrix class is implemented and its purpose. It is the largest class in the GPI backend with a wide set of responsibilities which leads it to interact with most other classes in some manner.

Overall

Matrix is a two-dimensional container following SkePU's Matrix interface. Its main responsibilities are tracking the state of its local and remote elements as well as providing an interface for accessing its remote elements. As such it works both as an abstraction layer for remote communication and as a state tracker. Furthermore it also works as an alias for the vector class by implementing a few extra functions such as a one-dimensional constructor. The motivation for the aliasing is that the differences between the two is minuscule implementation wise and this solution was the most straight forward one while also not replicating any code unnecessarily. If the GPI backend is to be extended beyond a proof of concept then the vector could be implemented as a separate class.

Partitioning and workload balance

Matrix uses a simple one-dimensional partitioning of the global elements where the node with the highest rank receives any left over elements. This one-dimensional partitioning of a two-dimensional container means that rows and columns may be split across multiple nodes as illustrated in Figures 4.4 and 4.5. Furthermore Figure 4.5 demonstrates another potentially problematic partitioning result as a small matrix is split such that no node owns an entire row. This case is however rare in practice and is the result of using more nodes than the problem will benefit from. Regarding the overlapping rows it is worth noting that there may at most be $\#nodes - 1$ of them in any given matrix and if it is sufficiently large they will constitute a low proportion of all rows. The reason why row splits are undesirable are due to it not being uncommon for a user-function to need an entire row for its calculations. In such a case it is both faster to fetch the row from a single node due to the high startup cost of remote reads, while also requiring less synchronization.

Another potential issue of this partitioning scheme is the element imbalance which may occur due to the last node receiving all elements which may not be evenly split. This extra load can be up to $\#nodes - 1$ elements, which may or may not be a significant amount of elements depending on the container's size. This issue is illustrated in Figure 4.4 and contrasted by Figure 4.3. It is worth noting that if $\#elements >> \#nodes$ then this imbalance will become very small as a proportion to the overall amount of elements. For example if we have N elements distributed among 3 nodes, then the last node may only receive at most two extra elements. If N then grows large these two extra elements will become proportionally insignificant.

Data transfers and caching

The Matrix class is important to the data transfers of the prototype as it manages the GASPI segments, which are the destination and origin of all read requests. The elements of the local partition of the Matrix are stored in a GASPI segment, but this segment also has extra memory allocated for the transfer of remote elements. The size of this segment is the size of the local

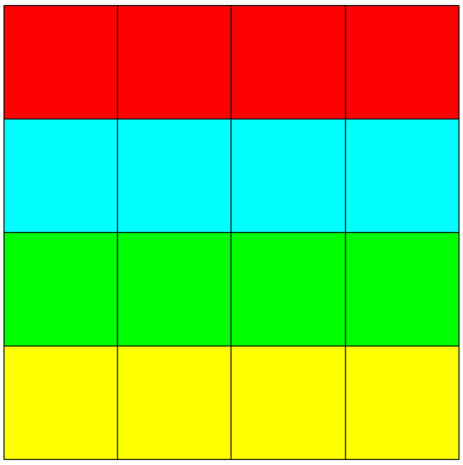


Figure 4.3: Partitioning of a matrix of size 4×4 with four nodes.

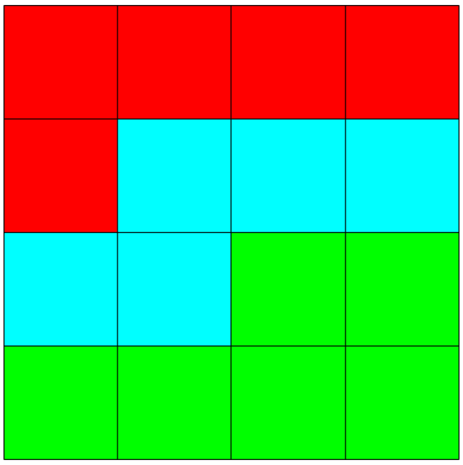


Figure 4.4: Partitioning of a matrix of size 4×4 with three nodes.

partition plus the global size of the container. Thus every remote element has its own unique position in the segment at the cost of high memory usage. While the local elements in Matrix also are stored in the same segment, logically the two parts are kept separate and referred to as the *container segment* and *communication buffer* respectively. While the container segment is only ever used for storing local values, the usages of the communication buffer are many and determined by the algorithmic skeleton.

One of Matrix most important functions is *proxy_get* which is called from the *proxy container* dummy class and allows for the access of any element, both local and remote. This class is used by certain skeletons, namely Map and MapReduce, and its purpose is to provide the user with a random accessing pattern of the elements. What *proxy_get* does is that it returns a value from the container given an index and it does so in a thread safe manner. Conceptually this function may either return the local or cached value instantly or it may need to transfer it. If this is the case then it also transfers all elements of the remote node and puts them in the communication buffer. Any further readings to this node will now read the cached data. By leveraging the state tracking fields mentioned in section 4.2 as well as tracking the state of the cache, the values within it may be reused for multiple operations. Thus the transfers are very large but the transfered elements may be reused for multiple operations through the cache, limiting the amount of transfers and hence their upfront overhead cost.

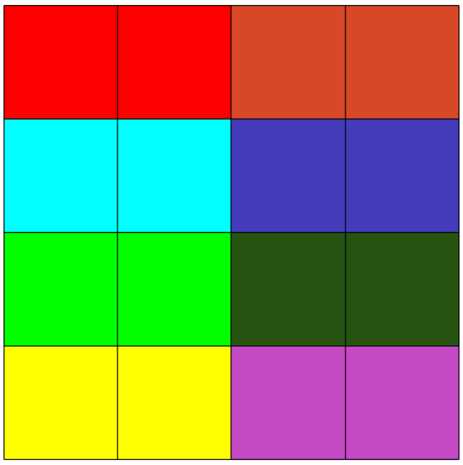


Figure 4.5: Partitioning of a matrix of size 4×4 with eight nodes.

Furthermore in *proxy_get* the main struggle is its synchronization complexity. For example different threads may want to read data from different containers of the same remote node. In such a case they need to synchronize their access to the local vector clock, as well as their access to the remote vector clock, in order to avoid overloading the remote node. To achieve correctness proxy_get uses three layers of locks. The first of which is contained in the Matrix class and is matched with a remote rank. These pairs thus ensure that only one thread is able to transfer data from the same remote rank's partition. Which is important as the data transfers always move entire partitions and the first-level locks thus remove the possibility of the same data being transfered twice. The second-level locks are shared among all local container-objects and guard the access to the remote vector clocks. They are paired up with a remote rank and thus every level-two lock corresponds to a remote rank. Before any local thread is able to poll a remote node's vector clock it first needs to gain access to the correct level-two lock. This makes it so only one thread is able to poll the same remote node. The last lock is the level-three lock, which unlike the other two levels is a single lock. A thread need to access it before it is allowed to modify the local vector clock. This lock is also shared among all local container objects within the same node. By putting these three levels together we see that level-one prevents multiple transfers of the same data; level two prevents unnecessary polling of a remote vector clock's state, and level three simply ensures correct usage of the local vector clock. A benefit of using this scheme is that it allows for multiple Matrix objects to read from the same remote node concurrently. Note however, that as Matrix objects only read from other partitions of the same objects; the concurrent reads occur when multiple objects want to transfer different data simultaneously. Lastly, this transfer design also allows for concurrent reads of different vector clocks through the level-two lock.

4.4 Map

Map is one of the larger classes in the GPI backend and the skeleton which is expected to be used the most. Some important issues which are brought up in this section is how remote elements are transfered through either *pre-fetch* or a *proxy container*. Furthermore the general structure of how the class operates is explained as well as some of its behavior which is marginally different from other backends.

Scheduling options

Every node is responsible for transforming the elements it owns, and the elements are in turn divided evenly among a set of threads within a single node. The scheduling is thus doubly static as both the inter- and intra-node work partitioning is done statically. The internal parallelization uses OpenMP which includes options for both dynamic and static scheduling, hence it may easily be changed. The assumption for static scheduling is that every iteration takes approximately the same amount of time. But, as the run time of the threads may include waiting for a remote node to reach a certain state, this assumption is dubious. The static scheduling was still chosen as it anecdotally resulted in a major speedup for Map when using a trivial user-provided function such as add. This is of course the sort of problem which static scheduling would excel at whereas the dynamic one would struggle. But the difference provided to be sufficiently large, the static one was up towards 100 times faster, so that the dynamic scheduler was deemed unable to handle such problems and instead the static scheduler was chosen. However a more thorough evaluation of the two scheduling models would be an interesting extension of the work and is needed in order to truly determine which is the better fit for the GPI backend.

Type deduction and divergence from the SkePU interface

The GPI backend's implementation of Map handles type deduction slightly differently than the other backends. First it compares the arguments given to the Map object when it is called, with the parameter type in the user-function. From this it can deduce how to handle every argument. Note that the same user-function may behave differently depending on the arguments given to a skeleton object as shown in Figure 4.6. By observing how the Map-objects are invoked we can see that the argument integer desired by the user-function may be supplied through either a container or as a stand-alone scalar. Deducing which case is requested is done during the Map-object's invocation unlike other SkePU backends which does this during the object's instantiation. One effect of doing the deduction at the call-site is that the GPI backend does not need the concept of arity unlike the other backends. In the other backends this concept is used to signal how many of the provided arguments are containers and how many of them are not. It is provided at the creation of the skeleton objects and often deduced automatically. But this means that there is less flexibility in reusing the same object as the arity is bound to it. For example in Figure 4.6 only one case in A and another one in B would be allowed in non-GPI backends as they can not handle different arities on the same object.

Furthermore the arity is either stated explicitly or deduced automatically, however some user-function signatures are ambiguous and hence the deduced arity may be incorrect. The functions provided in Figure 4.6 are examples of such ambiguous ones. Thus another benefit of avoiding the arity concept is the fact that the automatic type deduction of the GPI backend is unambiguously correct. But more than this it also allows for containers and scalars to be provided in an arbitrary order as long as they match the user-function's signature. This is different from the SkePU standard which states that scalars must come after any container arguments and the difference is highlighted in Figure 4.7.

