Using Task Parallelism for Distributed Parallel Skeleton Programming

- Implementing a StarPU Back-End to SkePU 2

Henrik Henriksson

Supervisor: August Ernstsson
Examiner: Christoph Kessler
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsmann vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är skadligt för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Henrik Henriksson
Abstract

We extended the parallel skeleton programming framework SkePU 2 with a new back-end utilizing StarPU, a task programming framework for hybrid and distributed architectures. The aim was to allow SkePU to run on distributed clusters, using MPI through StarPU.

The implemented back-end distributes data and work across participating ranks. While we did not implement the full SkePU API, the “Map” and “Reduce1D” skeletons were successfully implemented. During the implementation, we discovered some differences in API design between SkePU and StarPU. We combine the type-safe templates used in the SkePU API with the C-style void*-heavy API of StarPU. This requires the implementation to use more complex templates than normally desired. While we could preserve most of the SkePU 2 API when moving to a distributed memory situation, some parts had to change. In particular, we needed to change the semantics of SkePU 2 containers with regards to iterators and random access.

We benchmarked the performance of the implemented back-end against an MPI+OpenMP reference implementation on two problems, n-body and a simple reduction. While the n-body problem demonstrates promising scaling properties, reductions do not scale well to larger number of ranks. A performance comparison against the MPI+OpenMP reference implementation reveals that, aside from the higher communication overhead, there may also be some overhead in the work performed between communications, potentially performing at below 60 – 70% of the reference. In most cases, the new back-end to SkePU exhibits significantly lower performance than the reference.

Extending the implemented solution to cover the full API and improving performance could provide a high level interface to distributed programming for application programmers. Indeed, subsequent developments of SkePU 3 extend and improve our StarPU back-end.
Preface

I have always had an inclination to make computers suffer. One of my first programming projects was computing primes, at an age far too young to really understand the footguns presented by K&R. Naively at first, but quickly becoming more sievely. Studying computer engineering and venturing into systems administration only made my desire to transform electricity into heat stronger, at times producing something useful or beautiful as a byproduct. My hope is that this work can help others do the same in the future.

Henrik Henriksson
Acknowledgments

I would like to extend my gratitude to all the people who have helped and encouraged me during this project. First of all, I’d like to thank my supervisor August Ernstsson and examiner Christoph Kessler for helping me through the process, giving me feedback, and, of course, the opportunity to work on SkePU. Johan Ahlqvist managed to fix some nasty bugs in my code, for which I am grateful. The StarPU team was helpful during the development of the back-end, providing valuable feedback. Additionally, I would like to extend my thanks to everyone else who have worked on SkePU before and after me.

While still on the academic side of things, I’d like to extend my thanks to Kristian Sandahl, for the opportunity to teach in his courses, through which I have extended my set of skills considerably.

During the project, I’ve burnt quite a few CPU-hours at the compute cluster hosted by the academic computer club Lysator. With my friends there, I’ve also burnt many, many, many hours over pizza, discussing and learning, everything from high performance computing to high performance data destruction.

I’d also like to thank my current colleagues at the National Supercomputer Centre at Linköping University. In particular, Filip Polbratt provided valuable technical insights and sanity checks, close to the hardware, while Johan Raber helped create a run-time environment for final benchmarks. I may also have used a handful of CPU-hours on idle nodes.

Finally, I would like to thank Sophie Ziske, who, apart from being a wonderful partner, got me to finally get this project done.
Contents

1 Introduction

1.1 Motivation .......................................................... 1
1.2 Research Questions .............................................. 2
1.3 Scope and Delimitations ......................................... 2
1.4 Method .............................................................. 2

2 SkePU, StarPU and Parallel Programming

2.1 SkePU ................................................................. 3
2.2 Data Parallel Computation Patterns ........................... 5
2.3 StarPU ................................................................. 8
2.4 Tools for Parallel Programming ................................. 9
2.5 Message Passing Interface ....................................... 11
2.6 Hardware Architecture for Parallel Computation ........... 13

3 Related Works

3.1 Skeleton Programming Taxonomy ............................... 15
3.2 Quaff ................................................................. 16
3.3 The Muenster Skeleton Library (Muesli) ....................... 16
3.4 Chapel ................................................................. 17
3.5 Other Work on SkePU ............................................. 17

4 Method

4.1 Implementation ..................................................... 18
4.2 Performance Characteristics .................................... 18

5 Design and Implementation

5.1 Method .............................................................. 21
5.2 Overreaching Design Decisions ................................. 22
5.3 Memory Management ............................................ 22
5.4 Meta Skeleton Programming .................................... 28
5.5 Skeleton Implementations ....................................... 35

6 Performance Evaluation

6.1 Method .............................................................. 38
6.2 Benchmarking Results ............................................ 41
## Contents

### 7  Discussion
- 7.1 Design and Implementation ................................................. 46
- 7.2 MPI ........................................................................ 46
- 7.3 Meta Skeleton Programming .................................................. 47
- 7.4 Unification of Data Structures ............................................... 47
- 7.5 Implementation Complexity ..................................................... 48
- 7.6 Flexibility and Ease of Use ...................................................... 49
- 7.7 Performance .................................................................. 49
- 7.8 Ethical Considerations .......................................................... 51
- 7.9 Future Work .................................................................. 51

### 8  Conclusions
- 8.1 Implementing a StarPU back-end to SkePU .............................. 53
- 8.2 Changes to the SkePU API ....................................................... 53
- 8.3 Overall Performance Characteristics ....................................... 54
- 8.4 Final Remarks .................................................................. 54

### Bibliography

#### A  Benchmark Code

#### B  Raw Benchmark Measurements
Chapter 1

Introduction

Moore’s Law is currently going through the final stages of grief [33, 34], and will soon be dead, as documented in academic research, popular science and other media. We are confronting a fundamental limitation, imposed by the laws of physics and the constraints of reality itself. To progress in our endeavours to compute, we must embrace parallelization. However, developing parallel software usually comes with a large hit to developer productivity. This issue grows even larger when we add accelerators and distributed systems to the mix.

With this thesis, we aim to alleviate this situation. Developers need better tools for working with parallelism. Developers need tools on a higher level of abstraction. And finally, developers need tools that allow portability between different systems and architectures.

1.1 Motivation

One main motivation behind parallel programming as a field of study is the hard limits on single-threaded performance that physics constrain us to. The case for parallelization only grows stronger when adding any criteria of timeliness. Certain computations would simply be impossible to do on any existing single threaded hardware in a reasonable wall time. In recent years, the use of Graphics Processing Units (GPUs) for parallel computations has grown tremendously, as have the use of distributed computing clusters.

When the resources of a single system are insufficient, one can attempt to resolve the situation by adding more hardware. Systems with powerful CPUs and multiple GPUs are a common occurrence, allowing for a plenty of computational power. However, some tasks require access to even more resources, necessitating the need for distributed systems. Contemporary HPC clusters can contain thousands of such nodes [43].

Programming these distributed systems is hard, especially when it involves GPUs or other special hardware. The software need to support multiple different system architectures in order to be portable between systems. Manufacturers, interfaces and even programming languages can differ between different systems, requiring much time and skill from software developers.

We have decided to use the “academic we” when we refer to the singular author of this thesis.
1.1.1 Aims

SkePU \cite{9} is a framework for skeleton programming, a method for enabling programmers to access parallel hardware at a high level of abstraction. SkePU was, when this project began, limited to single multi-GPU nodes. Scaling out SkePU across clusters instead of single nodes required a new back-end. We elected to use StarPU \cite{4}, a library for task parallelism on distributed architectures, to extend SkePU with capabilities for distributed compute. Chapter 2 describes both SkePU and StarPU.

1.2 Research Questions

RQ1. Using the SkePU 2 API and the StarPU framework, can we support distributed execution of parallel skeletons across a cluster?

RQ2. What changes are needed in the SkePU API to make this viable?

RQ3. What are the overall performance characteristics of such a system?

1.3 Scope and Delimitations

Due to time constraints, we did not create a fully featured StarPU-based back-end to SkePU. Instead, the aim is to implement enough to demonstrate viability and performance characteristics. Using GPUs is also left out of scope, even though both SkePU and StarPU are able to handle such hardware. Exploring if it is architecturally possible to extend the resulting implementation into the realm of GPU-compute is, however, left within the scope of this project.

1.4 Method

Research questions one and two both relate to how models, assumptions and APIs within and between SkePU, StarPU and MPI interact when put together. We explored these two research questions by implementing a solution, documenting the result. The method for this is further described in Section 5.1.

Research question three relates to the performance characteristics for such a system. The method for this is further described in Section 6.1.
Chapter 2

SkePU, StarPU and Parallel Programming

This chapter aims to introduce and describe SkePU and StarPU, the work we build this thesis on. Additionally, we present some background on the software and hardware used for parallel programming.

In this thesis, the primary focus is to continue the work on SkePU, a parallel programming skeleton framework. Enmyren and Kessler presented SkePU in 2010 \[9\], which has since been further developed into SkePU 2, published in 2017 \[13\].

The functionality developed during this project, as presented in this report, was available late 2019, while this thesis was presented early 2024. As such, some of the work we present in this thesis have already been used in work surrounding SkePU. The implementation we present within this report was merged into SkePU 2 before the release of SkePU 3. To give a more complete picture of work on SkePU up until writing, we also present related works published chronologically after our work in implementing the StarPU back-end for SkePU 2.

2.1 SkePU

SkePU \[13\] provides skeletons, abstractions for commonly used patterns in parallel programming. The user provides a user function that is used within a skeleton. This is similar to a function argument given to a higher order function\(^1\). This user function is then compiled into several different variants for use with the different backends of SkePU. At the time of writing, these are CUDA \[6\], OpenCL \[26\] and OpenMP \[17\]. To ensure compatibility, the user function must be written in a subset of C\texttt{99}. During compile time, these are then compiled together with the skeleton implementations, thereby emitting parallel code for these different architectures.

2.1.1 Previous Work on MPI in SkePU

Majeed, Dastgeer and Kessler presented an extension of SkePU 1, to support MPI, in 2013 \[21\]. The paper presents Cluster-SkePU, supporting MPI-distributed operations over the framework’s vector container with good performance in several workloads. However, there are some improvements to be made. First of all, the implementation has not been ported to SkePU 2, and is thus not currently available for use. Majeed et al. conclude that “To the best of our knowledge, this is the first skeleton library implementation for GPU clusters”, pointing to the need of more work in this area.

\(^1\)A higher order function is a function that takes a function as an argument or returns a function. For skeleton programming, the method for this might not be just a function call. The skeleton programming framework in question may for example apply transformations on the function at compile time, or otherwise go outside of what higher order functions usually provide within a certain language.
Cluster-SkePU distributes memory among participating ranks by utilizing collective MPI calls, such as \texttt{MPI Scatter}. While this approach was shown to work well \cite{21}, it has its limitations. The collective MPI calls are used in a way that sends an equal amount of data to each participating MPI-process, essentially statically distributing memory among the nodes. This works, assuming the workload is proportional to the memory usage and all MPI-ranks are running on equal resources. Even simple problems, such as rendering a Mandelbrot set, has enough variance in workload per memory unit that much time will be wasted waiting for other nodes. A more dynamic system for memory communication and work scheduling could enable higher efficiency in these cases.

### 2.1.2 SkePU 3

Ernstsson, Ahlqvist, Zouzoula and Kessler presented SkePU 3 \cite{12}, and Ernstsson also goes into the details of the SkePU 3 API in his PhD thesis \cite{11}. SkePU 3 introduces plenty of new features, including new skeletons, such as \texttt{MapPairs} and new collections, such as tensors. This version also introduces multi-variant user functions and a StarPU-MPI back-end \cite{3}. Removing some support for the array subscript operator ([ ] ) from the API allows for different memory consistency models in data containers, this was a notable breaking change from the SkePU 2 API. The performance of the StarPU-MPI back-end to SkePU 3 was also benchmarked, showing a high overhead as compared to single node SkePU 3 without StarPU.

While the implementation we present within this thesis was a part of SkePU 3 at time of writing, there have been developments on distributed execution of SkePU since. Here we present some related works within the context of SkePU 3.

**EXA2PRO** The work on SkePU 3 was a part of the project to develop the EXA2PRO framework \cite{28}, aimed at improving developer productivity for applications that target heterogeneous systems. The benchmarks performed indicate that some SkePU 3 code using StarPU-MPI scaled poorly due to the excessive coherence communication volume required in a Brain Modeling application using the Map skeleton.

**Performance and Usability of SkePU 3** Tedhamre, in his Master’s thesis \cite{40}, evaluates the performance and understandability of SkePU 3 code, but does not venture into the realm of distributed programming. With regards to performance, Tedhamre concludes that while highly optimized parallel code will beat SkePU, the framework can compete in cases with less optimized code.

**MPI+OpenMP Back-End** Zeijlon, in his Master’s thesis \cite{45}, presents an MPI+OpenMP back-end to SkePU without StarPU and compares this to the StarPU back-end. The thesis also explore the effects of crossing NUMA-boundaries, especially with regards to OpenMP. Benchmarks of the different back-ends show the NUMA-aware MPI+OpenMP implementation taking the lead. The StarPU-MPI back-end has a large overhead, both in communication and compute.

**Global Address Space Programming Interface Back-End** Almqvist, in his Master’s thesis \cite{1}, presents a GPI back-end to SkePU, aiming to run on HPC clusters. This implementation can rely extensively on one-sided communication operations. Benchmarks show that the performance of the StarPU and GPI back-ends have significantly differing characteristics. Either one can end up with better performance than the other, depending on the benchmark scenario. The experiments used large problem sizes, so both back-ends show reasonable scaling properties. However, the experiments show a performance loss for the StarPU back-end when moving from one to two nodes.

\footnote{Cartesian mapping of 1D vectors into a 2D matrix.}

\footnote{Based on the implementation described within this report.}

\footnote{An implementation of the Global Address Space Programming Interface (GASPI) \cite{16}.}
2.2. DATA PARALLEL COMPUTATION PATTERNS

Nomenclature As we are now dealing with plenty of SkePU implementations, versions and back-ends, it may be good to clarify what we refer to in the rest of this report. We will be using “SkePU 2 StarPU-MPI” in order to refer to the implementation created as part of this project and distinguish it from other work done on the StarPU back-end during later SkePU 3 development.