Execution and argument generation

In Map the workload is statically partitioned in the same way as the destination Matrix container is. That is, every node is responsible for generating the result of all indexes which it owns in the destination container, following the so called ”owner computes rule”. The first step in this process is to build an argument tuple, which through type deduction is given four possible values: an index, a scalar constant, a scalar value from a container and a random access proxy container. The container-scalar values are fetched in a process called *pre-fetching* where all elements which the node is responsible for in the destination container, but does not currently possess, are fetched from remote nodes. This may occur due to the SkePU Map's syntax allowing for argument containers which are larger than the destination container and may hence be partitioned differently. All such elements are fetched remotely and put in their

Figure 4.6: A demonstration of how a user-function may behave differently depending on how its skeleton object is called. Note that the arity is different between skeleton calls within the same case, meaning that this example would only work in the GPI backend.

```
int i, j;
skepu::Matrix<int> m{i,j};

int mult_f1(skepu::Index1D i, skepu::Vec<int> a, int b){
        return a(i.i) * b;
}

int mult_f2(int a, int b){
        return a * b;
}

auto mult1 = skepu::Map(mult_f1);
auto mult2 = skepu::Map(mult_f2);

// Cases A
mult1(m, m, 2);
mult1(m, m, m);

// Cases B
mult2(m, m, m);
mult2(m, m, 4);
mult2(m, 4, 4);
```

Figure 4.7: An example of the difference between the GPI prototype's interface and the standard SkePU one. The improper version is acceptable in the GPI backend but not in the other backends, whereas they both accept the proper one.

```
int i, j;
skepu::Matrix<int> m{i,j};

int mult_proper(skepu::Index1D i, skepu::Vec<int> c, int d){
        return c(i.i) * d;
}

int mult_improper(skepu::Index i, int d, skepu::Vec<int> c){
        return c(i.i) * d;
}

auto mult_proper = skepu::Map(mult_proper);
auto mult_improper = skepu::Map(mult_improper);

mult_proper(m, m, 2)
mult_improper(m, 2, m)
```

corresponding communication buffer, where they are later fed into the argument tuples. This process thus knows beforehand exactly which elements it needs and thus fetches them in as few reads as possible. It is worth noting that Matrix objects only communicate with other objects of the same global container structure, which means that the transfers are handled by the argument containers and not the destination one. As it currently stands the entire pre-fetch chain of events is single threaded, the internal parallelization through threads only starts after the pre-fetch is done.

Another way to access remote elements is with the random access proxy container, which is essentially a dummy object holding a pointer to the container object which it belongs to. By using this object in the user-function it allows for access to any value within the container. Due to the light-weight nature of this object, creating and adding it to the argument tuple is a computationally cheap operation. The remaining two argument types are fairly simple concepts, the index is a type containing the current execution index and the constants-scalar are values which are the same for all indexes. The tuple itself corresponds to the arguments given to every execution of the user-function and is primarily needed due to variadic template programming. The parallel execution begins after all the pre-fetching is done and it divides the local indexes evenly among the existing threads. Every thread performs a two step execution where it first builds the argument tuple, and then applies the user-function. In this phase if a remote value is accessed through the random access proxy container it results in a call to *proxy_get* which is explained in section 4.3.

Outside of this tuple-generating and user-function executing section Map performs various other tasks. At the start of a Map call it waits for the constraints of both the destination and argument containers, afterwards it flushes any existing argument container. The flush is only done if there exist unflushed changes in the double buffer and it guarantees that the correct values are made available for remote reading. At the end of a Map call new constraints are set for every container used as an argument. By matching the provided arguments and user-function parameters the harshness of the constraints depends on whether the container was used with random access or pre-fetch. If it used random access then the constraint is set harshly and assumes that all nodes read from it as a precaution. However if it used pre-fetch then the node can deduce which nodes would pre-fetch it and hence set the constraint for only these nodes.

4.5 Reduce

Unlike SkePU's Map, its Reduce is a simpler skeleton far more similar to the computational pattern it is based on. The GPI implementation follows a typical distributed approach where every node is responsible for calculating their partial sum which is later combined with the rest.

Accumulation

Much like in Map, every node is responsible for the indexes that it owns, all of which are used in the local accumulation. It works by evenly dividing the local indexes among the available threads and letting them accumulate a partial sum. These partial sums are then combined by a single thread and put into the communication segment where they are globally available. After this begins the distributed accumulation phase, which is single threaded. This is an iterative process where, in every iteration, a node reads the partial sum of another node and combines it with its own. With every subsequent iteration half of the nodes are done, resulting in: $\lceil(\log_2(\#nodes))\rceil$ iterations. This results in a single node owning the final value, which it will then broadcast to all the other nodes in a manner which is the inverse to how it was generated; a single node reads the global sum, and in every subsequent iteration the number of readers double as the number of nodes holding the final value also doubles. Within a single iteration no node is read multiple times as to avoid overloading. At the end of this algorithm every node now holds the global sum. The first mentioned algorithm is a distributed reduce, whereas the latter is a distributed broadcast. They are very similar and share the property of only allowing a node to be read or perform a read at most once during an iteration. As such a node only needs to handle at most $\lceil(\log_2(\#nodes))\rceil$ requests for either one of them, reducing the chance of overloading a single node. The iterations are kept separate through operation numbers, resulting in Reduce using a large amount of them every time it is executed. This is however not an issue as the operation numbers are unsigned 64 bit integers resulting in $\approx 1.8 \cdot 10^{19}$ possible values.

Meta tasks and data dependencies

Unlike Map, Reduce outputs a scalar and as such does not use a destination container. It also does not need to flush any changes in its argument-containers as the only remote data needed is the partial accumulation. This means that Reduce does not need to use any constraints, instead other forms of synchronizations are used. As such every node is free to start its local accumulation regardless of the states of the remote nodes, however the global accumulation phase does of course require all nodes to partake. After the global accumulation phase however, a particular data dependency occurs which does not fit the constraint model. A constraint prevents a node from modifying its local values in a container segment, but not those in the communication buffer. As such every node must validate that every remote node which is going to read the global sum from it; has done so before exiting the Reduce. If a node did not do this it may replace the global sum in its communication buffer before it was read. In practice the Reduce skeleton requires such tight communication between all the nodes at this point that almost no drift is possible. As such this wait is likely quite short.

4.6 MapReduce

The GPI implementation of MapReduce works quite differently depending on if it can reuse an existing GASPI segment by having a container argument or if it can not. Both implementations are however similar and based on the previously explained implementations of Map and Reduce.

Design and Structure

The implementation of MapReduce is based on both Map and Reduce, but it has a few key differences. One of these is the fact that no argument to MapReduce is guaranteed to be a container, which is a key assumption for the Map and Reduce implementation. By having a container argument it is possible to use its partitioning scheme to divide the workload and its communication buffer for the reduction part of MapReduce. If it does not exist, the Map size needs to be provided beforehand and a GASPI segment created so that a reduction may be done. As such the behavior of MapReduce is quite different depending on if a container argument exists or not. To differentiate between the two cases the GPI backend looks at whether the first argument is a container or not. In the SkePU standard, container arguments are required to come before constant type arguments. Hence, if the first argument is a constant, all following arguments should be as well. The GPI backend is able to handle any ordering of the container- and constant-arguments for Map, but this assumption is still done as the user is expected to follow the SkePU standard. However MapReduce does handle the case outside of the standard where a constant is followed by a container like in the following example: $MapReduce(constant, matrix)$. But this handling will use the less optimized solution which does not use the argument container's GASPI segment for data transfers. This is mostly a demonstration of how the GPI backend is able to handle certain problem formulations outside of the SkePU standard and not something expected to be used.

From a high structural level both versions of MapReduce are quite similar, they divide their given indexes evenly among their available threads. And for every element they first build an argument tuple, then execute Map while immediately consuming the generated value with Reduce. Afterwards it enters a global accumulation phase similar to Reduce's where it first reduces all the partial values and then broadcasts the global value. The implementation thus leverages one of the most important aspects of MapReduce and that is immediate consumption of the generated Map value. This thus results in less memory being needed while also removing the need to write down the intermediate value only to read it later.

With a container argument

In Map the number of elements it works on is the same as the size of the destination container. In Reduce it is the size of the input container. But MapReduce is unable to do this as it has no data-container and a Map is allowed to use fewer indexes than the container has elements. As such the size of the MapReduce is defined before its execution, otherwise it defaults to the size of the first container argument as per the standard. The execution works just like a combination of Map and Reduce, it waits for the argument constraints and then flushes the containers. Afterwards it pre-fetches and builds the argument tuple and applies the map-user-function. The difference is that the result is stored in a local variable and immediately used by the reduce-user-function. Thus the local accumulation is done simultaneously as the Map part is executed. After this broadcast, the local accumulated values are combined exactly like in Reduce with a distributed reduce and broadcast. The partial sums are stored in the GASPI segment of the first container argument. This creates a data dependency which is shared with Reduce and explained more thoroughly in Section 4.5. Briefly, the partial sums may overwritten in the GASPI segment during the next operation; as such a node must ensure that the partial accumulations are no longer needed before leaving the MapReduce operation. Furthermore, unlike Map, MapReduce does not modify a destination container and as such do not need to use constraints. Meaning that unlike Map there is no section at the end of MapReduce which adds constraints.

Without a container argument

If the first argument provided is a constant then this case is chosen. While it is referred to as the ”no container case” it may in fact have container arguments after the constant argument. This does go against the SkePU interface and is discussed more above with summation being that it is possible but unadvised and slightly slower to do this.

The use case for this version of MapReduce is to first call ”setDefaultSize” and then call the MapReduce instance object, as seen in Figure 4.8. In the setDefaultSize function, a global GASPI segment is created if one does not previously exist. This segment is then used during the reduction part for the distributed scatter and gather of the local accumulations. The execution of the MapReduce works the same as the case with a container, except for the usage of the global segment. From a design perspective, another minor difference is that certain meta-data and utilities are accessed through a global singleton. For example, the Matrix class provides access to the vector clock and related waiting functions; which needed to be made available through the global singleton as well to accommodates this MapReduce case. In the other MapReduce case, as well as for the other skeletons, such utilities are accessed through Matrix as an old, and now incorrect, assumption was that every skeletal invocation would have a container argument. These utilities should be made only available through the global singleton but due to time constraints this change has not been implemented yet. This minor design oversight is unlikely to have any noticeable impact on the prototype however.

Figure 4.8: Example syntax of SkePU's MapReduce's no container version

```
int add_f(int a, int b){
        return a + b;
}

auto obj = MapReduce(add_f, add_f);
obj.setDefaultSize(N);
int result = obj(10, 10)
```

4.7 Benchmark programs

The programs presented here are used to benchmark the prototype, a more theoretical description of them is presented in Section 2.14. The programs are all based on the already existing SkePU examples of the same name, which in turn are naive solutions to their theoretical counterpart.

The n-body problem

The n-body problem program works by simulating a set of particles of the same mass in three dimensions. Furthermore the particles are spawned in a predetermined manner, which ensures that their movements are minimal and thus avoids any potential collisions. The implementation uses a SkePU Map where every particle first updates their acceleration by observing its distance to all other particles. Afterwards it updates its velocity depending on the acceleration and time interval, and finally it moves according to its velocity. It is thus a naive implementation which reads the data of every other particle through a proxy container. This program is iterative and every iteration performs the calculations explained above. The execution time which is being measured includes the iteration-loop as well as the initialization of the containers but not their creation. The evaluation of the prototype using this program uses $2 \cdot 10^5$ particles and 20 iterations.

The Mandelbrot program

This program is based on the Mandelbrot set and attempts to deduce which pixels in an image belong to the set and which do not. The implementation used in this project is based on the previously existing one from SkePU. It differs however by not gathering all the pixels on to any one single node, nor print out the resulting image. Furthermore the program was made more computationally intensive as previous runs of the already existing implementation proved to be too short for reliable measurements to be made. As such, the threshold which determines that a pixel has increased to such a large absolute value that it will never converge, was increased. Furthermore the maximal number of iterations spent at a single pixel was increased from 10^3 to 10^4. The former change increases the amount of computations done for every pixel which is not part of the set and thus reduces the load imbalance. However for pixels which are far from being in the set the difference will be marginal as they increase fast enough that the new threshold is reached quickly. The latter change increased the computational intensity of pixels which do belong to the set as they are evaluated for more iterations before being classified. This would thus increase the work imbalance and help even out the previous change.

From the SkePU syntax the implementation relies solely on a Map whose only arguments are constants in the form of the dimensions of the image. This program is thus notable in the fact that there is no communication in it and that the computational time between two pixels may differ by a factor up to 10^4. The execution of this program as a benchmark uses a picture of $60000 \times 60000 = 3.6 \cdot 10^9$ pixels. Every pixel corresponds to an element in the matrix where it is modified by a SkePU Map and the program's execution time only measures the time spent within the Map. A smaller version of this program was run and the exported image saved in Figure 2.9.

The Taylor program

This program attempts to calculate a Taylor series of the natural logarithm at a certain point using a fixed number of terms. The implementation is mostly unchanged from the original SkePU example it is based on, except for the code which measures the execution time. By applying a MapReduce of size N, every element is first transformed according to the Taylor series through the map-function and then summed up with addition as the reduce-function. This program is not especially computationally intensive, but it demonstrates the utility MapReduce has by allowing for a very large amount of elements to be used. It is thus meant to use a much larger MapReduce than would be possible with a Map due to its memory usage.

The matrix matrix multiplication program

There exists many different algorithms of matrix matrix multiplication, and even within SkePU there are multiple different implementations. The one this program is based on uses the general proxy container Mat and not the more specialized MatCol nor MatRow. As such this program does not highlight a backend's performance on a matrix matrix multiplication but rather its relative performance compared to other runs of the same program. The reason for not using MatCol nor MatRow is that neither of these are implemented in the GPI backend as discussed in Section 4.1. The program itself uses a single SkePU Map with three equally sized squared matrices of floats, two of which are randomly initialized either within the range of $(0, 9)$ or $(3, 9)$. The dimensions of the three matrices are 18000×18000 and what the program does is calculate the matrix matrix multiplication of the initialized matrices and stores it in the uninitialized one. Note that unlike many other programs this one is not iterative, the matrix matrix multiplication is only done once. Lastly, the measured execution time covers only the SkePU Map call which performs the matrix matrix multiplication, meaning that both the initializations and creation of the matrices are excluded from it.

The matrix vector multiplication program

This program is based on the SkePU example of the same name, and just like the matrix matrix version it does not use the more specialized access patterns of MatCol or MatRow. Furthermore it uses two vectors and one matrix of floats with the matrix being initialized with values between three and nine and one of the vector with values between zero and nine. Using a SkePU Map with the Mat access pattern it performs the matrix vector multiplication and stores the result in the uninitialized vector. This Map is then used as many times as there are iterations, alternating which vector is used as a data destination and which one is used as an argument. The dimensions of the matrix is 50000×50000 while the vectors are of size 50000 with the number of iterations being 10^4. The execution time measured only includes the SkePU Map which performs the matrix vector multiplication and thus excludes the initialization step of the containers as well as the creation of all objects.

5 Method

This chapter presents both what hardware the programs run on and which compiler was used and what program they were used to compile. Furthermore it elaborates on how the data points presented in this chapter were generated.

5.1 The Sigma and Tetralith cluster

All the runs presented in this thesis have been done on either the Sigma or Tetralith cluster. These two clusters are operated by the National Supercomputer Centre (NSC) in collaboration with LiU. The two clusters follow the same network architecture and use very similar hardware, Sigma is essentially just a smaller version of Tetralith. While certain nodes have different hardware, the ones which we used did not. These nodes have two Intel Xeon Gold 6130 CPUs, which each have 16 cores, 32 hardware threads, 96GiB of main memory and a SSD disk[1]. Since the clusters are expected to have equivalent performance, the test runs were split among the two in an arbitrary fashion. The reason for this was that while the performance is expected to be identical the allocated processing time of this thesis were split among them. However, most of the runs presented in this paper were done on Sigma.

5.2 Installation and compilation

The cluster version of StarPU called StarPU-MPI was built on the cluster using gcc 7.3.0, and the version of StarPU-MPI was 1.3. SkePU and GPI however were built using gcc 6.4.0. The reason for this discrepancy are building issues which mandated changing the gcc version from the default value on Sigma and Tetralith of 6.4.0 to 7.3.0. The programs themselves were compiled using g++ and its MPI equivalent mpic++, all of them using g++ version 7.3.0. Furthermore the C++ version used was C++11, which is the oldest supported version for SkePU, and the flag ”-O3” was added to all compiled programs to indicate that execution speed is to be prioritized over memory usage.

The GPI programs were compiled directly at the cluster whereas the StarPU programs were pre-compiled by another computer and then compiled at the cluster. The g++ version to

[1]NSC's website describing Sigma and Tetralith's hardware:
`https://www.nsc.liu.se/systems/sigma`
`https://www.nsc.liu.se/systems/tetralith`

build StarPU and SkePU at this computer is 7.5.0 and the StarPU-MPI version 1.3. Furthmore the GPI version used for this computer as well as the clusters was GPI-2 version 1.4.0.

5.3 Creating the measurements

The execution of the programs were limited with regard to time usage of the two clusters. This project was alloted a certain amount of ”core-hours” and thus the programs were only able to be run a limited amount of times as to not exceed the alloted core-hours. Furthermore some of these core-hours were used to debug issues which only arose on the cluster and to test run the programs in order to figure out their expected runtime. The result of this is that most programs were only able to be run a single time when creating the result for the thesis. This was deemed acceptable as during development the runtime between multiple runs of the same program with the same backend was minuscule, typically differing around a few seconds for a program which runs for thousands. Still this is a weakness of the generated data, but it was chosen rather than cutting certain programs from evaluation.

The time measured by the programs primarily correspond to the time they spend in the SkePU calls, which in turn corresponds to their most computationally intensive and most time consuming phase. This means that the creation of containers is not included in the presented execution time and neither are initializations, except for in the n-body problem. These factors are not wanted in the measurements, as they might add new variables such as the quality of their memory allocation and random number generation which would dilute the performance differences of the actual SkePU skeletons. More details regarding this is laid out in Section 4.7 for every program. The execution time generated this way is different for every node and the presented time is always that of the slowest node unless explicitly stated. The motivation for this is that the program on a global scale can not be considered finished until every node is.

6 Results

This chapter lays out the results of all the programs for both the StarPU and the GPI backend and later on in the chapter, the variance of the programs' and backends' execution speed is also presented.

6.1 The n-body problem

Minor issues

Due to an error in the implementation of the n-body-problem program the calculations were slightly different between the two backends, although not their complexities. In the StarPU backend the particles were all initialized along a line instead of spread around a three dimensional space. As such, two dimensions were never used leading to a constant velocity, acceleration and position in these dimensions. However this does not affect the amount of computations done and hence it is assumed that it should not affect the execution time in a meaningful way. This erroneous initialization was discovered and changed in the GPI backend, meaning that the two backends worked on different numerical values.

N	GPI	StarPU
1, 2	1.91	0.95
2, 4	1.96	1.95
4, 8	1.99	1.93
8, 16	1.98	1.87

Table 6.1: The differential speedup, which is the speedup between the node pair given in N, of the StarPU and GPI backend for the n-body-problem program.

Comparison

By observing the non-logarithmic graph in Figure 6.1 we can see that both backends start with a similar execution time. But when the node count goes to two the execution time of the

StarPU backend increases while that of the GPI backend decreases. After this divergence the graphs do not converge until the node count is much higher, at 16. This is however an artifact of the graph; if we look at the logarithmic one instead, we see that execution times continue with about the same relative difference between them. In fact, by comparing the relative differences it shows that at two nodes GPI has an execution time of 52.1% of StarPU, and at 16 nodes this becomes 47.3%. So while the two non-logarithmic graphs seem to be converging their relative difference is in fact increasing. We can also see this in Table 6.1 where the differential speedup of GPI is slightly higher than that of StarPU. For 16 nodes the scaling of StarPU seems to be falling off slightly as the differential speedup is only 1.87, but more data points would be needed to validate if it is a trend. Overall the performance of the GPI backend is noticeably better than the StarPU backend for this task, it scales better and starts doing so after two nodes. However, for a single node the StarPU backend is slightly faster, although this is not a particularly noteworthy data point when comparing cluster execution.

6.2 Matrix matrix multiplication

As in the n-body problem we can see in Figure 6.2 that StarPU's execution time increases when going to two nodes whereas GPI's decreases. Furthermore the pattern that StarPU is quicker on a single node also holds, but in this case it is much more pronounced. Overall we can see that StarPU outperforms GPI in all cases with the largest difference being at a single node where it is almost five times faster. By observing the logarithmic graph of Figure 6.2 we can see that the log execution time of StarPU has a sharper decline than GPI, which corresponds to better scaling. For $N = 2$ StarPU needs about 53.9% of GPI's execution time whereas for $N = 16$ it needs 44.3%. The relative difference between the two backends is thus increasing and while the non-logarithmic graphs may make it look like they are converging this is only the case in absolute numbers.

6.3 Matrix vector multiplication

In Figure 6.3 we see once again the characteristic increase of execution time for StarPU when increasing from one node to two. But we also see that the scaling of StarPU is very inconsistent, only becoming slightly faster at four nodes than at one, then stagnating and keeping about the same execution time for eight nodes as for four. At eight nodes the StarPU backend seems to reach its saturation point and becomes slower with increased node count. To contrast this, the GPI backend has a much more consistent speedup and catches up with StarPU at eight nodes and outperforms it at 16. In Table 6.2 this trend is shown even more clearly as GPI has a consistent differential speedup whereas StarPU does not. However by looking at these numbers it is clear that the scaling of GPI is limited and quite a bit away from linear differential speedup, which would correspond to a factor of two. Notably the speedup between eight and 16 nodes is particularly weak with a factor of 1.39 and may indicate that GPI is close to the point where it does not scale anymore.

N	GPI	StarPU
1, 2	1.46	0.77
2, 4	1.67	1.51
4, 8	1.60	1.05
8, 16	1.39	0.66

Table 6.2: The differential speedup, which is the speedup between the node pair given in N, of the StarPU and GPI backend for the matrix vector multiplication program.