2.2 Data Parallel Computation Patterns

There are several computation patterns in data parallel programming. These patterns are what skeleton programming aims to create abstractions for [15]. At the implementation level, these abstractions often end up utilizing higher order functions. In this section we describe the patterns used in SkePU 2, where they are applied on arrays or matrices. In more general terms, variations of the patterns can apply to other data structures, such as trees or graphs.

Each data parallel pattern applies some function on data. Depending on the pattern, the function may be required to have certain mathematical properties in order for the pattern to be parallelizable. We describe the patterns as SkePU 2 [13] implements them, but a bit simplified.

2.2.1 Map

A map applies a function \( f \) on each element in a data structure, see Algorithm 2.1. This is a very common pattern, even in serial code. The pattern can be applied in-place, as a structure preserving mutation of the data. Alternatively, the result of the operation can be stored in a separate data container, preserving the original data. The pattern is embarrassingly parallel, each element can be computed separately, with no dependencies between elements of the data structure. Listing 2.1 shows how two vectors can be added using this pattern in SkePU 2.

**Algorithm 2.1** Serial pseudocode for the map pattern.

```
1: for all index \( i \) ∈ input do
2: output[\( i \)] ← \( f(\text{input}[\text{index}]) \)
3: end for
```

Listing 2.1: Example usage of the map pattern in SkePU 2.

```
1: double add(const double a, const double b) { return a + b; }
2: skepu2::Vector<double> vector_add(skepu2::Vector<double> &a, skepu2::Vector<double> &b) {
3: auto vector_adder = skepu2::Map<2>(add);
4: skepu2::Vector<double> result(a.size());
5: vector_adder(result, a, b);
6: return result;
7: }
```
2.2.2 Reduce

The reduce pattern performs a reduction over data, as shown in Algorithm 2.2. A common example is calculating the sum of values in an array. The function in use must be associative, but commutativity is not required. Using the operator ⊕ instead of a function is a common notation. Listing 2.2 shows how the pattern can be used in SkePU 2 to sum all elements of an array.

The pattern is parallelizable by utilizing divide-and-conquer strategies. The work is split up recursively into parts, where each part can be calculated independently, and then recursively reduced to a single value.

Algorithm 2.2 Serial pseudocode for the reduce pattern.
1: result ← initial value
2: for all element ∈ input do
3: result ← result ⊕ element
4: end for

Listing 2.2: Example usage of the reduce pattern in SkePU 2.

```plaintext
1. double add(const double a, const double b) { return a + b; }
2. double sum(skepu2::Vector<double> &v) {
3.   auto reducer = skepu2::Reduce(add);
4.   return reducer(v);
5. }
```

2.2.3 Scan

The scan pattern creates a prefix sum from its input. The pattern is similar to the reduce pattern, but instead of resulting in a single value, all intermediate values are saved in the result. This is shown in Algorithm 2.3. As with the reduce pattern, the operator ⊕ must be associative. Listing 2.3 demonstrates how the pattern can be used in SkePU 2 to calculate the prefix sum of an array.

Algorithm 2.3 Serial pseudocode for the scan pattern.
1: for all idx ∈ input do
2: output[idx] ← output[idx − 1] ⊕ input[idx]
3: end for

Listing 2.3: Example usage of the scan pattern in SkePU 2.

```plaintext
1. double add(const double a, const double b) { return a + b; }
2. skepu2::Vector<double> cumulative_sum(skepu2::Vector<double> &a) {
3.   auto cumulative_summer = skepu2::Scan(add);
4.   auto result = skepu2::Scan(a);
5.   cumulative_summer(result, a);
6.   return result;
7. }
```
2.2.4 MapReduce

The mapreduce pattern combines the map and reduce patterns into a single pattern, as shown in Algorithm 2.4. This pattern uses two functions. We use one function applied on each value, \( f \), and an operator \( \oplus \) for the reduction step. Applying the map and reduce patterns after each other will yield similar semantics. However, by combining these two into the mapreduce pattern, the result can be computed without modifying the original input data or creating intermediate copies of the input data.

Algorithm 2.4 Serial pseudocode for the mapreduce pattern.

1: result ← initial value
2: for all element ∈ input do
3: result ← result \( \oplus f(\text{element}) \)
4: end for

Listing 2.4: Example usage of the mapreduce pattern in SkePU 2.

double add (const double a, const double b) { return a + b; }
double mult (const double a, const double b) { return a * b; }
double dot (skepu2::Vector<double> &a, skepu2::Vector<double> &b) {
    auto dotter = skepu2::MapReduce(2)(mult, add);
    return dotter(a, b);
}

2.2.5 MapOverlap (Stencil operations)

The mapoverlap pattern is also known as stencil operations. This is similar to the map pattern, but includes a read-only area around a read-only element, and not just the element. In the one dimensional case, we specify the size of this area using the radius \( r \). This is shown in Algorithm 2.5. In the case of higher dimensions, the behaviour is similar. For each element, the output is a function of the element itself and a limited region around the element. Listing 2.5 shows how the pattern can be used to compute a moving average with SkePU. It should be noted that the SkePU 2 API for the MapOverlap skeleton is well developed, supporting both 1D and 2D operations, along with options on how to handle the edges of the data structure [13]. SkePU 3 also extends this API further, with increased ergonomics [12].

Algorithm 2.5 Serial pseudocode for the mapoverlap pattern.

1: for all index \( i \) ∈ input do
2: output[\( i \)] ← \( f(\text{input}[\( i - r \)], \ldots, \text{input}[\( i - 1 \], \text{input}[\( i \)], \text{input}[\( i + 1 \)], \ldots, \text{input}[\( i + r \)]) \)
3: end for

Listing 2.5: Example demonstrating stencil operations in SkePU 2.

double avg (int overlap, size_t stride, const double * input) {
    double sum = 0;
    for (int i = -overlap; i <= overlap; ++i) {
        sum += input[i*stride];
    }
    const double count = 2*overlap + 1;
    return sum / count;
}
skepu2::Vector<double> running_average (skepu2::Vector<double> &a) {
    auto convolver = skepu2::MapOverlap(avg);
    convolver.setOverlap(2);
    skepu2::Vector<double> result(a.size());
    convolver(result, a);
    return result;
2.3 StarPU

StarPU is a library for task-parallel programming in C with support for scheduling and executing tasks on CPUs and GPUs. The StarPU handbook is a reference for programmers using the library, and also what the background on StarPU presented here is based on. StarPU also has support for managing data transfers over MPI. StarPU allows programmers to schedule multivariant tasks on blocks of data, forming a task graph, which is asynchronously executed on heterogeneous hardware, scheduled dynamically during runtime. Using multiple implementations of each single task, combined into codelets, it is possible to let a single task run on any available hardware, be it GPU or CPU. By running multiple tasks simultaneously, more available hardware can be used. StarPU will also handle the data transfer between different hardware devices, to ensure the computation defined by its task graph can proceed in a coherent manner. The MPI-capabilities of StarPU allow programmers to use the memory management and memory transfer abilities of StarPU while creating distributed programs. MPI-operations are implicitly or explicitly inserted into the task graph, allowing coherent operation across a distributed system.

2.3.1 StarPU’s Task and Data Management Models

StarPU provides handles to data. The user can register data to a handle and let StarPU manage that data. The data can be a single value, a vector or one of several other data-types. Users can even define their own. StarPU will handle memory transfers between different devices, mostly transparent to the user, for example from host to device memory or using MPI between two cluster nodes. StarPU injects such transfers into the task graph where needed. Handles are passed on to the code running the actual computations, where they are exposed as buffers on the targeted device. Listing 2.6 shows an example where we register a few such handles.

```
const size_t buf_size = 128;

starpu_data_handle_t a_handle;
starpu_data_handle_t b_handle;
starpu_data_handle_t sum_handle;

double * a_buf = starpu_malloc(buf_size*sizeof(double));
double * b_buf = starpu_malloc(buf_size*sizeof(double));
double * sum_buf = starpu_malloc(buf_size*sizeof(double));

// Register handles
starpu_vector_data_register(&a_handle, 0, (uintptr_t)a_buf, N, sizeof(double));
starpu_vector_data_register(&b_handle, 0, (uintptr_t)b_buf, N, sizeof(double));
starpu_vector_data_register(&sum_handle, 0, (uintptr_t)sum_buf, N, sizeof(double));

// Use handles
// [...]

// Clean up
starpu_data_unregister(a_handle); starpu_free(a_buf);
starpu_data_unregister(b_handle); starpu_free(b_buf);
starpu_data_unregister(sum_handle); starpu_free(sum_handle);
```

StarPU uses the concept of a codelet, a specification of how a certain task should be performed. Each codelet can contain several different implementations, for example an OpenMP CPU-implementation, a CUDA implementation and an OpenCL implementation. The user also defines some metadata, for example the number data buffers and their access modes. Unlike skeletons or higher order functions, the codelet abstraction is just a collection of metadata and function pointers. Listing 2.7 shows a simple such codelet, capable of adding two vectors together.

---

5In this case, a directed acyclic graph with computational tasks as nodes and data dependencies as edges.
Listing 2.7: Example of a simple StarPU codelet.

```c
void cpu_sum_func(void * bufs[], void * cl_arg) {
    struct starpu_vector_interface_s * buf_handle_a = bufs[0];
    struct starpu_vector_interface_s * buf_handle_b = bufs[1];
    struct starpu_vector_interface_s * buf_handle_out = bufs[2];
    size_t size = STARPU_VECTOR_GET_NX(buf_handle_a);
    double *a = (double*)STARPU_VECTOR_GET_PTR(buf_handle_a);
    double *b = (double*)STARPU_VECTOR_GET_PTR(buf_handle_b);
    double * out = (double*)STARPU_VECTOR_GET_PTR(buf_handle_out);

    for (int i = 0; i < size; ++i) {
        out[i] = a[i] + b[i];
    }
}

starpu_codelet vec_add_cl = {
    .cpu_func = cpu_sum_func,
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W }
};
```

Users can schedule tasks, asking StarPU to execute one of the implementations in a codelet on data referred to by provided handles. The StarPU runtime system is then free to schedule the execution, taking data dependencies and usable implementations into consideration. Tasks are executed asynchronously, the user can continue inserting tasks during execution. After all tasks are inserted, the user simply waits for the completion of the task graph execution. Listing 2.8 shows this in action, adding together two vectors.

Listing 2.8: Example of how to submit a task in StarPU.

```c
// Submit the task, will be executed in the background
starpu_task_insert(& vec_add_cl,
    STARPU_R, a_handle,
    STARPU_R, b_handle,
    STARPU_W, sum_handle);

// Submit other tasks as needed, StarPU will manage
// data dependencies and data movement
// [...]  

// Wait for all submitted tasks to complete
starpu_task_wait_for_all();
```

Access to data handles can be requested in different modes, signifying how the data will be used. Handles can be requested in read, write and read-write modes, affecting how tasks are scheduled in the task graph. The access mode is also used to ensure coherency between uses of the same handle by ensuring data transfers and synchronizations are inserted into the task graph at appropriate locations.

### 2.4 Tools for Parallel Programming

Many applications currently use CUDA [6], OpenCL [26], MPI [25] or OpenMP [17] as their parallelization framework. All these are heavily used in industry. CUDA is proprietary and only available on GPUs produced by Nvidia. OpenCL is an open standard and is not limited to a single manufacturer or platform and is instead available on GPUs, CPUs and other hardware accelerators. OpenMP primarily provides parallelization on multicore CPUs, but has recently gained support for offloading computations to accelerator hardware such as GPUs. However, neither solution handles distribution and the programmer must explicitly write all communication if they are to be used in a cluster environment, often using MPI. These are often used at a fairly low level of abstraction and requires much knowledge about the underlying hardware to provide optimal performance.

Of particular relevance to us is what SkePU and StarPU can leverage. This includes OpenMP, OpenCL and CUDA.
2.4.1 OpenMP

OpenMP [17] is an API primarily designed to facilitate parallel programming in shared-memory environments. In C and C++, OpenMP is implemented through compiler pragmas, which the compiler will use to parallelize code across multiple threads. The semantics of the program with and without such pragmas often remain similar or identical. OpenMP offers a range of parallelization constructs, encompassing everything from reductions and parallelization of embarrassingly parallel loops to task-based parallelism.

2.4.2 OpenCL

OpenCL [26], [39] is a framework for parallelization on heterogeneous platforms, with the aim to support multiple platforms with the same framework. Various CPUs, GPUs, DSPs and FPGAs from multiple vendors support OpenCL. OpenCL defines an API for a C and C++ library that can be used to compile, load and run kernels on compute devices. The kernels are, similar to OpenGL, written in a C-like language, which can then be loaded using the OpenCL library API.

2.4.3 CUDA

CUDA [6], [31] is a commercial, proprietary, parallel computing platform and programming model created by Nvidia for GPGPU-purposes. CUDA uses a separate compiler, providing additional functionality on top of C++. As with OpenCL, CUDA also has a concept of kernels, but in CUDA these are written in standard C++ (with some limitations) and then compiled to run on GPUs. Kernels can be written inline with other code, and code can be shared between kernels and CPU-code as both use the same language.

2.4.4 Other platforms and libraries

Among others, OneAPI [24] and SYCL [2] have emerged as alternatives to CUDA and OpenCL, opening up the vendor agnostic possibilities. Additionally, domain specific libraries such as PyTorch [30] provide parallelization at a very high level of abstraction within the domain of machine learning.

---

6Starting with OpenMP 5, there is also support for using various accelerators, such as GPUs. The OpenMP 5 specification was released just a few months before we started working on this project. As such we did not consider OpenMP 5 for use in this project.
2.5 Message Passing Interface

Message Passing Interface (MPI) [25] is a standardized API for message passing in parallel computing. The MPI standard specifies an API, which is available through multiple different implementations. We will be using OpenMPI [25], but many other implementations exist. The standard specifies an API and semantics for the API. As such, implementations can optimize for different hardware platforms. We base the background in this section on the documentation for OpenMPI [25], a popular open source implementation of MPI.