6.4 The Mandelbrot Program

For the Mandelbrot program the StarPU backend had almost no scaling as the execution time of the program was: 1450, 1430, 1390 seconds for 1, 2 and 4 nodes respectively. Furthermore there was almost no difference in execution time between the nodes, the difference between the fastest and slowest node when using four nodes was 18 seconds. Due to these times seeming to be almost unchanged with increasing node quantity the experiments were not run for more than four nodes, allowing the computational hours of the cluster be spent on other parts of the project.

The GPI backend starts by being substantially slower than StarPU for one node, having about the doubled execution time. Unlike StarPU however this decreases noticeably with increasing node count, and at $N = 2$ it has almost caught up with the StarPU backend as shown in Figure 6.4. Furthermore we can see a large spread between the execution times of the fastest and slowest node. The execution time of the fastest node increases quicker than the average or slowest but then seems to plateau around $N = 8$. At this point there exist two nodes which finish within 14 seconds and for $N = 16$ there are four nodes that terminate within 16 seconds. The difference between the fastest and slowest node for $N = 8$ and $N = 16$ is less than two seconds, unnoticeable in the normal graph and over-emphasized in the logarithmic one.

6.5 The Taylor program

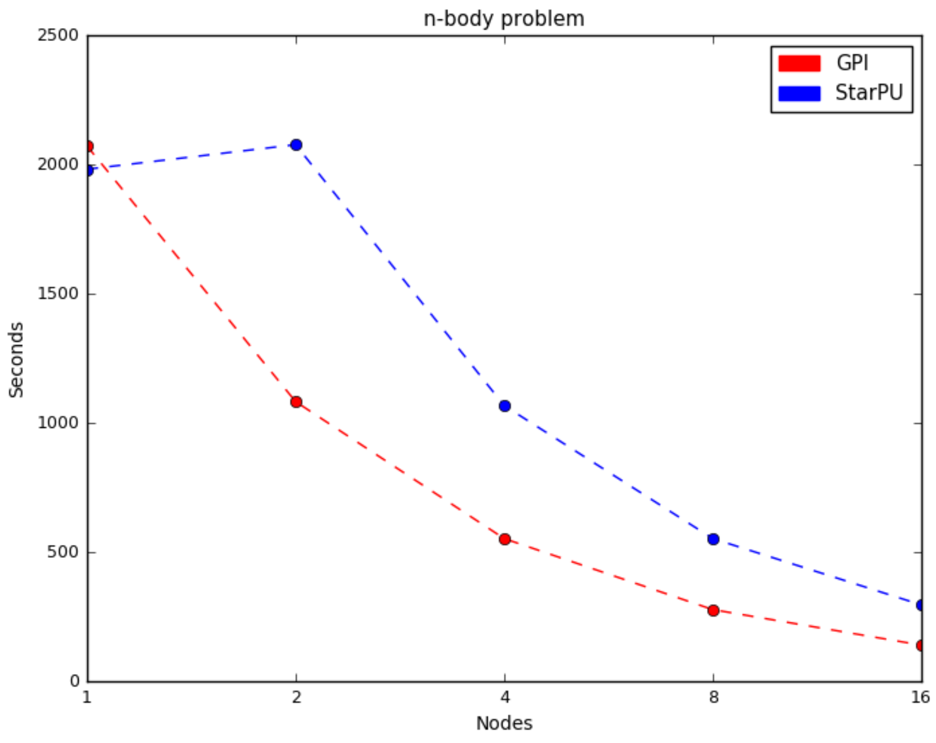
Unlike the other programs the experiment for the Taylor series only concluded with a single data point. Given the problem size of $N = 10^{11}$ the GPI backend was able to calculate the Taylor sum in just under 76 minutes using a single node, 4534 seconds to be more precise. This was not done for the StarPU backend as this program is meant to demonstrate MapReduce's capabilities and to verify that the GPI backend's implementation of it is correct.

6.6 Variance in execution time

The time each individual node took to perform the task in every program was measured and their variance formalized in Table 6.3 and Table 6.4. In these tables the variance is zero when there is only one node as there is only a single data point, but in the StarPU table the variance in the matrix vector multiplication is also zero. For this program there are multiple data points, they are just identical. In Section 6.4 it is mentioned that the Mandelbrot program is only run with up to four nodes with the StarPU backend and hence some data is missing in Table 6.4's Mandelbrot column, indicated by a "-".

By observing the two backends it is clear that the GPI backend has a much higher variance than the StarPU one, the difference for the same program and nodes amount are always magnitudinal, and sometimes upwards of six orders. But despite this, the amount of variance in the programs seems to follow the same order within both backends. This is in descending order of variance: Mandelbrot, matrix matrix multiplication, and n-body problem. Missing from this ordering is the matrix vector multiplication which has no variance in the StarPU backend but enough variance to put it between the matrix matrix multiplication and the n-body problem in the GPI backend.

The metric "deviation / mean" shows how large the deviation is as a proportion of the mean and helps illustrate how much the nodes differ given the total runtime. Using it we can see that for the StarPU backend the proportionally highest standard deviation is still only about 0.6% of the mean execution time. In the GPI backend for the Mandelbrot program this becomes much more pronounced however, reaching up to 85% of the mean execution time. In this case a node which is within one standard deviation away from the mean will only run for 15% of the mean runtime. Apart from this problem however the difference in the nodes' execution time remains fairly low when compared to the mean, even for the GPI backend. In both the n-body problem and matrix vector multiplication the deviation is less than 0.1% of the mean for all tested node amount. It is significantly higher in the matrix matrix multiplication program reaching up to 5% of the mean when using the GPI backend. To contrast in the StarPU backend the variance never reaches above 0.22% of the mean in the matrix matrix multiplication, demonstrating just how much lower its variance is.



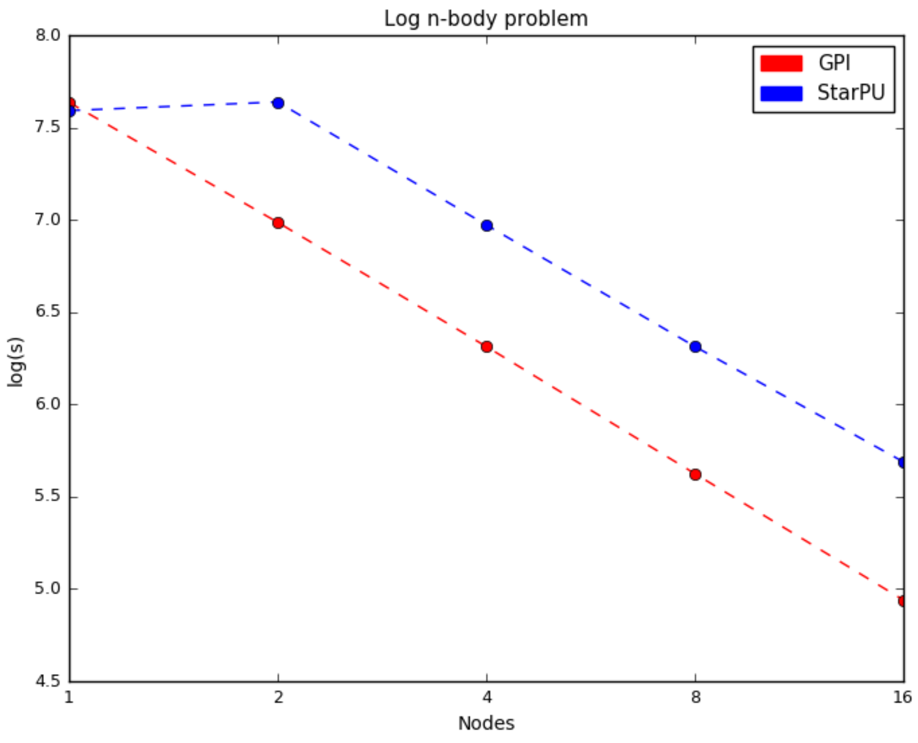
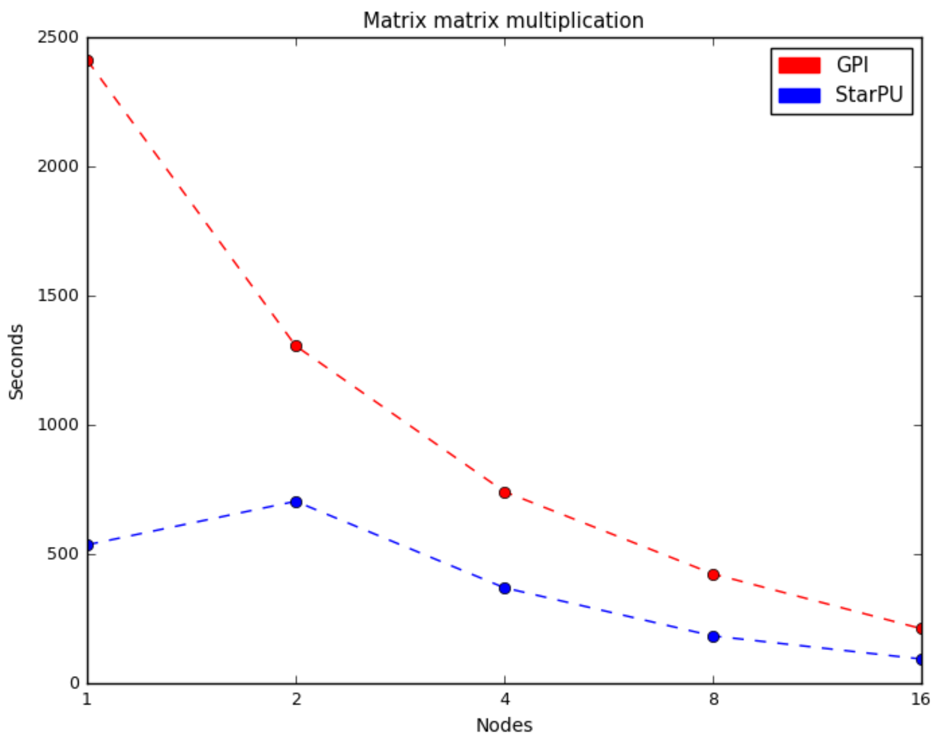


Figure 6.1: Runtime of the n-body problem with $2 \cdot 10^5$ particles running for 20 iterations.