2.5.1 Concepts

When an MPI program starts, the same program is started concurrently on one or more processes in a cluster, often on separate compute nodes. Each such process is assigned a rank, a number uniquely and sequentially identifying it. The word “rank” is also often used to refer to MPI-processes. The running ranks share no memory[7] but the MPI implementation provides message passing according to the semantics in the MPI specifications. The underlying mechanism for message passing is implemented by MPI-libraries for a wide variety of hardware and interconnection mechanisms. For multiple ranks on a single node, shared memory on the node can be used as a message passing mechanism. In HPC-contexts, RDMA over Infiniband is often used for communication between nodes.

At a high level, MPI encompasses various sets of functionality, including for send/receive-style message passing, collective operations and an API for one-sided communication.

2.5.2 Send/Receive

MPI_Send and MPI_Recv is the most straightforward way to send and receive data between two ranks. The receiver runs MPI_Recv and the sender runs MPI_Send. Both ranks provide the ranks of each other and a buffer where sent data should end up or come from. With MPI_Send/MPI_Recv, the operation is blocking for both ranks participating, but non-blocking variants (MPI_Isend/MPI_Recv) are provided. It is important to note that this manner of communication involves exactly two ranks per message, one sender and one receiver.

2.5.3 Collective Operations

To allow for more than two nodes to communicate efficiently, MPI also supports collective operations. These redistribute data across all participating ranks.

• MPI_Bcast distributes data from one rank to all others so all ranks have the same data,
• MPI_Scatter distributes data from one rank to all other ranks so all n ranks get \( \frac{1}{n} \) parts of the data,
• MPI_Gather distributes data from n ranks to one rank, the reverse of MPI_Scatter,
• MPI_Allgather is semantically similar to MPI_Gather followed by MPI_Bcast but allows for performance optimizations.

MPI also allow for some amount of collective computations, such as

• MPI_Reduce for reductions on data scattered across ranks,
• MPI_Scan for prefix reductions on data scattered across ranks.

All of the collective operations mentioned so far have variations, for example non-blocking variants.

[7] The programmer is free to sidestep the MPI-library, and the MPI implementation is free to utilize shared memory for performance reasons. Semantically, MPI is a message passing API, but shared memory can be used and abused by both application programmers and MPI-implementation vendors.
2.5.4 One sided communication

MPI_Put and MPI_Get are one-sided communication primitives. Unlike MPI_Send and MPI_Recv, the other participant does not need to participate in the communication, apart from setup and synchronization. These functions allow for, conceptually and practically, reading and/or writing from/to memory on another rank. To control this, the programmer opens up “windows” (MPI_Win) on ranks to allow for access control and more granular synchronization options.

2.5.5 Control Flow

As most operations within MPI need to be invoked on all participating ranks simultaneously, the control flow across all ranks must be coherent. For example, if an MPI_Scatter operation is to be performed, all participating ranks must invoke that specific function with the correct arguments. If any rank does not participate, all other ranks will block on the operation, waiting for the last participant. Control flow in general may diverge, for example a single rank can be responsible for writing results to disk. However, all participating ranks must perform MPI-operations together, in an order that avoids deadlocks.

2.5.6 MPI Communication in StarPU

There are two different variants in how to use MPI with StarPU. We can either have one central rank dispatching tasks to others, or have each rank build identical task graphs, thus avoiding a central dispatcher. The latter model is what we use, and describe here. We base what we present in this section on StarPU’s online documentation [38].

StarPU has the ability to automate much of the data transfer between different nodes. This is done by registering handles as being available over MPI. Each handle is registered with an owner, that has allocated memory for the actual data. Other nodes can register the handle with or without actually allocating any memory up front.

When scheduling tasks, StarPU implicitly inserts the appropriate MPI send and receive operations in the task graph. Since the data dependencies between tasks are known to all ranks, these operations can be inserted such that the appropriate data is made available to each task.

One remaining problem is deciding which rank should execute a particular task. By default, this is chosen based on the input handles. If the task only has one writable handle, the owner of the writable handle will execute the task. In other cases, the user must select where the task should be executed.

Compared to the examples presented in Listings 2.6, 2.7 and 2.8, there are surprisingly few changes needed to extend the code to run on multiple ranks using MPI, as long as we stay within the StarPU API. Before operating directly on any StarPU-data, outside of a StarPU codelet, care must be taken to ensure proper fencing and data transfers. In general, some functionality has MPI-specific implementations that replace the “local” variants. For example, we need to use starpu_mpi_insert_task instead of starpu_task_insert.

---

8 Not all ranks need to participate, but participating ranks must know the other participants. For the purpose of this text, we can assume MPI_COMM_WORLD everywhere.

9 It should be noted that under some circumstances it is possible to register a handle without any memory allocation performed outside of the StarPU runtime system, but for our purposes we can assume each handle to have a registered owner with allocated memory.
Explicit Data Transfers  In some cases there are benefits to doing explicit data transfers without scheduling tasks. The overhead can be lower, with regards to both computation and code complexity. Explicit transfers are also useful for getting data in and out of the StarPU runtime system.

StarPU provides abstractions on top of some of MPI’s functionality, allowing programmers to use handles instead of pointers when interacting with MPI. There is also communication functionality on a higher level of abstraction than raw MPI. The high level API will insert the transfer into the task graph, ensuring proper synchronization automatically.

Implicit Data Transfers  StarPU can implicitly insert data transfer tasks into the task graph. This requires users to create the task graph on all ranks. The created task graph must be coherent across all participating ranks. This implies that relevant control flow and program execution must be coherent as well, similar to how MPI works.

Caching  StarPU allows caching handles on MPI-ranks other than the owner. As StarPU knows how codelets will use each handle, caching and invalidation can be managed through the task graph. For example, STARPU-R in Listing 2.7 indicates that the data referred to by a_handle won’t be modified.

2.6 Hardware Architecture for Parallel Computation

In recent years, GPUs have joined the more general purpose CPUs. Using GPUs can yield a far higher degree of parallelism than previously possible. Both GPUs and CPUs are still, mostly, built on the same fundamental components, some kind of processing units and some kind of memory connected together, but do have some fundamental differences. In particular, GPUs tend to focus more on memory bandwidth for well formed reads, while CPUs usually provide better memory latency for random access.

Even though the implementation of SkePU 2 StarPU-MPI that we present in this report targets general purpose CPUs, we aim to present some background on GPU compute to give context and aid in discussions. In addition to what is relevant for shared-memory systems, we also present some background on the interconnection networks used in compute clusters.

2.6.1 GPGPU

Early General-purpose computing on GPUs was achieved by exploiting the APIs used for rendering graphics, for example by using DirectX 9. Nowadays, we have textbooks, such as Attack in Packs, on which we base this section on GPGPU.

To support real-time graphics, numerous mathematical operations were hardware accelerated, resulting in high raw performance. However, these graphics pipelines were not flexible, especially not for general purpose usage. Despite the limitations, researchers employed creative approaches to successfully crunch numbers. Nvidia and other vendors now support general purpose compute on GPUs via APIs such as CUDA or OpenCL.

GPUs often have small caches, if they have any cache at all. The designs often prioritize bandwidth at the expense of access granularity and latency. Accessing a large contiguous block of memory is efficient on GPUs, but accessing a small portion of memory, like a single pointer, can
be highly inefficient. Consequently, it is crucial to ensure efficient memory access and latency hiding when writing software, with techniques such as coalescing of memory accesses playing a significant role.

Unlike multi-core CPUs, where threads can operate independently, GPUs are equipped with execution units that execute the same instruction for each thread in the execution unit, but on different data. Branching is handled by refraining from executing any instructions in certain threads, having them wait for threads taking other paths through branching code. This implies that divergent branches in the code can be costly, as some threads will be idle during branching. However, within each execution unit, the hardware required for individual threads can be shared, allowing for less chip area per thread. The advantage of reducing the hardware needed for each thread is that it becomes feasible to accommodate significantly more threads on the same silicon area and power envelope.

The large number of cores coupled with the extensive amount of available bandwidth enable GPUs to efficiently process substantial volumes of dense data. This quality makes GPUs well suited for tasks where memory accesses are predictable and data can be structured into large enough chunks, for example matrix-matrix multiplication. They may be less suited for tasks involving lots of pointer chasing and random memory access, such as code compilation or databases.

2.6.2 Other Accelerators

There are other accelerators available for offloading work off a CPU. A Field Programmable Gate Array (FPGA) can be used as “programmable hardware”, implementing any desired functionality directly in hardware. An Application Specific Integrated Circuits (ASIC) is an integrated circuit made specifically for one application.

2.6.3 Interconnection Networks

In the context of this report, “interconnect” or “fabric” will denote high-performance networks connecting compute nodes within an HPC-cluster. The interconnection network in a compute cluster is what enables efficient parallelization of distributed applications.

Remote Direct Memory Access (RDMA) is an important aspect of a compute fabric. RDMA allows access to memory located on other nodes within the cluster, without requiring any data to pass through any CPUs. Bypassing the CPU can also improve latency, as network cards can directly access memory via the PCIe-bus. In contemporary fabrics, this capability extends to the GPUs and the storage.

The network topology used for compute fabrics in HPC often differ from other types of networks. Low latency and high bisectional bandwidth is of great concern. Topologies such as fat trees are often used, as to avoid communication bottlenecks.

---

14Where consecutive GPU threads access consecutive memory addresses.
3.1 Skeleton Programming Taxonomy

A survey of available frameworks for parallel programming was made in 2009 by González-Vélez and Leyton \[15\]. The authors present and compare over 20 different skeleton programming frameworks, both data- and task-parallel, in several languages. We must note that the research is 15 years old at the time of writing, when GPGPU was still a relatively new subject. The authors provide a method for classification and comparison of skeleton programming frameworks. They use this to look at many of the then available frameworks and libraries, some of which are still active projects.

From the surveyed skeleton programming frameworks, González-Vélez and Leyton recognized commonly implemented skeletons. These were classified into three different major categories, forming a rough taxonomy:

**Data-parallel** Skeletons that parallelize an operation across a data structure. Map and Reduce are examples of such data parallel patterns. Most of the SkePU API fall within this category.

**Task-parallel** Skeletons that enable application programmers to parallelize using tasks, forming a graph of tasks, subject to parallelization. While StarPU is not a skeleton programming framework, the programming model is similar.

**Resolution** Skeletons that enable application programmers to work within a specific family of problems. González-Vélez and Leyton mention divide-and-conquer and branch-and-bound as examples. In a more contemporary context, we could consider for example back-propagation to fall within this category.

Additionally, González-Vélez and Leyton looked at the programming paradigms used. In addition to the “usual suspects” of programming paradigms, functional, imperative and object-oriented, González-Vélez and Leyton use coordination as a separate paradigm. This refers to a paradigm where application programmers describe the algorithm in a high level language, separately from lower level code dealing with the underlying infrastructure. The surveyed frameworks are also classified on skeleton nesting, the capability to hierarchically apply skeletons.
CHAPTER 3. RELATED WORKS

Classification of SkePU and StarPU  As the survey is older than SkePU, SkePU is not one of the surveyed languages. However, we would still like to classify SkePU as a (mostly) data-parallel, object-oriented framework, with little focus on skeleton nesting. While StarPU was neither surveyed by González-Vélez and Leyton, nor is it a skeleton programming framework, we could argue that StarPU is a task-parallel library, close with both the coordination and imperative paradigms.

3.2 Quaff

Introduced in 2006, the task parallel skeleton library Quaff [14] leans heavily into C++ templates. SkePU makes heavy use of C++ templates internally, but does not require the application programmer to do any heavy template programming. Quaff on the other hand, relies on the application programmer to define a task graph using only templates and types. According to the authors, this is the defining feature of the library. The task graph is then fully instantiated during compile time, targeting MPI. By extensive use of compile time template expansion, Quaff exhibits little overhead, including on soft real-time tasks. Conceptually, the application programmer writes C++ functions normally, then use templates and the type system to define a task graph using the provided skeletons, that the compiler can instantiate into a callable object. In a similar fashion to what SkePU achieves, this fully eliminates virtual function calls.

3.3 The Muenster Skeleton Library (Muesli)

The Muenster Skeleton Library [10], [42] is a C++ library targeting execution on CPU, GPU and heterogeneous clusters, on top of MPI, OpenMP and CUDA. The library provides for both data parallel and task parallel skeletons.

3.3.1 Simultaneous CPU-GPU Execution in the Muenster Skeleton Library

Relevant to the multivariant tasks provided by StarPU codelets and dynamic scheduling, Wrede and Ernsting [44] explore methods for how to use both CPU cores and GPUs within a system simultaneously. Wrede and Ernsting create a distributed matrix across available nodes, equally divided among the ranks. However, once on a rank, the local part of the matrix is statically divided up among available CPU and GPUs. For example, one can assign 20% of the rank-local data to a CPU and 40% each to two GPUs. The experimental results show that it is possible to get a speedup likely 10 ∼ 20% in many problems. Wrede and Ernsting suggest exploring both dynamic and static approaches for determining how to divide work. We note that Wrede and Ernsting do not mention any results with regards to the power usage of the presented methods, this may be an avenue for future work.

The strategy for memory distribution is conceptually similar to previous work on heterogeneous execution in SkePU by Öhberg, Ernstsson and Kessler [23]. The StarPU backend we present could be useful for further developments in this direction, as this could enable dynamic scheduling through StarPU. The performance characteristics of such dynamic systems are not fully evident, but the work on Muesli could provide some indication of what to expect.

---

1The separation of “code that schedules” (infrastructure) and “code that is scheduled” (algorithm) is an integral part of StarPU’s design, providing a separation between the algorithmic implementation and the underlying infrastructure.

2The authors make no mention as to the readability of compiler errors.

3And we strongly concur.

4We must note that while this work by Wrede and Ernsting [44] is relatively recent, published 2018, there has been significant changes in available hardware since, best exemplified by historical Top500-lists [43], with, among other changes, an increasing focus on GPU-performance and directly connecting GPUs to interconnects. We do not expect contemporary hardware to behave in a similar manner.