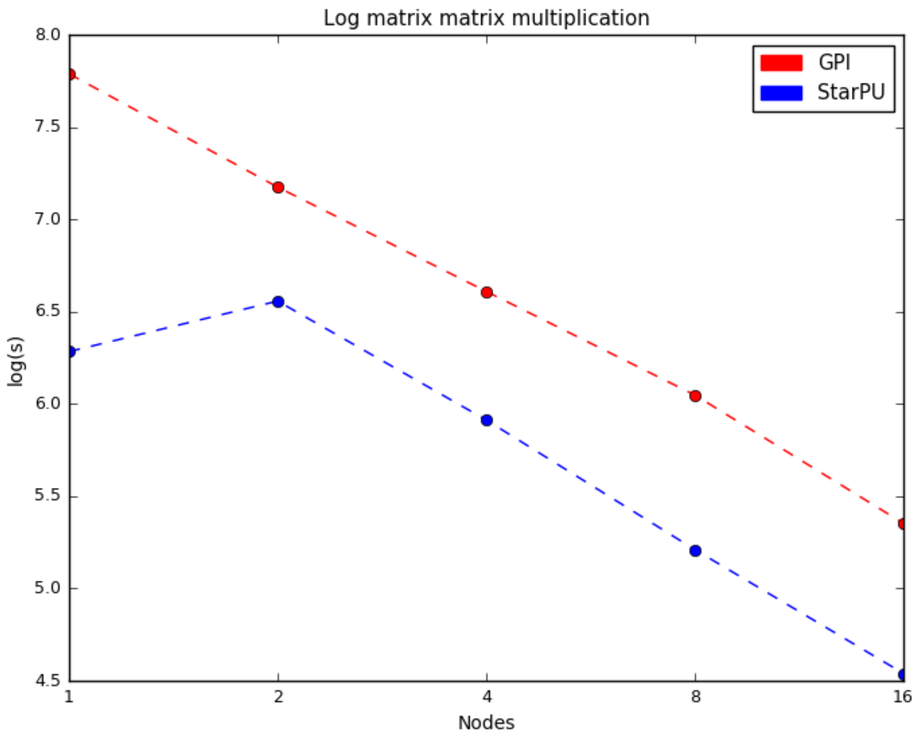


Figure 6.2: Runtime of the matrix matrix multiplication program $M_1 = M_2 \times M_3$ where both M_2 and M_3 is of dimension 18000×18000 totaling in $3.24 \cdot 10^8$ elements each.

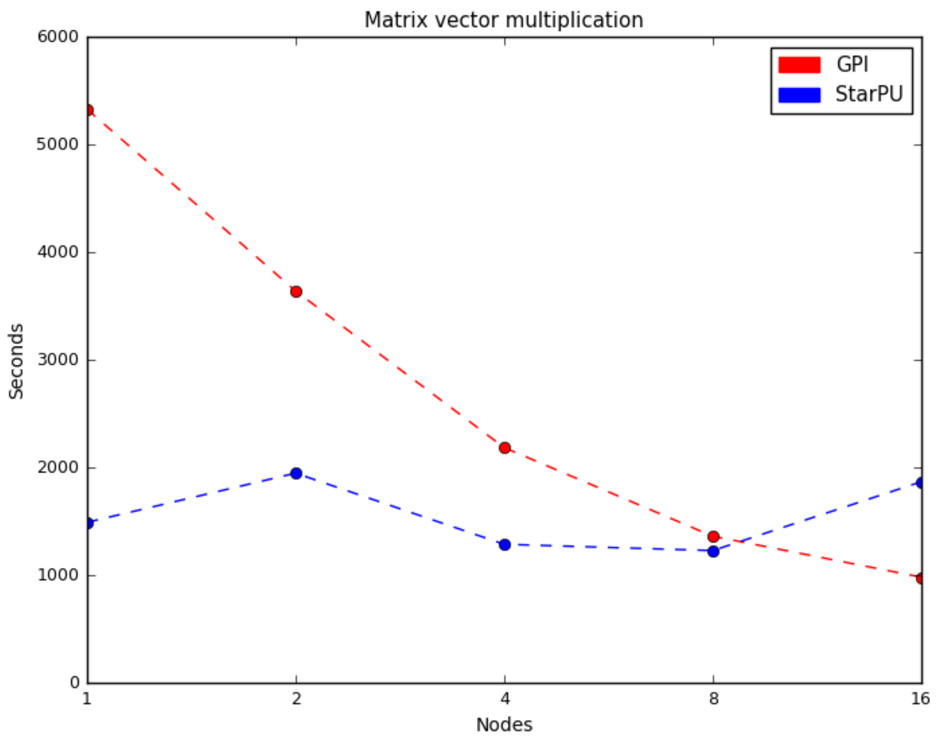
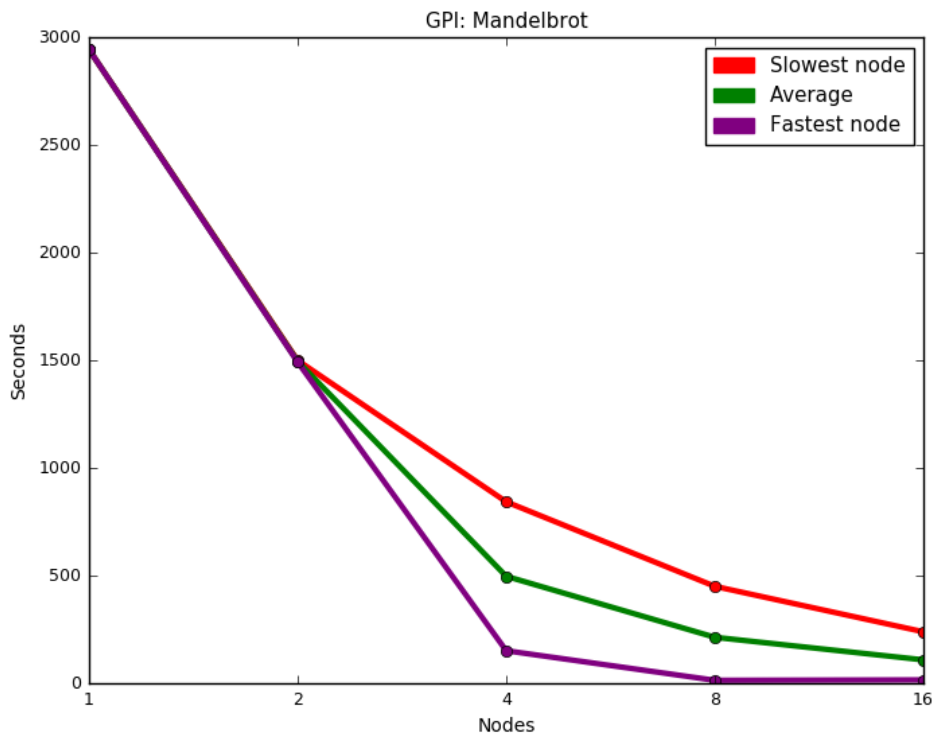


Figure 6.3: Runtime of the matrix vector multiplication program $M = MV$ where M is of dimension 50000×50000 and V of length 50000. M thus has $2.5 \cdot 10^9$ elements and the program was run for 10^4 iterations.



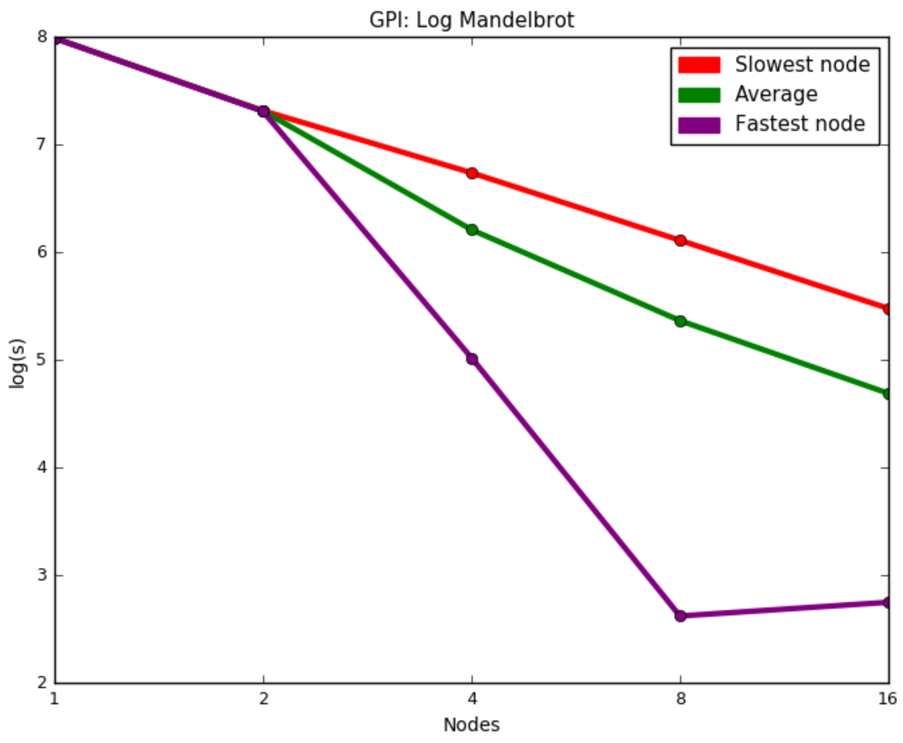


Figure 6.4: Mandelbrot program on a 6000×6000 matrix where the max iteration cap for intensive pixels is 10^4.

GPI								
	n-body problem		Matrix matrix multiplication		Matrix vector multiplication		Mandelbrot	
N	Variance	Deviation / Mean	Variance	Deviation / Mean	Variance	Deviation / Mean	Variance	Deviation / Mean
1	0	0	0	0	0	0	0	0
2	$5.00 \cdot 10^{-5}$	$6.54 \cdot 10^{-6}$	191	$1.07 \cdot 10^{-2}$	0.64	$2.18 \cdot 10^{-4}$	5.48	$3.66 \cdot 10^{-3}$
4	$6.67 \cdot 10^{-7}$	$1.47 \cdot 10^{-6}$	191	$1.89 \cdot 10^{-2}$	$1.55 \cdot 10^{-3}$	$1.80 \cdot 10^{-5}$	497	0.80
8	$5.00 \cdot 10^{-7}$	$2.55 \cdot 10^{-6}$	388	$5.10 \cdot 10^{-2}$	$3.70 \cdot 10^{-3}$	$4.49 \cdot 10^{-5}$	$3.32 \cdot 10^{4}$	0.853
16	$1.58 \cdot 10^{-6}$	$9.00 \cdot 10^{-6}$	26.4	$2.59 \cdot 10^{-2}$	$6.40 \cdot 10^{-4}$	$2.58 \cdot 10^{-5}$	$7.35 \cdot 10^{3}$	0.790

Table 6.3: The variance between the nodes when executing the problems with the GPI backend. The deviation divided by mean indicates how much the execution time varies between nodes as a proportion of the runtime. Thus a value of 10^{-2} in this column corresponds to the standard deviation being 1% of the mean execution time.

StarPU								
	n-body problem		Matrix matrix multiplication		Matrix vector multiplication		Mandelbrot	
N	Variance	Deviation / Mean	Variance	Deviation / Mean	Variance	Deviation / Mean	Variance	Deviation / Mean
1	0	0	0	0	0	0	0	0
2	$4.08 \cdot 10^{-7}$	$3.08 \cdot 10^{-7}$	$2.88 \cdot 10^{-4}$	$2.41 \cdot 10^{-5}$	0	0	0.658	$5.67 \cdot 10^{-4}$
4	$7.06 \cdot 10^{-5}$	$7.88 \cdot 10^{-6}$	$6.95 \cdot 10^{-4}$	$7.13 \cdot 10^{-5}$	0	0	76.6	$6.36 \cdot 10^{-3}$
8	$5.51 \cdot 10^{-5}$	$1.34 \cdot 10^{-5}$	$1.64 \cdot 10^{-4}$	$6.98 \cdot 10^{-5}$	0	0	-	-
16	$1.04 \cdot 10^{-3}$	$1.09 \cdot 10^{-4}$	$4.21 \cdot 10^{-4}$	$2.19 \cdot 10^{-4}$	0	0	-	-

Table 6.4: The variance between the nodes when executing the problems with the StarPU backend. The deviation divided by mean indicates how much the execution time varies between nodes as a proportion of the runtime. Thus a value of 10^{-2} in this column corresponds to the standard deviation being 1% of the mean execution time. Note that the zeros in the table are due to the underlying data and "-" represents a lack of data.

7 Discussion

In this chapter the results of the programs are analyzed using the knowledge of how the prototype works provided in the implementation chapter. First the performance of the programs are discussed and how the design affects this result. Afterwards the design itself is evaluated given how it affects the programs and other important factors such as usability. Lastly the node variance of execution speed is discussed and then the thesis is analyzed in a wider societal context.

7.1 Results of the programs

In this section the results of the programs are discussed and how the design affects this.

The n-body problem

This program showed the most promising results for the GPI backend as it performed better than the StarPU one for all numbers of nodes except one. Afterwards at two nodes the StarPU backend's characteristic increase in execution time meant that the two backends diverge drastically at this point. But it is not the case that they diverge at $N = 2$ only to converge later as the scaling of GPI continues to be slightly better than StarPU's as shown in the Table 6.1. As such the points where the relative performance of the StarPU backend is the best compared to the GPI backend is at $N = 1$ followed by $N = 2$ and then by the remaining indexes in ascending order. Thus the GPI backend both scales better and has a lower execution time when using at least two nodes.

These good results likely stems from the fact that this problem matches very well with the underlying implementation of the GPI backend. Whenever a remote value is read through the proxy container all remote values of the same node are also read, all of which are needed in the problem. This means that all the elements in the matrix which are transfered alongside the requested element are all used, which is unlikely to be the case given an arbitrary user-function. Furthermore the particles are accessed from lowest to the highest index by the user-function, which matches how they are partitioned internally by the Matrix class resulting in a high cache hit rate. Thus this problem matches very well with the underlying design of the GPI backend, more so than what can be expected by an arbitrary SkePU program. However it also matches well with GPI itself as it first accesses all elements from one node and then performs all calculations using these elements before accessing another node. Thus the program has overlapped its computations and communication sections in a manner which GPI needs in order to achieve good performance. Thus to conclude the GPI backend's performance on this task is significantly better than what can be expected and as such should not be viewed as representing its performance in general. Instead it can serve as a demonstration of what the backend is capable of when its design and given program lines up.

The matrix matrix multiplication program

For this program the GPI backend performs poorly for a few nodes while also scaling worse than the StarPU backend, essentially showing a worse performances for any amount of nodes. Compared with the n-body problem the GPI backend performs significantly worse here despite some similarities between the problems. In both of them the entire content of a container is transferred to all nodes through the proxy container. The difference however lies in how the elements are transferred, for in the matrix matrix multiplication an entire column of elements is needed before the first element can be calculated. Assuming that $\#elements \gg \#nodes$ a column will always be distributed over all existing nodes. Combined with the fact that remote reads fetch all elements of the target, it results in the whole container being transfered for the first indexed element. An aspect which was pointed out as being needed for achieving good performance with GPI is the need for computational and communicational overlapping [37]. As such having an intensive communication phase followed by a computation phase is ill fitted with how GPI should be used, and as such poor performance is expected.

Another issue is the cache locality of how the transferred elements are used. They are stored in a contiguous row-wise array and hence when accessing them column-wise poor cache locality is expected. In fact, as the size of the matrix in this problem is very large, 18000×18000, it is not possible for the node to cache it in its entirety. As such whenever the second element in a row is accessed the whole column must first be read, and by this time the row has been removed from the cache. This thus means that every column access is a guaranteed cache miss, heavily slowing down the execution time. This is a general problem for matrix matrix multiplication, but the way the GPI backend stores the remote values ensures that it occurs. If the transfers were more granular, storing only partial rows in a contiguous memory, then a single cacheline might correspond to multiple partial rows. In this case the cache might be able to store a few columns in the cache, which if possible would reduce the cache miss rate. The issues here are thus two fold, firstly the transfers are too coarse and thus front-load the communications too much. Secondly the way the transferred elements are stored is ill-suited for the cache.

Lastly it worth noting that the this problem is difficult to handle with a simple random access pattern such as the proxy containers Vec and Mat. As such SkePU has more specialized ones for these cases, such as MatRow and MatCol. This comparison is done without them as they are not implemented in the GPI prototype as mentioned in Section 4.1, but this means that results should only be viewed relative to each other. The StarPU backend's performance on a matrix matrix multiplication is better than the presented execution time in this thesis as the program itself is unoptimized and does not use all of the StarPU backend's features.

Matrix vector multiplication program

For this problem we can see that the GPI backend scales better while the StarPU one has a more uniform run time which is lower up to and including eight nodes. The big contrast between them is the constant speedup for the GPI backend for every point, as shown in Figure 6.3. As mentioned in Section 2.14 the complexity of matrix vector multiplication grows in the same order of magnitude in both the communication and the computation aspect. Hence achieving a speedup for it is more difficult as both factors need to scale. By comparing the speedup in the n-body problem shown in Figure 6.1 and matrix vector multiplication's speedup in Figure 6.2, we can see that the speedup in the later is lower even for the GPI backend. This illustrates how it is more difficult to achieve speedup for the matrix vector multiplication. The StarPU backend is thus able to achieve good scaling at the matrix matrix multiplication problem but not at the matrix vector one. This seems to indicate that its issues lie with the communicational aspect rather than the computational one.

The design of the GPI backend matches much better with this program rather than the matrix matrix one. It only uses local values within the vector and it reads entire rows of the Matrix to calculate the result. As discussed in Section 4.3 a row may only be owned by at most two nodes assuming that $\#elements >> \#nodes$. Hence, while accessing a row in the matrix a node has to do between zero and two remote reads depending on who owns the row. Since these calls transfer all elements from the remote node this still results in a large scale data transfer at the start of the execution, but fewer of these transfers are done compared with the matrix matrix multiplication program. Furthermore the transfers may be more spread out in the matrix vector program, for example the first and last element within the same row may be owned by different nodes. This results in this iteration having two communication phases separated by a computational phase. Furthermore once an element has been accessed from a remote node, no more transfers will be done from it for subsequent indexes. As such an iteration dominated by a long communication phase may be followed by multiple iterations without any such phase. Which in turn matches decently with the desired interleaving of GPI and may explain why the GPI backend's performance is notably better for the matrix vector multiplication relative the matrix matrix one.

The Taylor program

The Taylor program's experiment demonstrated that the GPI backend is able to perform tasks with MapReduce which would not be possible with a map followed by a reduce. In this experiment to store the matrix of floats with size 10^{11} the current GPI implementation would require 400 GiB of main memory for each node. Even with a less memory intensive implementation it can not avoid allocating on average $\frac{400}{N}$ GiB given N nodes. While this is certainly not impossible, it would either require expensive hardware for low values of N or that the program is only executable using multiple nodes. This experiment has demonstrated that both of these issues can be sidestepped using a MapReduce instead of a Map for the Taylor program.

The measured execution speed of 76 minutes is much slower than what would be reasonable for a program of such size. Hence it is likely that some unwanted effect is also being measured, such as for example the swap speed of the virtual memory. This thus hints at a performance issue somewhere within the Taylor program or the GPI backend itself.

The Mandelbrot program

The StarPU backend generated curious results for the Mandelbrot program as the execution time did not scale with increased nodes. It is possible that either a bug in the program or in the backend might have lead to this strange non-scaling and more experiments are needed to rule out this possibility. Assuming this is not the case, another explanation would be that the additional overhead of StarPU's load balancer is enough to offset any gains in increased computational power. As this issue only arises in the Mandelbrot program, which does not include a communications phase as stated in Section 4.7, the primary scaling overhead-cost would come from StarPU's scheduler. Furthermore the main property which differentiates this program from the others is its large work imbalance. For a dynamic scheduler to perform poorly at an unbalanced task is however unexpected as it is typically one of its main strengths as discussed in Section 2.10. However as the StarPU backend is only dynamically scheduled internally within a node, it may still struggle with imbalances between the nodes. Regardless, further research is clearly needed before the cause of the poor scaling is able to be determined.

Unlike the StarPU backend the GPI one is statically scheduled which may lead to many issues with the Mandelbrot program. Firstly, the theoretical performance of a static scheduler on the Mandelbrot program varies significantly depending on the quality of the partitioning of the pixels. If a set of pixels is divided into two it is possible that all of the intensive ones end up in one of them. Which means one partition gets essentially all of the work and the other one gets none. The partitioning of the Mandelbrot program in the GPI backend is the same as that of the Matrix class, hence every node gets a one-dimensional contiguous row or rows of pixels. Figure 2.9 is a lower resolution version of the image which the Mandelbrot program works on. By observing it we can see that most rows contains pixels both in and outside of the set except near its top and bottom. As such, the load imbalance occurs when the amount of nodes increases enough to make some partitions primarily contain pixels at the top and bottom of the image. Per Figure 6.4 this seems to be when $N = 8$ as that is when the fastest node stops scaling meaningfully. Furthermore the partitions around the center of the image contain a large amount of intensive pixels. By decreasing their size while keeping the intensive to non-intensive pixels ratio the amount of work decreases as the amount of nodes increases. This is why the program gets a speedup with increased node amount while using a static scheduler.

The internal parallelization of the GPI backend is also static, meaning that every thread is given a set of pixels at compile time. This is likely to lead to the same load imbalance as the one between nodes since some threads may get many more intensive pixels than the others. However this is not measured in any experiment and as such it is not possible to tell as to which extent such an imbalance exists and to what degree it affects the program. An indication of this issue is seen in the large discrepancy between the GPI backend's performance and the StarPU one's when using a single node. For this the GPI backend needed 2900 seconds to the StarPU backend's 1500 seconds, almost doubling its execution time. While this may be due to a work imbalance among the threads it is important to note that the StarPU backend outperformed the GPI backend in every program when using a single node. In most of them the discrepancy was even larger than this, in fact this relative difference is the second lowest one after the n-body problem. As such it is not clear whether this imbalance has a significant effect on the program's run time or not. But regardless of if it does or not, the fact that the GPI backend is able to achieve a speedup using a doubly static scheduler for such an imbalanced program is unexpectedly positive. Especially considering that no speedup is guaranteed due to the static scheduling.