5With outliers, for example the n-body problem yields a 52% speedup.
3.4 Chapel

Chapel [5, 41] is a programming language developed primarily by Cray/HPE for use on large scale parallel systems. Unlike libraries and frameworks such as StarPU, SkePU and Muenster, the programming language itself assumes distributed memory. Thus, Chapel includes, for example, distributed matrices in the language. Instead of ranks, Chapel uses the term *locales*, as most communication is one sided. Programmers can transparently read from and write to variables and data stored on other locales, implemented using one sided RDMA operations. Listing 3.1 shows a simple Chapel program computing a dot-product. Note that the computation will execute in parallel across available locales, automatically parallelized and vectorized. The mechanism in play here is that the operator * is automatically promoted to execute in parallel across the arrays forming its operands, while reduce is a built-in parallel construct. For more control, Chapel offers language primitives for parallel and vectorized loops, as well as constructs for task parallelism, aiming to allow programmers to use lower level constructs when needed.

One drawback of such a solution, as compared to skeleton solutions, is the loss of the C++ ecosystem. Chapel attempts to mitigate this by providing a FFI but it is not transparent to the user. Chapel also supports GPUs, with devices available as sublocales. GPGPU is, as of writing, an active area of development. Instead of compiling to CUDA or OpenCL, the Chapel compiler generates the requisite byte code as part of normal compilation, leveraging existing LLVM infrastructure.

Listing 3.1: Computing a dotproduct in Chapel.

```chapel
use Random, BlockDist;

// Configuration parameters
config const size = 4096;
config const seed = 0;

// Define a data distribution
const BlkSpace = blockDist.createDomain({1..size});

// Create two distributed arrays and fill them with randomness
var A: [BlkSpace] real; fillRandom(A, seed);
var B: [BlkSpace] real; fillRandom(B, seed+1);

// Compute the dotproduct
var dotproduct = (+ reduce (A*B));
writeln(dotproduct);
```

3.5 Other Work on SkePU

We describe the work done on SkePU itself in Section 2.1. Of particular note is other work on distributed back-ends, such as the work by Majeed, Dasgupta and Kessler on Cluster-SkePU [21], the MPI+OpenMP back-end by Zeijlon [45] and the GPI back-end by Almqvist [1]. Additionally, the EXA2PRO-project [28] and SkePU 3 [12] extend the work we present here.

---

6With various libraries available as an underlying communication mechanism, including MPI.
7Foreign Function Interface, an interface for calling code written in other languages.
Research questions one and two both relate to how models, assumptions and APIs within and between SkePU, StarPU and MPI interact when put together. We explored these two research questions by implementing a solution, documenting the result.

Research question three relates to the performance characteristics for such a system. To measure these, we will benchmark SkePU 2 StarPU-MPI against a reference implementation.

4.1 Implementation

We created the implementation with a few high level requirements in mind

- Existing SkePU-code should keep working, this implies that the semantics and syntax of the API should be preserved as much as possible,
- StarPU should be used to parallelize SkePU-code across MPI-ranks,
- The software architecture should be possible to extend with GPU-support in the future, even if it may not be implemented initially.

We created a high-level software design, followed by an implementation based on these high-level requirements. We iteratively created the implementation, continually evaluating and changing the design and the code. The end product presented and analyzed as the result of this project is a result of this process. Time available to us limited the scope of this project, and a fully working, bug-free implementation of all of SkePU was not achievable. To prioritize during development, we focused on creating an extensible structure, allowing for future development while also ensuring the preservation of the SkePU API.

We implemented a small amount of automated tests to ensure semantics were preserved between the existing SkePU-API and the developing SkePU 2 StarPU-MPI implementation.

4.2 Performance Characteristics

In order to analyze the performance of SkePU 2 StarPU-MPI, we studied performance for two different problems, the n-body problem and a simple summation to serve as a reduction. We compared SkePU 2 StarPU-MPI to hand-written reference implementations using MPI and OpenMP.
4.2. PERFORMANCE CHARACTERISTICS

4.2.1 Performance Analysis

In order to analyze the performance characteristics of the implemented StarPU backend to SkePU, we selected two different algorithms, an n-body solver to benchmark the map skeleton and a simple summation to benchmark the reduction and communication capabilities. The MPI-collectives `Allgather` and `Reduce` were used in the reference implementations, which should roughly align with the communications in the implementations using SkePU 2 StarPU-MPI.

Reference Implementations For both algorithms, we implemented functionally equivalent MPI/OpenMP codes to act as reference implementations to enable comparisons on absolute performance and not only scaling factors. The inner loops of the MPI-implementations used the exact same code as the user functions used for the SkePU+StarPU implementation, but using raw C pointers instead of SkePU containers. To avoid the compiler applying optimizations that altered the performance characteristics too much, these programs were compiled with `-O0`. In the case of SkePU 2 StarPU-MPI we compiled all binaries with `-O3`.

SkePU Implementations For the SkePU 2 StarPU-MPI benchmarks, we used a slightly newer version of the SkePU header library than what we wrote as this commit fixed some issues with deadlocks. However, the changes between what we implemented and this version are fairly small and should not affect overall performance. We well note that this will technically bring the SkePU 2 StarPU-MPI implementation into very early SkePU 3 territory, but we consider the change small enough to still use the designation “SkePU 2 StarPU-MPI” for the purposes of this report. To ensure compatibility with the selected header library code, we used an early version of the SkePU 3 precompiler. The source code to SkePU can be downloaded from repositories on GitHub.

Benchmark Environment We created two Apptainer containers, one with the SkePU precompiler and one with the StarPU run-time, a compiler suite and the needed support libraries to run StarPU distributed across a cluster using Slurm as the scheduler. Relevant to performance are

- StarPU v1.3.11,
- UCX v1.13.1,
- OpenMPI v4.1.5,
- GCC v11.3.1.

Other packages were present, but should not affect performance. The MPI references were compiled and executed within the same environment, using the same compiler. We based the container images on Rocky Linux, the SkePU precompiler on Rocky Linux 8 and the runtime on Rocky Linux 9. The differences in versions made it easier to fulfill the required dependencies. As for hardware, the benchmarks ran on two nodes, each with dual socket AMD Epyc 7742 CPUs with 64 cores each. Each node has eight rails Mellanox HDR 200Gb/s Infiniband on a non-blocking fat-tree. Each rank was assigned 16 cores, but was only allowed to use 8 for compute. The StarPU runtime was allowed an additional control thread, and thus used 9 cores in total. Allocating 8 cores for compute had good reasons. First of all, it’s the largest power

1 This shouldn’t be an issue with larger codes, but in this case, with what should be considered microbenchmarks, there are more possibilities for the compiler to add unexpected optimizations, such as aggressive constant folding. The compiler was especially eager to apply what we presume was constant folding to the reference reduction implementation.
2 Git commit f6f403c6 by Johan Ahlqvist.
3 Git commit f6f403c.
4 Configured with 2 NUMA nodes per socket.
5 As configured via `STARPU_NCPU` and `OMP_NUM_THREADS`.
of two that still allows for a separate control thread on an exclusive core. The second reason was that only using half of the available cores (and a quarter of available SMT-threads) should further reduce the chance of congestion on the resources that remain shared, such as memory bandwidth.

For the StarPU-runtime, we were careful to allow cores to be pinned properly. For each rank, all threads were pinned to cores within the same NUMA node. We confirmed that the pinning worked as expected via normal Linux monitoring tools, and also confirmed that the “other” threads were not used.

Using UCX as the underlying communication library, we could pin each rank to a specific network card and disallow any and all shared memory communication between ranks. This means that any MPI-communication between two ranks on the same node are forced to communicate via the fabric. While UCX is very much capable of utilizing multiple rails for a single rank, this was explicitly disabled in favour of assigning each rank its own interface. We confirmed that all interfaces were used during benchmarking by using external network monitoring tools, looking at Linux port counters.

We designed this setup to allow for fewer physical nodes, while still maintaining a very good approximation of 16 nodes. In principle, the same design should scale out to more nodes as well, but we deemed 16 ranks to be enough. We used overprovisioned hardware and actively underutilized it, in order to ensure we benchmarked the characteristics of software, and not software on top of some specific hardware architecture.

Selection of Problem Parameters  We selected various problem sizes for both the n-body and the reduction benchmarks, selecting the various problem sizes to indicate the extreme scaling properties. The smaller sizes were expected to exhibit “bad” scaling, while the large sizes were expected to exhibit “good” scaling. To allow for maximum parallelism, we selected all parameters as powers-of-two, including the number of ranks.

Measurement of Wall Time  For all implementations, we only measured the inner loop. We measured time on rank 0 with MPI_Wtime, barrier to barrier. The inner loops that performed the actual calculations ran for a set number of iterations, selected to get the measured time up into the range of a second or two for most parameter sets. We stored the times into log files, then later aggregated these into CSV-files for later analysis. We ignored any time spent outside of the inner loops, such as time used for startup, data initialization memory allocations or cleanup.

6For 8 ranks, we have a maximum of $128/8=16$ cores available per rank.
7We set the environment variable UCX_NET_DEVICES based on SLURM_LOCALID.
Chapter 5

Design and Implementation

In this chapter we describe how we add MPI-support to SkePU through the use of the StarPU library. This includes descriptions of the implementation of smart containers, memory management and scheduling.

The full source code of SkePU is available online [37].

5.1 Method

The development process for SkePU 2 StarPU-MPI follows a freeform iterative development process [20], starting from broad requirements. We created the implementation with high level requirements in mind:

- Existing SkePU-code should keep working, which implies that the semantics and syntax of the API should be preserved as much as possible.
- StarPU should be used to parallelize SkePU-code across MPI-ranks.
- The software architecture should be possible to extend with GPU-support in the future, even if it may not be implemented initially.

We created a high level software design, followed by an implementation based on these high level requirements. We iteratively created the implementation, continually evaluating and changing the design and the code. The end product presented and analyzed as the result of this project is a result of this process. Time available to us limited the scope of this project, a fully working, bug-free implementation for all of the SkePU API was not achievable. To prioritize during development, we focused on creating an extensible structure, allowing for future development while also ensuring the preservation of the SkePU API.

We implemented a small amount of automated tests to ensure semantics were preserved between the existing SkePU-API and the developing SkePU 2 StarPU-MPI implementation.
CHAPTER 5. DESIGN AND IMPLEMENTATION

5.2 Overreaching Design Decisions

The fundamental differences in a distributed programming model required us to allow for some changes in how SkePU works when running distributed. The largest difference is that we require control flow to be coherent across all ranks. Additionally, we decided to allow for some changes in how containers work.

5.2.1 Control Flow

A large part of the operations in the MPI-standard require participation from all ranks involved in the operation. This implies that control flow should be largely coherent across ranks. The MPI-functionality in StarPU also follows this principle, greatly affecting the design of SkePU

StarPU-MPI. Previously, users of SkePU could do any operations outside of the skeletons, such as writing to files or communicating over the network. However, in the MPI-variant it was decided to not fully support this, due to the need for coherent control flow. SkePU-programs compiled with the implemented MPI-support are running as one copy per rank in the cluster. This places some additional responsibility on the SkePU user to ensure coherency.

5.2.2 Containers

One large source of boilerplate code in initial experiments were the use of two different container classes, skepu2::Vector<T> and skepu2::Matrix<T>. Since a vector is mostly just a special case of a matrix, they were joined together into a single data structure, skepu2::Matrix<T>. A single typedef provides the skepu2::Vector<T> class. This is a big change in terms of API from the previous SkePU implementation. Care was taken to ensure as much compatibility as possible. No changes to the API were made exclusively due to this, but it was a contributing factor to changes in iterator semantics. This change reduced the code complexity by quite a lot, since there is only a single implementation of a data container.

Disallowing References to Container Elements As any container element can, potentially, reside in the memory of any other rank, directly accessing such memory cannot be done. While the implementation transparently fetches and writes data according to the SkePU-API in most cases, C++ references to elements could not be solved. As such, statements such as for example a[b] = 1 are disallowed and must be replaced with a.set(b, 1).

5.3 Memory Management

In this section we describe how we manage memory, from the ground up.

5.3.1 Low Level Container Indexing and Blocking

To manage low level indexing and partitioning of data, we implemented the cut_structure class. The class allows for splitting up a 1D vector of data into smaller blocks of data, as demonstrated in Figure 5.1. We can then use the class to perform various index and size calculations. To support this, we store some metadata about how the 1D vector is cut up. While the class contains some metadata on how data is partitioned, the class does not store any actual data. Listing 5.1 shows the public API of this class. The partitioning scheme we use for cut_structure is fairly simple, we structure everything into a specified number of blocks of equal size.

By using two instances of cut_structure, one horizontal and one vertical, we can split up a matrix into a grid of blocks. Figure 5.2 demonstrates this. While we could use other partitioning schemes, this was flexible enough, while still being fairly easy to implement. Later on, we use this pair of one vertical and one horizontal set of cuts to implement the central parts of the data distribution.
Listing 5.1: Public API of the cut_structure class.

```cpp
class cut_structure {
public:
    inline cut_structure(const size_t &size, // Total size of structure
                          const size_t &parts // Desired number of blocks
    );

    // Yield an index to first element contained within a block
    inline size_t block_start_idx(const size_t &block) const;

    // Yield an index to the first element in the *next* block
    inline size_t next_block_start_idx(const size_t &block) const;

    // Yield an index of block containing a specific global element
    inline size_t block(const size_t &idx) const;

    // Total size of the data structure
    inline size_t size() const;

    // Number of blocks in total
    inline size_t block_count() const;

    // Size of the block containing an element at index
    inline size_t block_size_by_elem(const size_t &idx) const;

    // Size of specified block
    inline size_t block_size(const size_t &block) const;

    // Offset from start of block for an element at an index
    inline size_t offset_in_block(const size_t &idx) const;

    // Yield the number of elements from idx to end-of-block
    inline size_t size_in_block(const size_t &idx) const;
};
```

Figure 5.1: Splitting up a large vector into multiple smaller blocks.

### 5.3.2 Intermediate Data Containers

The intermediate class starpu_matrix_container is an implementation of a distributed matrix, built using StarPU handles and the cut_structure class. Listing 5.2 shows the public API for this class. We can later use this class for building other containers.

On a high level, this class implements a distributed “matrix of matrices” together forming a larger matrix, using the distribution shown in Figure 5.2. This class provides an easy and safe way to access both the StarPU handles (for task scheduling purposes) as well as directly accessing underlying data. Within the class, we store a matrix of StarPU handles. Each StarPU handle then refers to some data representing a matrix. However, while we have a reference to all data on all ranks, we do not allocate all data on all ranks. Only the owning rank actually allocates the memory. We stripe ownership, and thereby allocations, across ranks in a round-robin fashion. The current API contains no provisions for configuring how we distribute the data. We have used a simple heuristic, where we have one block per rank and per dimension, but do not expect this to be the best way to do this. While more complex heuristics are possible, we decided against this to save on time and implementation complexity. Internally, we use the public API of cut_structure, and tried hard to not make assumptions outside of this API. This makes it possible to add appropriate API surface that configures the data partitioning scheme without changing much, or any, internal mechanisms.

---

1. For example, a $16 \times 16$ matrix across 4 ranks will become 16 blocks of $4 \times 4$. 

As data distribution and locality can be a very important factor for performance, we concentrated all logic dealing with data distribution into the `starpu_matrix_container` class. That way, there is a single location where we deal with such issues. During the rest of the development, we tried to limit the number of assumptions the rest of the system has w.r.t. data distribution, keeping in mind that the round-robin fashion in which we distribute data may not be optimal for all use cases. In particular, we tried to abstract away the concept of ownership as much as possible. We did this aiming to make future modifications to these parts as easy as possible.

To fulfill the API-requirements of `skepu2::Matrix<T>`, there are methods for getting and setting individual elements in the distributed matrix. Since we use `starpu_matrix_container` during scheduling, we also need to allow for direct access to the underlying StarPU handles. To prevent any other part of the implementation from relying on some particular data distribution or ownership details, we made the two `cut_structure`s, horizontal and vertical, fully private.

Allgather

In some cases, the SkePU 2 API requires that user functions have random access into the entirety of a container. In order to support this use case, the `starpu_matrix_container` provides the `allgather` method. This operation has similar semantics to the allgather operation in MPI. Calling this method yields a read only StarPU handle with a single contiguous copy of the full matrix on all ranks. The aptly named `invalidate_unpartition` method provides a way of invalidating such copies. The application programmer writing code targeting the SkePU API should not need to care about such low level details. We issue such invalidations whenever

---

2While writing to this memory is certainly possible, we did not design the API with that in mind.
Listing 5.2: Public API of the starpu_matrix_container class.

```cpp
template<typename T> class starpu_matrix_container {
public:
    // Constructor and destructor
    starpu_matrix_container(const size_t &height, const size_t &width);
    ~starpu_matrix_container();

    // Information on size
    size_t width() const;
    size_t height() const;
    size_t size() const;

    // Utility methods for scheduling, pass through to 'cut_structure'.
    size_t row_block_height(const size_t &row) const;
    size_t col_block_width(const size_t &col) const;

    // From an element idx, yield a StarPU handle and a 2D offset to the specified
    // element. Additionally, yields remaining 2D size within the block, starting
    // from idx.
    helpers::handle_cut largest_cut(const Index2D &idx);

    // Setters and getters for single elements
    T operator()(const size_t &row, const size_t &col);
    T operator[](const size_t &col);
    void set(const size_t &row, const size_t &col, const T &value);
    void set(const size_t &i, const T &value);

    // Yield MPI-rank that owns a specific block or element
    size_t elem_owner(const size_t &row, const size_t &col) const;
    size_t block_owner(const size_t &block_row, const size_t &block_col) const;

    // Yield a StarPU-handle to a block
    starpu_data_handle_t & get_block_by_elem(const size_t &row, const size_t &col);
    starpu_data_handle_t & get_block(const size_t &block_row, const size_t &block_col);

    // Yield a StarPU-handle that will, eventually, contain the full matrix
    starpu_data_handle_t allgather();

    // Invalidate the data in the handle give out by 'allgather'. If allgather has
    // been called, this must be called on any subsequent write to the distributed
    // matrix
    bool invalidate_unpartition();
};
```

there is a write to any element in the starpu_matrix_container from outside the task graph\(^1\)
or whenever we schedule a task using a handle from the container in a writable manner\(^2\).

5.3.3 Matrix API

We preserved most of the API that SkePU 2 has for skepu2::Matrix<T>. However, we added some changes and additional limitations, which break backwards compatibility. The implementation of skepu2::Matrix<T> in SkePU 2 StarPU-MPI relies heavily on the abstractions starpu_matrix_container provide. Listing 5.3 shows the declaration of the skepu2::Matrix<T> class. Our implementation of skepu2::Matrix<T> only contains a 2D-size, a 2D-offset and a shared pointer to a starpu_matrix_container. Thus, we should consider skepu2::Matrix<T> a view into some underlying data. Several instances of skepu2::Matrix<T> can reference data at different positions contained within the same instance of a starpu_matrix_container. However, the current API takes little advantage of this\(^3\). Such functionality could be useful for implementing divide-and-conquer style algorithms.

---

\(^1\)For example via starpu_matrix_container::set, exposed via the API for skepu2::Matrix.

\(^2\)There are ways to allow for partial invalidation, which could avoid expensive retransmissions of data. Note that this is mostly relevant for the allgather method, as StarPU’s caching functionality will deal with most other cases.

\(^3\)We made some attempts to add support for explicit use of a MatrixView<T> and creating implementations of iterators based on this, but abandoned the attempt in order to avoid the rather large changes to the public API of SkePU it would entail. It might even require some larger changes to the precompiler. However, we consider the functionality provided by Matrix::subsection to be good enough for most cases.
Listing 5.3: Declaration of the starpu2::Matrix<T> class in SkePU 2 StarPU-MPI.

```cpp
template<typename T>
class Matrix {

private:
    // Reference to underlying starpu_matrix_container. Note that this is a shared
    // pointer, multiple matrices can use the same underlying distributed data.
    std::shared_ptr<cluster::starpu_matrix_container<T>> m_data;

    // Size and offset into m_data for this particular matrix. In practice, most
    // operations within this class are implemented as a passthrough into
    // starpu_matrix_container, only offset by m_offset and range-checked with
    // m_size.
    Size2D m_size;
    Offset2D m_offset;

public:
    // Public skepu2::Matrix API with no changes.
    // Omitted from this listing for brevity, all without significant changes.
    // - Constructors
    // - Destructors
    // - Various typedefs
    const Matrix<T> & getParent() const;
    Matrix<T> & getParent();

    size_t size() const;
    size_t total_rows() const;
    size_t total_cols() const;
    size_t capacity() const;
    bool empty() const;

    void flush();
    void clear();

    void swap(Matrix<T> & from);
    Matrix<T> & operator~(); // Transpose

    typedef MatrixIterator<T> iterator; // While semantics are modified, iterator
    // API remain unchanged.
    MatrixIterator<T> begin();
    MatrixIterator<T> end();

    // These are unchanged, but noteworthy. This is the API for creating a new
    // Matrix instance targeting the same underlying starpu_matrix_container.
    Matrix<T> subsection();
    Matrix<T> subsection(size_t row, size_t col, size_t rowWidth,
                         size_t colWidth);

    // These methods are changed, but present within the SkePU 2 API. The main
    // difference is that the non-distributed API return writable references to
    // the underlying memory (‘T&’), something we cannot provide with distributed
    // memory.
    T at(size_t index);
    T at(size_t row, size_t col);
    T operator[](const size_t index);
    T operator[](const Index2D index);
    T operator[](const size_t & row, const size_t & col);

    // These additions allow us to write individual elements, something previously
    // provided by writable references.
    void set(size_t index, const T & data = {});
    void set(size_t row, size_t col, const T & data = {});

    // These are additions, and passthrough to underlying starpu_matrix_container.
    // Due to implementation details, these are public, even though they are not
    // intended for use by application programmers.
    size_t block_height_from(const Index2D & idx);
    size_t block_width_from(const Index2D & idx);
    cluster::helpers::handle_cut largest_cut_from(const Index2D & idx);

    // Provide access to the internal workings of the class. Again, these are
    // public due to implementation details. Note that we don’t return a shared
    // pointer, making the lifetime of the referenced value less well defined (but
    // not strictly undefined).
    cluster::starpu_matrix_container<T> & data();
    const Offset2D & offset() const;
    const Size2D & size2D() const;
};
```
Listing 5.3 contains the declaration of this class. While we proxy some functionality from starpu_matrix_container to allow for scheduling tasks later on, we do not intend for application programmers to use those.

The StarPU API disallows access to data owned by a StarPU handle from any code running outside of the task graph. We can explicitly make a specific handle readable or writable at some point in the task graph. This requires calling appropriate synchronization mechanisms, that insert barriers and memory transfers into the task graph. This allows us to access the data without risking race conditions. However, we need to let go of this access at some point, to allow StarPU to resume execution of the task graph. This makes it very hard to allow the application programmer to own a reference to data owned by a StarPU handle. Thus, we had to remove this possibility from the API. In practice, we removed the `T& operator[](const size_t index);` operator providing a writable reference for an element. This means that application programmers cannot use expressions such as `a[idx] = data;`, and must replace these with something like `a.set(idx, data)`. Additionally, as we distribute the memory across ranks, this is now an expensive operation that may need global synchronization and invalidation. Ideally, the application programmer should use a user function instead.

### 5.3.4 Iterators

SkePU uses iterators to allow for applying a skeleton on only parts of a container. As such, they are fairly central to the API and implementation. Due to the unification of `skepu2::Vector<T>` and `skepu2::Matrix<T>`, as well as some differences arising from using distributed memory, we decided to modify the semantics of iterators. In SkePU 1 and 2, iterators iterate over contiguous elements. However:

- We are working with a distributed matrix, and contiguous elements is no longer equivalent to contiguous memory.
- Attempting to emulate the semantics could be possible, but would lead to extra complexity.
- Iterating over contiguous elements allows for divide-and-conquer in only one dimension.

Thus, we decided to change the semantics and let iterators operate over a rectangular region. The first element is in the upper left corner, while the last element is in the lower right corner. Figure 5.3 shows this in action. Note that the semantics remain identical for 1D vectors, so we retain partial backward compatibility.

![Figure 5.3: Changes in the semantics of SkePU iterators.](image)
5.4 Meta Skeleton Programming

During the initial implementation efforts, we concluded that there would be a large amount of “boilerplate” for each skeleton, to ensure we propagate type information throughout the skeleton implementations and the StarPU API. In an attempt to reduce the complexity and extra code, we decided to create an abstraction layer between the implementation of the skeletons and the StarPU API by means of metaprogramming. This “meta skeleton” took the form of the class `multivariant_task`. Its most prominent feature is to create StarPU codelets and ensure the proper handling of types within the created codelet. `multivariant_task` also handles the creation of StarPU tasks, performing much of the work one would expect the skeletons to perform. Listing 5.4 shows the public API of the class. At first glance this looks rather scary. Fortunately, application programmers should not need to care about this class unless they want to implement their own skeletons.

To explain how the `multivariant_task` template fits together, we will go through the template arguments and the provided methods. Finally, we will explain the intended usage pattern.

```
template <typename ResultArgs, typename ElwiseArgs, typename ContainerArgs, typename UniformArgs, typename Self>
class multivariant_task {

private:
  // Private details omitted for brevity
public:
  multivariant_task();

  template <typename... Args, size_t... RI, size_t... EI, size_t... CI, size_t... UI>
  void element_aligned_impl(pack_indices<RI...>, pack_indices<EI...>, pack_indices<CI...>, pack_indices<UI...>, Size2D size, Args &... args);

  template <typename... Args> void element_aligned(Size2D size, Args &... args);

  template <typename... Args, size_t... RI, size_t... EI, size_t... CI, size_t... UI>
  void element_aligned_res_per_block_impl(pack_indices<RI...>, pack_indices<EI...>, pack_indices<CI...>, pack_indices<UI...>, Size2D size, Args &... args);

  template <typename... Args> void element_aligned_res_per_block(Size2D size, Args &... args);

  template <typename... Args, size_t... RI, size_t... EI, size_t... CI, size_t... UI>
  void element_aligned_sweep_impl(pack_indices<RI...>, pack_indices<EI...>, pack_indices<CI...>, pack_indices<UI...>, Size2D size, const skepu::SweepMode dir, Args &... args);

  template <typename... Args> void element_aligned_sweep(Size2D size, const skepu::SweepMode dir, Args &... args);

  /* SkeletonBase compat fns. */
  auto finishAll() -> void {};
  template <typename T> auto setExecPlan(T) -> void {};
  template <typename T> auto setBackend(T) -> void {};
  auto resetBackend() -> void {};

  starpu_codelet *cl;
  struct starpu_perfmodel *perf_model;
};
```
5.4.1 Task Arguments

SkePU’s API allows for large flexibility with regards to what arguments application programmers can pass to skeletons and user functions, as well as what they can return. The details in how SkePU propagates these arguments through the code are fairly involved. The API and internal mechanics were initially designed by Ernstsson for SkePU 2 [13]. SkePU 2 StarPU-MPI uses very similar mechanics, even though we add multivariant_task as an additional abstraction. The specifics vary between different skeletons, but in general we can use the following argument types:

**Elementwise arguments (EI)** With elementwise arguments, we pass a container to the instance of the skeleton, which then passes a copy of a single element from container to the user function. Skeletons usually iterate over the elements in the elementwise arguments.

**Container arguments (CI)** Some skeletons allow for random access to a full container inside of a user function. We pass the skeleton instance a container, which is then passed on to the user function.

**Uniform arguments (UI)** Some skeletons allow the user to specify arguments passed on as-is to the user function. We give a value to the skeleton which is then passed on as a copy to each invocation of the user function. We can use this to pass constants into user functions.

**Results (RI)** Skeletons need to store the result in some way, either in a single variable or a container. Some skeletons, such as Map, expose this to the application programmer.

We designed multivariant_task to allow for as many combinations as possible of these. The parameter packs in the templates allow us to provide as many arguments as we need. One could for example allow for two elementwise arguments, one container argument with random access and single value as an output. As we do this using templates, we can preserve type information, from the application programmer invoking the skeleton, through to the user function, and then back. The methods that multivariant_task provides will use the type information to generate appropriate codelets as well as calls to the user functions.