7.2 Design and implementation

This section outlines strengths and weakness of the GPI backend's implementation and design as well as discussing how they affect the experiments. Another focal point is on how well the design is able to utilize GPI and which changes might improve this factor.

Drift and memory consistency

The design of the GPI backend has put a significant focus on keeping the memory consistency weak and allowing a large drift between the nodes. This is achieved with state tracking logic using operation numbers, flush states and tracking modifying operations. As such the design only requires synchronization after a modification has been applied to a container with an unflushed change, thus in theory allowing for an infinitely large drift as long as a container is not used after a transformation. This feature is important to ensure that the prototype does not enforce data constraints which do not exist. However the reality of the experiments does not make use of it as they are all either iterative with an implicit barrier after every iteration, or non iterative. As such the maximal drift is at most one iteration in all of the experiments. The motivation for this feature is for the backend to use less communication and synchronization, which should lead to improved performance. But as there are no experiments which evaluate the effect of this feature it is difficult to tell what impact it had on the performance.

Communication and computation overlapping

An important aspect of the GPI backend is the overlapping of computations and communications as it is a prerequisite for GPI to outperform MPI [37]. The prototype's design attempts to utilize this through its random access pattern through proxy containers used in Map and MapReduce. The effectiveness of this however largely depends on the structure of the user-function. For example, as discussed in Section 7.1, the program based on the n-body problem does this well whereas the matrix matrix multiplication fails to do so. The prototype's design thus allows for the interweaving, but the user-functions seems to determine whether it actually occurs or not. The difference between a well-fit and ill-fitted user-function is seen in the performance difference of aforementioned programs of Section 7.1. The result of this variance is reduced user-friendliness as it requires that the user is aware of the internal workings of the GPI backend. In contrast to this stands the purpose of an algorithmic skeleton, which is to abstract away hardware details as explained in Section 2.9. Thus the current design does leverage the strengths of GPI is some cases, such as the n-body problem and to a lesser degree the matrix vector multiplication program, and in some cases such as the matrix matrix multiplication program it does not. By making the data transfers more granular the design could be made more robust and difficult to front-load with communication. Such a change would likely improve the prototype's performance for tasks such as the matrix matrix multiplication program. However programs which access all elements of a node sequentially and already do not front-load the communications may perform worse with it. They would have to do multiple reads, which have a high upfront cost, from the same node unlike the current implementation. Dividing the execution into more communication and computations phases might however offset this downside even for programs which are well adapted to the current design. However as the variance of the program's performance is so large it seems warranted to attempt to normalize them more. Furthermore, if such a change results in making it more difficult to frond load the GPI backend's communication it could also results in improved usability.

The GPI backend has another way of accessing elements, the pre-fetch method used when the desired elements from a remote node are known before the execution of the user-function. This accessing pattern is also very coarse in the way that it first fetches all remote values before starting the computational phase. Dividing the phases in this way is problematic as it actively prevents an interweaving of computations and communications, making poor use of GPI. A more granular, streaming based, approach of fetching a subset of the remote elements followed by executing the index-subset and then repeating, would make much better use of GPI. The true performance of this feature is however hard to know as it was not compared to the random access pattern in any experiment. But as it conceptually breaks an important aspect of GPI it can safely be assumed that it is not beneficial for the execution time.

Memory handling

As explained in Section 4.2 the prototype uses a very large amount of memory as every container object has a communication buffer with the size of the container's global size. This solution is essentially a stub in order to limit the amount of functionality needed to be implemented. As such it is a lacking solution if the GPI backend is to be extended beyond a proof-of-concept implementation. The problem which this stub hides is the fact that the sizes of GASPI segments are immutable and the amount of them is limited to 256 segments. Given these facts, three categories of implementation are proposed:

1. Expand the communication buffer when needed through creating new segments, limiting the backend to only handle smaller programs with a few containers.

2. Having a global segment handler shared among all containers, which dynamically creates segments as transfer space is needed.

3. Having a single communication buffer per container which is logically divided into "cache lines" storing transferred data. When the buffer becomes full previously transferred data is ejected, thus closely mimicking a cache.

All of these come with their own strength and weaknesses. The first one is easy to implement but severely limits the use-cases of the backend. The second one comes with an overhead cost as these segments are adapted to the current task and attempting to reuse them for other tasks is not always possible. For example a reduce only needs $N * T$ bytes where N is the number of nodes and T is the size of the data type being reduced. If this reduce is followed by a large map, then the old segment is too small and a new one has to be created. Thus it must either over-allocate, which is the primary problem of the current implementation, or perform multiple allocations. The allocation problem is inherent to all dynamic memory systems and it comes with a trade-off between efficient memory usage and increased execution time through multiple allocations. The third solution requires elaborate multi-threading logic in order to determine the state of the whole buffer and to know when a cache line needs to be ejected. However it should be able to utilize a limited amount of memory better than the second solution, and it will also allow for more containers as it does not create new dedicated segments for data transfers. This solution is limited by thread synchronization overhead but may use a limited memory more efficiently than the other solutions. Thus the most promising alternatives are (2) for most systems but (3) for systems where memory is precious. Hence, which method would be the best fit for improving the GPI backend would depend on how it is intended to be used. If its scope does not change then (2) would fit the best.

There are of course multiple solutions outside of these three categories, some of which are based on external libraries or tools. In the GPI specification the developers mention that they do not want to implement a memory management functionality as its performance would be too dependent on the specific problem [21]. Instead they have created a different tool based on GPI called GPISpace, which is one abstraction-layer higher and thus provides its own solution to this problem [36]. Using GPISpace would essentially be an entire new backend of its own, but it is worth noting that there is a solution to this problem provided by the GPI team. Any extension of the current backend needs to handle the memory allocation issues, and as seen in this section there are many approaches possible.

7.3 Performance comparison

The performance of the GPI backend is highly dependent on the task, however when the task and design line up it is capable of solving a problem at almost half the time of the StarPU backend, as shown in the n-body problem. The issue is however that the coarse data transfer makes it easy to front-load the communications to such an extent as to reduce the performance. As such, the GPI backend seems to have a lower execution time if this can be avoided, which is not user-friendly as it puts an undue responsibility on the user. Furthermore the GPI backend scales better for most of the tested problems, but the StarPU backend typically outperforms it for a lower node amount. For example, in every problem the StarPU backend has a better performance with one node and in all but one case with two nodes. The previously mentioned scaling trend is most clearly shown in the matrix vector multiplication program. In it the GPI backend achieves mediocre scaling whereas the StarPU one does not scale at all. As such the StarPU backend starts with a significantly faster execution time, but as more nodes are added it is surpassed by the GPI backend, as seen in Figure 6.3. These things point to a trend where the GPI backend is more adept at executing on a large number of nodes whereas the StarPU one uses fewer nodes more effectively. The clear outlier of the trend is the matrix matrix multiplication program where the StarPU backend is superior with both a few and many nodes. But for the matrix vector multiplication, n-body problem, and the Mandelbrot program the trends holds.

Of particular note is the matrix vector multiplication as it is equally bound computationally and through its communications, as explained in Section 2.14. In order to scale such a problem both of these factors need to scale with an increased amount of nodes. Which is contrasted with a computationally bound program such as a matrix matrix multiplication where increased computational capacity is enough to scale the program. As the StarPU backend is able to scale the matrix matrix multiplication but not the matrix vector multiplication it seems likely that its bottleneck lies within its communications. This hypothesis is further strengthened by the fact that the StarPU backend's execution time is higher when using two nodes rather than one for all of the tested programs. As such, the primary strength of the GPI backend when compared to the StarPU backend seems to be its scaling potential through its different communication schema. GPI itself is of course integral to this, hinting that it may be the primary reason for the backend's scaling potential.

Lastly the StarPU backend seems to struggle with its work load balance when iterations are very uneven, as demonstrated by the Mandelbrot program. This is particularly strange as it uses a dynamic scheduler internally but still gets out-performed by the doubly static scheduler of GPI when using four or more nodes. Overall the potential for a bug to be the reason for this performance needs to be considered. If this is not the case then it seems likely that either the StarPU backend's dynamic scheduler incurs a very high overhead cost or that the work imbalance between its nodes prevents it from scaling.

7.4 Variance in execution time

The results presented in Section 6.6 show a very marginal variance in execution times for all programs except the Mandelbrot one. This is expected considering the low drift potential of the iterative programs such as the n-body problem and matrix vector multiplication. In these every node has to communicate with other nodes and is unable to drift ahead more than one iteration, as such the difference in execution time may not grow very large. The matrix matrix multiplication is however done in a single iteration and in the GPI backend there is no barrier preventing a node from finishing it earlier than another. Thus the drift in this program is larger than for the iterative ones. The last program, the Mandelbrot one, has as expected significantly higher variance than the other ones. Unlike the other programs it is heavily imbalanced by design and in both backends this is illustrated. However, in the StarPU backend even this program only makes the node's execution speed vary by less than 1%. The reason for this is likely that the locally dynamic scheduling of StarPU is able to better balance its workload, which would also explain why it has a lower variance for all programs relative to the GPI backend. However, if the StarPU backend would execute the Mandelbrot program on more nodes, it is likely that some of the nodes would receive very little work and thus vastly increase the variance. Apart from this, the perfectly synchronized execution speed of the matrix vector multiplication for the StarPU backend is rather strange and likely due to some issue within the program's implementation or possibly even in the backend itself.

The result that the statically scheduled GPI backend has a higher variance for all programs and node combinations is expected simply due to the different scheduling schemas. However the execution speeds presented in previous sections indicate that in some of the runs the GPI backend outperforms the StarPU backend, this is despite its variance always being higher. As such it illustrates that workload imbalances and subsequent unequal execution speed among the nodes do not necessarily mean that the system performs poorly. The overhead cost of balancing the workload can exceed its benefit, which be a factor in the StarPU backend's performances on the Mandelbrot program.

7.5 The work in a wider context

By attempting to improve the cluster execution of SkePU this will hopefully help bridge a gap between the different communities which SkePU caters to. The common denominator among these are the desire to execute code in parallel but other than that, they may be very disparate groups. Some of them may use SkePU to execute parallel CPU code, parallel GPU code or make use of its cluster capabilities. Improving SkePU and offering a single tool for these different usages may help unify these communities, furthering an exchange of ideas.

Apart from this the usage of GPI may help to demonstrate that there exists alternative tools to MPI and help with creating a more diverse ecosystem of message passing techniques. This is not strictly positive as it may make migrating and integrating different systems more difficult as they might not use the same technique. But it may also allow for more tools to exist which are better adapted to fit different systems and use-cases. While this project in itself will not have such an effect, it might be part of a larger evaluation of MPI alternatives, and possible reduce its dominating status in message passing.