5.4.2 Access Patterns

For the application programmer using the SkePU-API, the skeletons aim to provide a high level of abstraction to a specific computational pattern. In the same manner, multivariant_task aims to provide a high level API to specific memory access patterns for the distributed data structures we use in SkePU 2 StarPU-MPI. While Map, Reduce and MapReduce are three different skeletons, they all access data in a similar manner, but perform their work differently. In the non-distributed case, the amount of code required to implement the “access” is fairly small and most programmers would probably just duplicate the needed code in all three skeletons without thinking more about it. However, in the distributed case we are working with here, we need to deal with more issues, including data transfers, scheduling, cumbersome indexing and synchronization. Thus, we aim to create an abstraction.

---

6This is why we needed the method starpu_matrix_container::allgather.

7pack_indices is a homegrown variant of std::integer_sequence. As std::integer_sequence appears in C++14 and SkePU 2 use C++11, pack_indices appears throughout the implementation of SkePU 2.
5.4.3 Alignment

The SkePU 2 API allows us to use iterators as elementwise arguments when invoking skeletons. This gives the application programmer a great deal of flexibility. Important to note is that the API is not limited to a single elementwise argument, the API allows multiple such arguments as long as they are all of the same size. This works well and is not too complex to implement if we assume the "old" SkePU 2 iterator semantics together with the assumption that each iterator refers to a single block of contiguous memory. This is not the case in SkePU 2 StarPU-MPI:

- Iterators specify a rectangular view, not a contiguous block of memory.
- The data referred to by an iterator will be owned by some rank.
- The underlying data structure, starpu_matrix_container, uses a matrix of matrices. Containers of different sizes can use sub-matrices of different sizes.

Thus, two iterators can have wildly differing storage layouts, while still being valid skeleton arguments. This means that we may not have a 1:1-mapping between

- StarPU-handles used to store elementwise arguments,
- StarPU-handles used for output data,
- Tasks that perform the computation.

We must reconcile this in some way in order to retain the API.

While we cannot retain a 1:1-mapping between handles and tasks anymore, we must maintain a 1:1-mapping between the elements in all elementwise arguments. E.g. if we invoke a Map skeleton on two matrices, the user function will get access to one element from each matrix, each such element coming from the same position in the matrices (e.g. a[1, 2] and b[1, 2]). We could launch a StarPU task for each element, with the correct handles provided. However, that would cause huge scheduling overhead. Additionally, StarPU must serialize writable access to the output-handles, thus fundamentally limiting the parallelism. We mitigate this by scheduling one task for each the largest possible regions without any boundaries in any of the elementwise arguments.
Figure 5.4: Scheduling method for aligned elementwise arguments.
An Illustrating Example  Figure 5.4 illustrates this concept of element aligned access to blocks. We have three distributed matrices with data, “Output” (3 × 3), “Input A” (6 × 3) and “Input B” (3 × 6). These are all starpu_matrix_containers. In this example, we would like to perform a task with the top left 3 × 3 of each container as an elementwise argument. As a concrete example, this could be an elementwise addition of two 3 × 3 matrices formed by iterators, stored into an output container.

First, we “overlay” all the elements we would like to operate on, the top-left 3 × 3 region. In this example, we can see 4 distinct regions, with sizes 2 × 2, 1 × 2, 2 × 1 and 1 × 1. Together, these form the full desired 3 × 3 matrix.

We can then map tasks against each such region. That task will be responsible for performing all the computations for a region. In this example, task 2 will handle the bottom-left 2 × 1 region. One could say that each task has a shape and a position. In the implementation, we often refer to this as a “block”.

Finally, we ask StarPU to schedule everything for us. We schedule each task with the handles it needs to operate on, but they may not need to access all of the data contained within the handles. For example, we need to schedule the bottom-left task 2 with

- Handle 1 from “Output”, size 3 × 1,
- Handle 0 from “Input A”, size 2 × 3,
- Handle 1 from “Input B”, size 3 × 2.

For each handle we provide, we also need to provide an offset so the task knows where the relevant elements are. One important detail is that StarPU must add ordering dependencies between tasks writing to the same handles, so they are not scheduled concurrently. In this example, tasks 0 and 1 cannot be concurrent, neither can tasks 2 and 3, thus limiting us to two concurrent tasks.

5.4.4 Meta Skeleton Access Patterns

SkePU 2 StarPU-MPI contains two fundamental data access patterns, with an additional variant.

Elementwise Access

The first fundamental pattern we provide is the same pattern we described in the previous section on alignment, as shown in Figure 5.4. We aim to access each element in each elementwise array. The results from the operation are also assumed to be elementwise. Additionally, the user can provide uniform or container arguments as needed. This access pattern is what we later use to implement the Map skeleton.

Sweeping Access

With elementwise access, there are no data dependencies between different tasks. However, SkePU has skeletons that would require us to use results from one task in another. One example of this is the Scan skeleton. The idea here is that we pass intermediate results between tasks in some direction, introducing directional data dependencies. For example, let us say we want to compute the cumulative sum for each row in a matrix. Using the same setup as in Figure 5.4, we can do this by having task 0 compute the cumulative sums for the region it is responsible for. Then we need to put that data into a handle and pass that on to task 1, in order for task 1

---

8Using one block of data per rank initially looks like a good heuristic. However, with misaligned blocks like in this example, that could limit concurrency. Adding multiple blocks per rank could alleviate this issue somewhat, depending on the exact situation. This was one reason for the data partitioning heuristic described in Section 5.3.2. Of course, avoiding the issue entirely by only having aligned data may be the best option.
to know where we are. Apart from the additional data dependency we introduce, this pattern is similar to the elementwise access pattern and reuses much of that code.

We have implemented parts of this access pattern, but the SkePU-API is rich and we have not covered the full functionality we would need to have in this pattern in order to implement all of the SkePU-API. As we did not implement the full API for the Scan skeleton, this access pattern is also less well tested.

“Result Per Block” Elementwise Access Variant

Some skeletons first need to perform some elementwise or “local” operation, and then some type of “global” reduction into a single value. The Reduce skeleton is of course the primary example of this, as we need to access all elements, but need to combine the results from different tasks. However, each task is independent, and has no dependency to any other task.

With this pattern, we add an additional handle for intermediate results to each task we schedule. Then we can let the task graph execution proceed. Later, when all these tasks have concluded, we allow the skeleton implementation to access these handles and perform additional actions on these, so it can produce a final result.

5.4.5 Usage

Skeletons inherit from multivariant_task using the curiously recurring template pattern, allowing a static inheritance structure. Skeletons supply a static function as an implementation for the generated StarPU-codelet to use. multivariant_task then generates the skeleton implementation.

Currently, we only allow a single skeleton implementation. However, we could conceptually allow for skeletons to supply multiple different skeleton implementations, for example two different CPU-implementations and a CUDA-implementation. StarPU’s task scheduling functionality and codelets provide the tooling we would need to implement this.

As a practical example of how skeletons inherit from multivariant_task, Listing 5.5 shows excerpts from the declaration of the Map skeleton. In addition to the inheritance, the skeleton must provide an actual implementation that multivariant_task can invoke. The skeleton provides this via a method cpu. This cpu method is what we then use to generate the codelet.

Using the curiously recurring template pattern in this way may seem like overkill – why not just use higher order functions? Despite the added complexity of this curious solution, there are benefits. Primarily, SkePU skeletons are stateful objects with a lifetime, and we can mirror this pattern. Additionally, we can closely integrate the skeleton implementation with the access patterns where needed, for example to ensure everything remains static and linkable with C. Finally, in order to schedule tasks via StarPU we need to dynamically allocate some structures on the heap, primarily the codelet. Using inheritance, we can mirror the lifetime of the skeleton.

---

9In the curiously recurring template pattern, a child class inherits from a parent class, the child class giving itself as a template argument to the parent. This gives the parent full access to methods and variables in the child class while still using static binding.

10That must target OpenMP.

11This was not implemented, due to lack of time.
Listing 5.5: The Map skeleton inheriting from multivariant_task.

```cpp
template< size_t arity , typename MapFunc , typename CUDAKernel , typename CLKernel >
class Map : cluster::multivariant_task<
    std::tuple<typename MapFunc::Ret>,
    typename MapFunc::ElwiseArgs,
    typename skepu::helpers::mat_tuple_to_rau_type_tuple<typename MapFunc::ContainerArgs>::unwrapped,
    typename MapFunc::UniformArgs ,
// Finally provide ourselves as a template argument to the class we inherit
// from, this is the curiously recurring template pattern in action
    Map<arity , MapFunc , CUDAKernel , CLKernel>>
{
private:
   // [...] public:
   // CPU implementation of the skeleton
   template< typename MatT ,
    size_t ... RI ,
    size_t ... EI ,
    size_t ... CI ,
    typename ... Uniform>
   static void cpu ( const void * self ,
   Size2D size ,
   Offset2D global_offset ,
   MatT && bufs ,
   pack_indices<RI ... > ,
   pack_indices<EI ... > ,
   pack_indices<CI ... > ,
   Uniform... args);
// [...] ;
}
```

5.4.6 Typecasting

While SkePU’s skeletons and StarPU’s Codelets seem compatible on a conceptual level, they are less compatible on a lower level of abstraction. The largest point of friction is differences in how they use typing. SkePU relies heavily on types and templates throughout the implementation, while StarPU makes heavy use of untyped pointers in its public API. Internally, SkePU also relies on additional type information originating from the precompiler. The SkePU Skeletons almost exclusively expose variadic template functions, requiring additional data from the precompiler to determine what to do with the arguments. This variadicity propagates throughout the implementation, requiring explicit handling of variadic arguments when interfacing with StarPU. User functions in SkePU take strongly typed arguments, sometimes wrapped in helper classes. This means the untyped data given to the skeleton implementation must be typecast properly before calling the user function.

A detail to take notice of is that implementations used in a StarPU codelet cannot be non-static methods of a C++ class, they have to be static functions. SkePU skeletons are a non-static object when called by the application programmer. For some skeleton code, it is necessary to use member variables of that object when running user functions. In order to do this, we cast the this pointer to a void pointer and pass it as data to the codelet. In the codelet, we cast it back to the original type. This Pythonesque handling of objects allow skeletons to use non-static members inside of codelet implementations. It is debatable whether this is valid C++\textsuperscript{12}, but it allows for better preservation of the SkePU API.

\textsuperscript{12}Probably not.
5.5 Skeleton Implementations

We now have a solution for distributed memory management, as well as a high(er) level interface for scheduling tasks across this distributed memory. We can now use these abstractions to implement a skeleton. While the code listings presented in this section are not full implementations, we aim to demonstrate the core concepts while leaving out boilerplate and scaffolding.

5.5.1 Implementing the Map Skeleton

For the Map skeleton, the relevant declarations are in Listing 5.5, but we lack the actual implementation of the cpu method. Listing 5.6 shows the implementation of this method for the Map skeleton.

Listing 5.6: Implementation of the Map skeleton, using multivariant_task.

```c++

template <size_t arity, typename MapFunc, typename CUDAKernel, typename CLKernel>
void Map<arity, MapFunc, CUDAKernel, CLKernel>::cpu(
    const void* self,
    Size2D size,
    Offset2D global_offset,
    pack_indices<RI...>,
    pack_indices<EI...>,
    pack_indices<CI...>,
    Uniform... args)
{
    static_starpu::MatT & res = std::get<0>(bufs);

    for (size_t row = 0; row < size.row; ++row) {
        for (size_t col = 0; col < size.col; ++col) {
            Index2D i,col
            i.col = col;
            i.row = row;
            i.i = i.row*res.ld + i.col;
            res[i.i] = F::forward(MapFunc::OMP,
                global_offset + i,
                std::get<EI>(bufs)[i.i]...,
                std::get<CI>(bufs)...,
                args...);
        }
    }
}
```

The skeleton then needs to provide some manner of actually executing the skeleton. We do this via the backendDispatch method, a pattern inherited from SkePU 2. Listing 5.7 demonstrates how we use this to invoke the execution of the cpu-method with the elementwise access pattern.
Listing 5.7: backendDispatch method of the Map skeleton.

```cpp
template<size_t arity, typename MapFunc, typename CUDAKernel, typename CLKernel>
void Map<arity, MapFunc, CUDAKernel, CLKernel>::backendDispatch(pack_indices< EI... >,
  pack_indices<A I... >,
  pack_indices<C I... >,
  Size2D size,
  MatrixIterator<T> res,
  CallArgs &&... args)
{
  auto elemwise_iterators = std::make_tuple(get< EI >(args...).begin()...);
  this->element_aligned(size,
    res,
    std::get< EI >(elemwise_iterators)...,
    get<A I>(args...)...,
    get<C I>(args...)...);
}
```

There remains little in the presented code that indicates that this is a distributed program, executing across one or more distributed matrices. Our distributed matrix data type and our high level “meta skeleton” handle such low-level issues.

### 5.5.2 Implementing the Reduce Skeleton

In SkePU 2 StarPU-MPI we added support for the `skepu::Reduce1D` skeleton. There is support for three different modes in the API:

- `skepu::ReduceMode::RowWise` Row-wise reduction of a matrix. We compute an independent reduction for each row in the matrix, yielding a vector with the results.
- `skepu::ReduceMode::ColWise` Column-wise reduction of a matrix. We compute an independent reduction for each column in the matrix, yielding a vector.
- `skepu::ReduceMode::ElWise` Elementwise reduction of a matrix. We reduce all elements into a single scalar.

To simplify the code listings, we only present code relevant to the elementwise reductions, the other modes are similar. While code exists for all these modes, only the elementwise reductions is fully functional in SkePU 2 StarPU-MPI.

The `Reduce1D::cpu` method is structurally very similar to the `Map::cpu` implementation in Listing 5.6, the major difference being that we perform a reduction instead. The larger difference is in the `backendDispatch` method. Listing 5.8 shows the implementation of this method intended for elementwise reduction in vectors. Here, we again use the element aligned access pattern from `multivariant_task`, but this time the variant that provides an extra output handle per task, for use as a partial result. We broadcast the resulting value from each task to all other tasks, then each `rank` performs a final reduction of the partial. There are most likely ways to optimize this communication pattern. Asynchronous `MPI_Put` and a barrier is one option to consider. The computational pattern is almost identical to `MPI_Allreduce`, but we decided against this method due to difficulties in integrating SkePU user functions with the MPI API for function handles. A StarPU-native method that can remain within the task graph would be preferable.

---

13Providing complete support for all modes should be straightforward, but there was no time to complete the implementation.
14Just a standard OpenMP `#pragma omp reduce` along with managing a few edge cases and all the different modes. All in all, very similar to the previous implementation in SkePU 2.
15There should not be that many partial values, in practice we can expect one per rank.
5.5. SKELETON IMPLEMENTATIONS

Listing 5.8: backendDispatch method of the Reduce1D skeleton.

```cpp
template<typename ReduceFunc, typename CUDAKernel, typename CLKernel>
template<typename Iterator>
typeName ReduceFunc::Ret Reduce1D<ReduceFunc, CUDAKernel, CLKernel>
::backendDispatch(Size2D size, Iterator arg)
{
    // This particular implementation of backendDispatch only deals with
    // elementwise reductions of vectors,
    assert(this->m_mode == skepu::ReduceMode::ElWise);
    // Launch tasks, element aligned, with partial results per task
    std::vector<skepu::cluster::starpu_var<T>> partials;
    this->element_aligned_res_per_block(size, partials, arg);
    // Schedule a broadcast of all partial results so they become locally
    // available on all ranks.
    for (auto & val : partials) {
        val.broadcast();
    }
    // Initialize with initial value, provided by application programmer
    auto result = m_start;
    for (auto & val : partials) {
        // Perform a final reduction of the partial values. 'val.get()' will
        // block until the value is available locally.
        result = ReduceFunc::CPU(result, val.get());
    }
    return result;
}
```

One important thing to note here is that the Reduce skeleton forces us to leave the task graph. The skeleton yields a single scalar value, not a smart container. In order to produce this scalar value, we need to let the task graph execute while we wait for it to yield the results we want. This means that we cannot keep scheduling more tasks until we get our result, limiting concurrency and task graph size. With the embarrassingly parallel Map-skeleton there are fewer such limitations, and we can keep extending the task graph while it executes.\[16\]

---

\[16\] With plenty of exceptions of course. Accessing any data managed by StarPU or SkePU outside of the task graph may block for full or partial execution of the task graph. The same applies to using random access container arguments.
Chapter 6

Performance Evaluation

In this chapter we aim to determine the overall performance characteristics of SkePU 2 StarPU-MPI.

6.1 Method

In order to analyze the performance of SkePU 2 StarPU-MPI, we studied performance for two different problems, the n-body problem and a simple summation to serve as a reduction. We selected these two as

- They are representative for common communication patterns in data parallel programs,
- They stress large parts of the new code in SkePU 2 StarPU-MPI,
- They are easy to write reference implementations for,
- Together, they allow us to work in both $O(n)$ and $O(n^2)$, which allow us to test a wide range of problem size parameters.
- They both represent common, predictable, computational patterns.

Additionally, related work on distributed execution of SkePU used the n-body problem, allowing for close comparison with these. While only using two benchmarks is suboptimal, together they stress a surprisingly large part of the SkePU 2 StarPU-MPI codebase. In the end, both are using our `multivariant_task` and `starpu_matrix_container`. We compared SkePU 2 StarPU-MPI to hand-written reference implementations with MPI and OpenMP.

The N-Body Problem  We selected the n-body problem with a quadratic-time algorithm, as this will put significant stress on the `allgather`-method of our `starpu_matrix_container`. This is a common communication pattern, and be representative for similar problems.

Summation Problem  The problem of summing together a vector of numbers into a single scalar is straightforward. On smaller problem sizes, issues with communication performance should be clearly visible. For larger problem sizes, we should see almost no communication overhead at all, pure computational throughput should be the limiting factor. The communication pattern used here is also commonly used, in this case functionally similar to `MPI_Allreduce`. 
6.1. METHOD

6.1.1 Performance Analysis

We used the MPI-collectives *Allgather* and *Allreduce* in the reference implementations, which should roughly align with the communications in the implementations using SkePU 2 StarPU-MPI. The two selected problems cover a wide range of concerns, from workloads severely limited by communication (reduce with a small problem size), all the way to workloads that should not be limited by communication overhead (n-body with a larger problem size, where the $O(n^2)$ calculations will overtake the communication overhead). By scaling the problem sizes we use, we should be able to bottleneck both on communication and on raw compute.

Reference Implementations

For both algorithms, we implemented functionally equivalent MPI/OpenMP codes to act as reference implementations to enable comparisons on absolute performance and not only scaling factors. The inner loops of the MPI-implementations used the same code as the user functions used for the SkePU+StarPU implementation, but using pointers or C++ containers instead of SkePU 2 StarPU-MPI containers. To avoid the compiler applying optimizations that altered the performance characteristics too much, we compiled these programs with `-O0` ¹. In the case of SkePU 2 StarPU-MPI we compiled all binaries with `-O3` ². We use `-O3` to ensure we give the compiler ample opportunity to optimize the skeleton templates. Also note that the code does not concern itself with how many OpenMP threads we use, we control that via environment variables.

Listings A.3 and A.4 show the two reference implementations.

SkePU Implementations

For the SkePU 2 StarPU-MPI benchmarks, we used a slightly newer version of the SkePU header library than what we wrote ³ as this commit fixed some issues with deadlocks. However, the changes between what we implemented and this version are fairly small and should not affect overall performance. We well note that this will technically bring the SkePU 2 StarPU-MPI implementation into very early SkePU 3 territory, but we consider the change small enough to still use the designation “SkePU 2 StarPU-MPI” for the purposes of this report. To ensure compatibility with the selected header library code, we used an early version ⁴ of the SkePU 3 precompiler. The source code to SkePU can be downloaded from repositories on GitHub.

To implement the n-body benchmark, we used a separate smart container as a doublebuffer, while the reference implementation only keeps a partial buffer on the rank. However, as the doublebuffer is a distributed smart container, the differences should be smaller than an initial reading of the code implies. In practice, while we think this could affect the end result in a measurable way, we do not consider it to be a significant advantage either way. Additionally, we consider both implementations idiomatic.

Listings A.1 and A.2 show the two SkePU 2 StarPU-MPI implementations.

---

¹ For some larger problem sizes, we are probably limited by bandwidth to primary memory, and not raw compute. For the smaller sizes of the n-body problem, everything should fit in cache.
² This shouldn’t be an issue with larger codes, but in this case, with what should be considered microbenchmarks, there are more possibilities for the compiler to add unexpected optimizations. The compiler was especially eager to apply what we presume was constant folding to the reference reduction implementation. The code is complex enough to prevent any significant constant folding. However, as we’ll see later, `-O3` may not be enough of an advantage.
³ Commit f6f403c6 by Johan Ahlqvist.
⁴ Commit a74a22c.
CHAPTER 6. PERFORMANCE EVALUATION

Benchmark Environment

We created two Apptainer \[3\] containers. The first contains just the SkePU precompiler. The second contains the StarPU run-time, a compiler suite and support libraries we need to run StarPU distributed across a cluster using Slurm as the scheduler. Relevant to performance are

- StarPU v1.3.11,
- UCX v1.13.1,
- OpenMPI v4.1.5,
- GCC v11.3.1.

We compiled and executed the reference implementations within the same environment. We based the container images on Rocky Linux \[32\], the SkePU precompiler on Rocky Linux 8 and the runtime on Rocky Linux 9. The dependencies for the specific StarPU and SkePU versions we used were easier to fulfill in this manner.

As for hardware, the benchmarks ran on two Nvidia DGX A100 \[8\] nodes. These nodes have dual socket AMD Epyc 7742 CPUs, 64 cores each. The nodes have all memory channels fully populated, totaling 2TB system memory. Each node has eight rails Mellanox HDR200 Infiniband on a non-blocking fat tree. We left the GPUs in the nodes unused. We assigned each rank 16 cores, but only allowed it to use 8 for compute\[7\]. We allowed the StarPU runtime an additional control thread, and it thus used 9 cores in total. Allocating 8 cores for compute had good reasons in the context of benchmarking the implementations. First of all, it is the largest power of two that still allows for a separate control thread on an exclusive core\[8]. The second reason was that only using half of the available cores (and a quarter of available SMT-threads) should further reduce the chance of congestion on the resources that remain shared, such as memory bandwidth. On this particular hardware, this also ensures we can schedule all threads for a rank inside a single NUMA-zone.

For the StarPU-runtime, we were careful to ensure all threads could pin to cores. For each rank, we pinned all threads to exclusive cores within one NUMA node. We confirmed that the pinning worked as expected via normal Linux monitoring tools, and also confirmed that unused cores were in fact not used. For StarPU, we used the default locality work stealing scheduler\[9]. Using UCX \[36\] as the underlying communication library for OpenMPI, we could pin each rank to a specific, exclusive, network card\[10\] and disallow shared memory communication between ranks\[11\]. This means that we force any communication between two ranks on the same node to communicate via the 200 Gb/s Infiniband fabric instead of via shared memory. While UCX is capable of utilizing multiple rails for a single rank, this was explicitly disabled in favor of assigning each rank its own interface, as to better isolate the ranks. We confirmed that we managed to pin all interfaces properly by using external network monitoring tools\[12\].

We designed this setup to allow for fewer physical nodes, while still maintaining a good approximation of 16 “small” nodes. In principle, the same design should scale out to more nodes as well, but we deemed 16 ranks to be enough. We used over-provisioned hardware and actively underutilized it, in order to better measure the characteristics of the software.

---

\[6\]Configured by supercomputer staff to have 2 logical NUMA nodes per socket instead of the native 4 nodes. Due to the hardware architecture of this particular processor, we expect any effects of this to be very small. We pinned threads to cores corresponding to a native NUMA node, further minimizing any effects. This configuration should also affect both SkePU and vanilla implementations in a similar manner.

\[7\]As configured via STARPU\_NCPUs and OMP\_NUM\_THREADS.

\[8\]For 8 ranks, we have a maximum of \(\frac{128}{8} = 16\) cores available per rank, then divide by two to allow for a control thread.

\[9\]Due to how we structure the benchmarks, the selection of scheduler should have little effect. We limit both the width and depth of the task graph. In all our cases, we only have one block of the distributed smart container local to each rank, thus we will only have one task available to run, limiting any effects of the scheduler.

\[10\]In the same NUMA-zone, but in this particular system, we need to jump through a PCle-switch to reach the NIC.

\[11\]We set the environment variable UCX\_NET\_DEVICES based on SLURM\_LOCALID and disallowed shared memory transport.

\[12\]Ultimately, we base this on port counters as reported by the mlx5 Linux driver.
Selection of Problem Parameters

We selected five problem sizes for the n-body and six for the reduction benchmarks, selecting these sizes to cover the extremes of the scaling properties. We aimed for smaller problem sizes to exhibit “bad” scaling, while aiming for large problem sizes that could exhibit “good” scaling. To allow for maximum parallelism, we selected all parameters as powers-of-two, including the number of ranks.

Measurement of Wall Time

For all implementations, we only measured the inner loop. We measured time on rank 0, barrier to barrier. For each set of parameters, we measured three times and used the median. The inner loops that performed the actual calculations ran for a set number of iterations, selected to get the measured time up into the range of a second or two for most parameter sets. We stored the times into log files, then aggregated these into CSV-files for later analysis. We ignored time spent outside of the inner loops, such as time used for startup, data initialization, heap allocations or cleanup.

6.2 Benchmarking Results

Here we present the benchmarking results for the two different problems.

6.2.1 N-Body

The measured speedups for the two implementations of the n-body problem are presented in Figures 6.1 and 6.2. Tables B.1 and B.3 show the raw data as measured. Both SkePU and the reference implementation show reasonable scaling, with the reference implementation having a slight edge when more ranks are used. However, for fewer ranks and smaller problems, the SkePU implementation shows good scaling.

6.2.2 Reduction

The measured speedups for the two implementations of the reduction problem are presented in Figures 6.3 and 6.4. Tables B.2 and B.4 show the raw data as measured. The SkePU implementation requires very large problem sizes before the communication overhead can be overcome, and only shows scaling close to linear for the 67M case.

The reduction problem at the 8M problem size was a late addition, and while the benchmark ran on machines of the same model, it did not run on the same physical machines at the same time. Thus, there could be external effects affecting measurements, for example different ambient temperatures, resulting in slightly different clock speeds. While such issues should not affect the scaling properties in a major way, we should avoid comparisons of absolute performance between the 8M case and the other problem sizes.

While the performance characteristics of such operations are interesting, one can usually amortize the overhead. These are rank-local in nature, and we wish to show the effects of communication and synchronization. Finally, some of these (e.g. startup) will happen asynchronously, making them hard to measure without also measuring external systems such as the cluster scheduler.
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.1: Measured speedup for the n-body in SkePU 2 StarPU-MPI for different number of particles.

Figure 6.2: Measured speedup for the n-body reference implementation for different number of particles.
6.2. **BENCHMARKING RESULTS**

**Figure 6.3:** Measured speedup for the reduction in SkePU 2 StarPU-MPI for different number of elements.

**Figure 6.4:** Measured speedup for the reduction reference implementation for different number of elements.
6.2.3 Relative Performance

The relative performance between the StarPU back-end to SkePU and the reference implementation, is shown for the n-body problem in Figure 6.5 and for the reduction in Figure 6.6. Here, we define relative performance as

\[
\text{Relative Performance} := \frac{\text{Reference implementation runtime}}{\text{SkePU to StarPU-MPI runtime}},
\]

for each of the rank and problem size parameters.

In the general case, for these algorithms, the implemented back-end to SkePU performs at 20 ∼ 25% of the reference implementation. In the best case, this can go above 30%. The worst case, reductions over many ranks, is very slow compared with the reference, quickly dropping below 10% of the reference performance. In the largest reduction, we see performance at a rather steady 60% of reference across all measured number of ranks. This is noteworthy, as this persists regardless of the number of ranks, and at a problem size where the computation should be the bottleneck\footnote{Bandwidth between main memory and the CPU could also be a potential bottleneck.} not communication between ranks.
6.2. BENCHMARKING RESULTS

Figure 6.5: Relative performance of the n-body problem in SkePU 2 StarPU-MPI, as compared to the reference implementation.