Given an even wider context, the work done in this thesis aims at making it easier to use the complex hardware of a cluster. This in turn would make computationally demanding programs easier to write and hence accessible for more people, companies and organizations. If physics simulations become easier to write perhaps fewer on-ground experiments are needed, in turn reducing the monetary and environmental cost of building it. Furthermore, as simulations do not require any real-world construction they could be run more frequently and by more people due to their lower startup cost. Thus allowing for more competition as smaller companies would need fewer expensive on-ground experiments. From the scientific perspective making it easier to write cluster-code reduces the need for outside expertise in the form of a parallel programmer, allowing the research to progress faster. Hence the consequences of easier to use clusters is likely that more simulations are run, and hence that more solutions are able to be considered. From a societal perspective this might be viewed as more innovations being made quicker. However in the short-term the effects may primarily lie with increased electricity usage and hardware tear stemming from the larger cluster usage.

8 Conclusion

In this section the research questions are answered and potential extensions of the thesis presented.

8.1 Research questions

In this thesis it has been shown that the GPI backend's ability to overlap its communication and computation largely depends on the structure of the given user-function. An ill-fitted user-function may front-load the communications with the random access proxy container and as such create distinct computations and communications phases. If the prototype on the other hand uses the other communication pattern, pre-fetch, the backend is guaranteed to not overlap in the computations and communications. As such, it is clear that the prototype fails to have this overlap in these cases. However, there are also cases where the prototype does manage to achieve such an overlap, such as in the n-body problem and matrix vector multiplication. Hence the answer to research question (2) is that the backend is able to overlap its computations and communications, as this has been achieved in at least some cases. However, whether this overlapping actually occurs largely depends on how a specific user-function uses the backend. As such, while the backend is capable of overlapping its computations and communications it is not a given, nor possible for all user-functions.

The execution time comparison between the GPI and StarPU backend does not point out one as superior to the other. Instead what can be shown is that the StarPU backend is consistently faster for fewer nodes whereas the GPI backend has better scaling potential for certain tasks. However this scaling trend does not hold for all experiments as the StarPU backend scales better when the program's structure would result in front-loading its communications in the GPI backend, such as the matrix matrix multiplication. Apart from this, heavily imbalanced tasks seem to favor the GPI backend despite it using a static scheduler unlike StarPU's locally dynamic one. As such the answer to research questions (1) is that the GPI backend performs better relatively to the StarPU backend at tasks which do not front-load its communications, and at heavily imbalanced tasks. Furthermore the GPI backend performs poorly when compared to the StarPU backend if only a few nodes are used. Similarly, in reference to research question (3), we can conclude that the GPI backend scales better for tasks which avoid front-loading and also tasks which are heavily load imbalanced.

8.2 Future work

The work done in this thesis provides ample opportunity for future extensions in the form of research and improvements of the prototype. Firstly more rigorous evaluation of the GPI backend's internal parallelization would be of interest. Comparing the dynamic and static scheduling schema and their performance on different tasks would allow for a more informed decision regarding which one to use. Furthermore, the tool used in the internal parallelization OpenMP, may be compared with other ones. As SkePU has backends for both CUDA and OpenCL these two would be a natural choice and allow for GPU usage. The final part of such an extension would be to attempt to use one of SkePUs single node backends for the internal parallel execution. Possibly by having a modular approach as to which of the single-node backends is used internally.

Another direction for future work lies with using more and different metrics to evaluate the GPI backend. Currently the one used is execution time, but other metrics such as memory and communications usage would help give a more detailed view of the performance. In a similar vein adding additional benchmarks to compare the results to other than the StarPU backend would also help outline its the strengths and weaknesses. In this thesis it may be difficult to tell whether performance differences arise from one backend performing well or from the other one under-performing. By using more benchmarks such as stand-alone implementation of a specific program using MPI, the results would be more robust and comparable.

Finally the last direction for future work based on this thesis comes in the form of implementing the suggested improvements highlighted within it. The largest ones being the memory management models outlined in Section 7.2 as well as making its data transfers more granular as discussed in Section 7.2.

Bibliography

[1] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, et al. “GASPI – A Partitioned Global Address Space Programming Interface”. In: *Facing the Multicore-Challenge III*. Springer, 2013, pp. 135–136.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. “A View of the Parallel Computing Landscape”. In: *Commun. ACM* 52.10 (Oct. 2009), pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783.

[3] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. “StarPU-MPI: Task programming over clusters of machines enhanced with accelerators”. In: *European MPI Users’ Group Meeting*. Springer. 2012, pp. 298–299.

[4] Josh Barnes and Piet Hut. “A hierarchical O (N log N) force-calculation algorithm”. In: *nature* 324.6096 (1986), pp. 446–449.

[5] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. “Generic parallel programming using C++ templates and skeletons”. In: *Domain-Specific Program Generation*. Springer, 2004, pp. 107–126.

[6] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP - Portable Shared Memory Parallel Programming*. Cambridge, Massachusetts: The MIT Press, 2007.

[7] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. *The münster skeleton library muesli: A comprehensive overview*. Working Paper 7. ERCIS, 2009.

[8] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[9] HMP Couchman. “Mesh-refined P3M-A fast adaptive N-body algorithm”. In: *The Astrophysical Journal* 368 (1991), pp. L23–L26.

[10] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems - Concepts and Design*. 5th ed. Boston: Pearson, 2011, pp. 609–610.

[11] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[12] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. “The LINPACK benchmark: past, present and future”. In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[13] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users’ guide.* SIAM, 1979.

[14] Johan Enmyren and Christoph W. Kessler. “SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems”. In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications.* HLPP ’10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 5–14. ISBN: 9781450302548. DOI: `10.1145/1863482.1863487`.

[15] Steffen Ernsting and Herbert Kuchen. “Algorithmic skeletons for multi-core, multi-GPU systems and clusters”. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[16] August Ernstsson. *Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems.* 2020. DOI: `10.3384/lic.diva-170194`.

[17] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. “SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters”. In: *International Journal of Parallel Programming* (May 2021). ISSN: 1573-7640. DOI: `10.1007/s10766-021-00704-3`.

[18] J.F. Ferreira, J.L. Sobral, and A.J. Proenca. “JaSkel: a Java skeleton-based framework for structured cluster and grid computing”. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID’06).* Vol. 1. 2006. DOI: `10.1109/CCGRID.2006.65`.

[19] Göran Forsling and Mats Neymark. *Matematisk analys, en variabel.* 2nd ed. Solna, Stockholm: Liber, Aug. 2011. ISBN: 9789147100231.

[20] Tushaar Gangavarapu, Himadri Pal, Pratyush Prakash, Suraj Hegde, and V Geetha. “Parallel openmp and cuda implementations of the n-body problem”. In: *International Conference on Computational Science and Its Applications.* Springer. 2019, pp. 193–208.

[21] *GASPI: Global Address Space Programming Interface - Specification of a PGAS API for communication.* Tech. rep. 17.1. Fraunhofer ITWM, Feb. 2017.

[22] Leslie Greengard. “The numerical solution of the n-body problem”. In: *Computers in physics* 4.2 (1990), pp. 142–152.

[23] Daniel Grünewald. “BQCD with GPI: A case study”. In: *2012 International Conference on High Performance Computing & Simulation (HPCS).* IEEE. 2012, pp. 388–394.

[24] Daniel Grünewald and Christian Simmendinger. “The GASPI API specification and its implementation GPI 2.0”. In: *7th International Conference on PGAS Programming Models.* Vol. 243. 2013, p. 52.

[25] Tsuyoshi Hamada, Keigo Nitadori, Khaled Benkrid, Yousuke Ohno, Gentaro Morimoto, Tomonari Masada, Yuichiro Shibata, Kiyoshi Oguri, and Makoto Taiji. “A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs–towards cost effective, high performance N-body simulation”. In: *Computer science-research and development* 24.1-2 (2009), pp. 21–31.

[26] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. “Parallel programmer productivity: A case study of novice parallel programmers”. In: *SC’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing.* IEEE. 2005, pp. 35–35.

[27] Ivan Hristov, Radoslava Hristova, Stefka Dimova, P Armyanov, N Shegunov, I Puzynin, T Puzynina, Zarif Sharipov, and Zafar Tukhliev. “Parallelizing multiple precision Taylor series method for integrating the Lorenz system”. In: *arXiv e-prints* (2020).

[28] Amir Kamil and Katherine Yelick. “Hierarchical computation in the SPMD programming model”. In: *International Workshop on Languages and Compilers for Parallel Computing.* Springer. 2013, pp. 3–19.

[29] Hsiang-Tsung Kung. “Why systolic architectures?” In: *Computer* 15.01 (1982), pp. 37–46.

[30] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. “A Large-Scale Study of MPI Usage in Open-Source HPC Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC ’19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: `10.1145/3295500.3356176`.

[31] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. “Basic linear algebra subprograms for Fortran usage”. In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[32] Charles E. Leiserson. “Fat-trees: Universal networks for hardware-efficient supercomputing”. In: *IEEE Transactions on Computers* C-34.10 (1985), pp. 892–901. DOI: `10.1109/TC.1985.6312192`.

[33] Mario Leyton and José M Piquer. “Skandium: Multi-core programming with algorithmic skeletons”. In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing.* IEEE. 2010, pp. 289–296.

[34] Benoit B Mandelbrot. “Fractal aspects of the iteration of z$\rightarrow$ Λz (1-z) for complex Λ and z”. In: *Annals of the New York Academy of Sciences* 357.1 (1980), pp. 249–259.

[35] Peter Pacheco and Matthew Malensek. *An Introduction to Parallel Programming.* 2nd ed. Cambridge, Massachusetts: Morgan Kaufmann, 2020, p. 49. ISBN: 978-0-12-804605-0.

[36] Tiberiu Rotaru, Mirko Rahn, and Franz-Josef Pfreundt. “MapReduce in GPI-Space”. In: *Euro-Par 2013: Parallel Processing Workshops.* Ed. by Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 43–52.

[37] Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. “PGAS implementation of SpMVM and LBM using GPI”. In: *7th International Conference on PGAS Programming Models.* University of Edinburgh, 2013, pp. 172–184.

[38] J. L. Sobral and A. J. Proenca. “Enabling JaSkel skeletons for clusters and computational Grids”. In: *2007 IEEE International Conference on Cluster Computing.* 2007, pp. 365–371. DOI: `10.1109/CLUSTR.2007.4629251`.

[39] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[40] Xian-He Sun and Lionel M Ni. “Another view on parallel speedup”. In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing.* 1990, pp. 324–333.

[41] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: `10.1112/plms/s2-42.1.230`.

[42] CR Wan and David J Evans. “Nineteen ways of systolic matrix multiplication”. In: *International journal of computer mathematics* 68.1-2 (1998), pp. 39–69.

[43] Michael S Warren and John K Salmon. “A parallel hashed oct-tree n-body algorithm”. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing.* 1993, pp. 12–21.

[44] Xiong Zheng and Vijay Garg. "An optimal vector clock algorithm for multithreaded systems". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 2188–2194.