Figure 6.6: Relative performance of the reduction implementation in SkePU 2 StarPU-MPI, as compared to the reference implementation.
Chapter 7

Discussion

7.1 Design and Implementation

With the SkePU-2 StarPU-MPI implementation, we managed to implement part of the SkePU
API on top of StarPU, melding these two concepts together. While the initial idea was to extend
SkePU with a StarPU-based back-end, the end result is to a large extent a complete rewrite of
the SkePU library, reusing the Clang-precompiler part of the SkePU project. The SkePU code
has some flexibility with regards to implementing back-ends, algorithms and data structures.
However, in this case the “plumbing” was not flexible enough to accommodate a StarPU back-
end.

7.2 MPI

The programming model assumed by MPI and StarPU requires the application programmer to
be aware of the requirement to have coherent control flow across all ranks. The main limitation is
that StarPU requires that all ranks must build the same task graph. This requires some changes to the
SkePU API, especially surrounding any input and output. While much SkePU code should “just
work”, any code running outside of SkePU skeletons will be executed on all ranks concurrently.
For purely numerical code, this should be acceptable in most cases. However, when it comes
to I/O, the application programmer must consider the programming model carefully. SkePU 3
tries to remedy this with a SkePU::external construct, allowing the precompiler to ensure code
annotated as such only executes on a single thread by injecting appropriate synchronization
mechanisms.

As needed synchronization is done implicitly when using the SkePU API normally, application
programmers should not need to care that much. However, this should also cause unnecessary
synchronization. Reducing the amount of implicit synchronization would most likely require
that explicit synchronization is added instead. Another strategy would be to make synchroniza-
tion more granular, at the cost of implementation complexity. For example, modifying a single
element within a starpu_matrix_container currently results in a full stall of the task graph and
a full invalidation of any copies. We could limit such effects to individual handles instead.
7.3 Meta Skeleton Programming

As the amount of “boilerplate” required to implement any single SkePU skeleton on top of StarPU was fairly large and complex due to templates, with much of the program structure being shared between templates, it was decided that this needed some abstraction in the form of “meta skeletons” to reduce complexity.

As the `multivariant_task` template must be useful for implementing all skeletons exposed through the SkePU API, it must also itself implement all data access patterns and communication patterns needed to implement the SkePU API. In effect, `multivariant_task` acts as a very high level, typed, API on top of StarPU. This API could also be used as a basis for other back-ends, where `multivariant_task` is reimplemented, fully or partially, on top of another back-end, while skeletons built using `multivariant_task` can be left unmodified.

With the structure previously used by SkePU, each skeleton needed a full implementation on top of each data structure\(^1\). This implies that SkePU, for \(s\) skeletons and \(d\) data structures, requires \(O(s \times d)\) implementations of skeletons. As some code can be shared, this number is probably lower in practice. The number of data structures was reduced in this implementation, by unifying `skepu::Array` and `skepu::Matrix`, further described in Section 7.4. With `multivariant_task`, the number of skeletons are still the same, but the template exposes a structured method for sharing most of the code each skeleton needs, thus minimizing the effort in adding new skeletons. In effect, there is a relationship between `multivariant_task` and the skeleton implementations that is very similar to the relationship between a skeleton and a user function.

7.4 Unification of Data Structures

The unification of `skepu::Array` and `skepu::Matrix` in this implementation breaks the SkePU API in some ways, mostly minor. The most significant change is in how iterators work. Instead of iterating over the underlying memory region, this implementation iterates over a rectangular “view” of memory, as previously shown in Figure 5.3.

Normally, on a single node, SkePU 2 will use the same data layout for both `skepu::Matrix` and `skepu::Array\(^2\)`. In this context, where a matrix is stored in an array in memory, the semantics of iterating in order of memory location makes sense. It allows for, among other things, code sharing between skeleton and data structure implementations. However, when memory is potentially distributed across multiple MPI-ranks, the ordering of memory is usually not in any order that is semantically interesting for the application programmer. Letting iterators iterate over a rectangular window has some major benefits,

- It is a less “leaky” abstraction, as the internals of the datastructure are not presented to the application programmer using the API,
- The new semantics are potentially useful for implementing various divide and conquer algorithms, as the iterators can be used to cheaply and recursively create new views into the same data,
- The iterators form a single rectangular selection of elements in a matrix, greatly simplifying implementation\(^3\)

We are not aware of any major negative consequences of this change.

\(^1\)Currently array and matrix, with sparse arrays a possible future addition.
\(^2\)And share much of the implementation.
\(^3\)The old semantics should be possible to implement, by treating the selected region as up to three separate rectangular selections. The example in figure 5.3 would yield the regions \([0,1,2],[3,4],[5]\).
### 7.5 Implementation Complexity

Both the implementation of `multivariant_task` and the unification of SkePU's smart containers are attempts at reducing the complexity of the implementation. Some areas of the design and implementation are sources of more complexity than other, and complexity could in some cases not be avoided.

#### 7.5.1 Differing Philosophies in API-Design Between SkePU and StarPU

The API-designs of SkePU and StarPU follow very different philosophies. SkePU, to a great extent, utilize fairly complex C++ templates and attempts to respect the type system as much as possible. As much work as possible is done at compile time, so SkePU remains fairly minimal during runtime.

StarPU is written in C, and does very little at compile time. The StarPU-runtime is designed to allow for dynamic linkage, and should allow any language that can interface with the C ABI, through FFIs or otherwise, to also interface with StarPU. This means that the focus on, for example, type safety is not as pronounced in StarPU as in SkePU.

To compensate for this, the implementation needs to ensure types are cast back and forth correctly at all boundaries between SkePU and StarPU. In particular, type information must be removed when scheduling a StarPU codelet, but then recreated inside the codelet, before the codelet hands the data handles over to the user function. While this should not affect the performance much, it requires quite a lot of scaffolding during compile time. Adding to the complexity is that StarPU codelets respect the C ABI, and thus lack the concept of C++ objects, which are used by SkePU in many locations.

However, while this kind of API-mismatch increases the complexity of the implementation, there were surprisingly few instances where SkePU and StarPU present incompatible semantics in their APIs. As previously discussed, the requirement that the StarPU task graph is built identically across all ranks, requires some attention from the application programmer. Additionally, as StarPU allows for multiple ranks, some semantics around memory are significantly different, but this is more due to the programming model and underlying hardware than StarPU itself.

#### 7.5.2 Complexity From Changing Memory Models

SkePU on a single shared memory node requires very few synchronization mechanisms, as the main program flow is single threaded and the parallelism is managed by the SkePU-library transparently to the application programmer. In the implemented variant, more synchronization options are needed, and some mechanisms from MPI must be exposed to the application programmer. The StarPU task graph alleviates most of this in practice, by transparently managing data dependencies, but synchronization still need to happen. For example, fetching a single element from a container is a very cheap and simple task in normal SkePU, as it is mostly just following a pointer. On StarPU, this involves

- Waiting for the task graph to reach a certain point, on all nodes,
- Scattering the desired value to all nodes,
- Yielding the element to the application programmer.

As previous SkePU code and APIs did assume a shared memory model, many assumptions made are no longer valid. One example of this would be references to container elements, which are reasonable and useful in a shared memory context, but won’t work transparently when memory is distributed across ranks. The SkePU API takes this into account, by removing the support for indexing into containers via brackets, thus disallowing references.

The performance characteristics of some code cannot be preserved when moving into distributed contexts. In particular, with SkePU running in shared memory, especially with the OpenMP
back-end, it is essentially “free” for the application programmer to temporarily bypass SkePU and work on the contents of SkePU-containers without involving SkePU. With memory distributed across MPI-ranks, direct access to container elements go from being essentially “free” to very expensive. While this doesn’t add much complexity in the implementation, it does increase complexity and unpredictability for the application programmer.

7.6 Flexibility and Ease of Use

The implemented solution exposes an API for MPI-enabled applications at a very high level of abstraction. As long as the built-in skeletons are sufficient for expressing the desired algorithm, the ease of use should be mostly equivalent to the previous SkePU implementation. For many algorithms, MPI-enabled SkePU should prove more ergonomic than the corresponding MPI-code, and allow the programmer to work on a much higher level of abstraction. The ability to manage distributed memory in a C++-native RAII\footnote{Resource Acquisition Is Initialization} style should be particularly noteworthy.

In a shared memory model, the application programmer gets a built in escape hatch, and can work directly on the underlying data without involving SkePU. Such an “easy” escape hatch mechanism is currently missing in the StarPU based implementation, and could prove difficult to add. Allowing the application programmer to bypass SkePU and interact directly with StarPU is a possible solution, but will probably create a situation where at least some of the interface between SkePU and StarPU must become part of the public API to ensure stability. Another option would be to allow users to interact directly with\texttt{multivariant\_task} (or similar) and, in effect, write their own skeletons as part of their application.

7.7 Performance

While the implementation does not scale as linearly as the reference, it is not entirely unreasonable either, especially in the n-body benchmark. However, the reduction has somewhat underwhelming performance. This indicates a large communication overhead for this type of operation. It may be possible to optimize reductions to some extent, but getting to the same level of performance as raw MPI collective operations may prove difficult.

**Overhead** The largest reduction benchmark, with $\sim 67M$ elements, only perform at $\sim 60\%$ of the reference as shown in Figure 6.5. This holds true across all benchmarked numbers of ranks, including the single rank case, and may indicate that there is some overhead to raw compute. The communication overhead at this problem size should be almost negligible, as indicated by the scaling properties in Figure 6.1. Unnecessary copying of data or additional function calls the compiler was not able to inline could cause such performance degradation. The logic needed to support arbitrary user functions could also cause this. The reference uses simple hard-coded summation operations both for the local OpenMP-reduction and the collective MPI-operations, unlike the SkePU implementation that must support more general user functions. Another possibility would be that the more complex environment the skeletons are part of, featuring both function calls and dynamic linking, come in the way of automatic vectorization or other compiler optimizations.

**Performance Relative to Reference** The performance of SkePU 2 StarPU-MPI, relative to the reference, is not particularly good. Optimizations to gain drastically better performance should be explored and implemented before any kind of production work, as guided by profiling. SkePU 2 StarPU-MPI could have plenty of possibilities for optimizations, as any newly written piece of code. However, this would most likely require extensive in depth profiling of both the implementation and StarPU itself.
Performance Presented in Related Works  The performance characteristics we measured in the SkePU 2 StarPU-MPI implementation are similar to the results presented by the EXA2PRO project [28], Zeijlon [45] as well as Almqvist [1]. We expected this, as the benchmarked code is ultimately based on SkePU 2 StarPU-MPI. There are differences, but these may be due to differences in benchmarking methodology and differences in hardware. The high overhead to compute performance is there in all cases, as is the overhead to communication. However, we will try to explain these discrepancies to some degree. Both Almqvist and Zeijlon provide numbers for the n-body problem, allowing for some direct comparison. The EXA2PRO project does not provide any numbers we can compare directly.

Zeijlon [45] presents benchmarks on the n-body problem, but runs these at a significantly larger problem size, 64000 particles. For 16 ranks, they measure a speedup of ~ 11, which is lower than our speedup at 16 ranks and $2^{14}$ particles, ~ 16. The report does not provide in depth details on how the jobs running the StarPU-benchmarks were scheduled. The benchmarks done by Almqvist [1] on the n-body problem are similar, but use 20000 particles at a speedup of ~ 7 for 16 ranks.

With the information given, we think the following aspects contribute to the discrepancies:

- The hardware is not as over-provisioned, moving more bottlenecks from software to hardware. In particular, in our benchmarks we have a dedicated network interface per each 8 cores, on the same socket, while Zeijlon and Almqvist have one interface for 32 cores, with 16 on a different socket.

- Zeijlon discusses NUMA-locality, but not for the StarPU back-end. If the StarPU jobs were scheduled across a full node, the QPI-bus could become a bottleneck, limiting bandwidth. While we could extrapolate from Zeijlon’s other results that do attempt to measure the effects of a multi-socket system, this would not account for StarPU’s scheduling thread.

- Neither Almqvist nor Zeijlon explicate any considerations for the StarPU scheduling thread. If they did co-locate this thread on cores with active computations, this could increase scheduling latency and variance, affecting scaling properties.

The benchmarking methodology we use attempts to limit the effects that any particular hardware bottleneck has on the results, by actively under-utilizing already over-provisioned hardware. We benchmarked the performance across a range of problem sizes, differing from the ones used by other authors. In general, the different results are coherent in that there are issues in scaling and performance, but differ in their quantitative measurements. We must also conclude that the technical details in how we schedule jobs can have large effects on the end results.

Benchmarking Methodology

The use of nodes shared by multiple ranks may be a source of some smaller measurement issues, but we do not consider these to be major enough to reconsider the validity of the results. Using more benchmarks would have been beneficial, but should not change any conclusions; the results from the benchmarks we performed are clear and should extend to algorithms using similar communication patterns.

Possible Starvation  The relatively low performance of the reference reduction implementation visible in Figure 6.4 at 2 and 4 ranks for $n \in \{1024, 2048\}$ could indicate some kind of starvation or additional latency. Memory starvation is unlikely, the full set of test data, 16KiB should fit in L1-cache. It could also be that the single-rank case is significantly faster due to entirely

---

5However, we think the locality of the network interfaces are more important than the number of cores per interface. None of the n-body benchmarks should have saturated bandwidth or message rate of the hardware they used.

6In this case, two sockets Intel Xeon Gold 6130 with 16 cores.

72048 × 8 bytes for the relevant datatype.

8And in many other cases all data will fit within L3-cache.
7.8. ETHICAL CONSIDERATIONS

skipping any network communication, only needing core-complex local atomic operations, thus skewing the results.

However, while this is a discrepancy, it does not seem to affect the StarPU runtime in the same manner, as SkePU 2 StarPU-MPI shows almost linear scaling.

Compiler Optimizations To avoid issues with aggressive compiler optimizations causing unexpected performance characteristics in the reference implementations, we compiled these with most optimizations turned off. This, of course, drastically affects performance. However, communication performance should remain similar. As the idea was using the reference implementation as a performance reference, predictable characteristics were preferable to getting the highest possible performance.

7.8 Ethical Considerations

High Performance Compute require a considerate amount of resources, at a considerate cost. This includes financial, environmental, societal and opportunity cost. Inefficient use of such resources will result in unneeded costs, be it for the tax payer, future generations or other research. To not digress too much from the concerns of computer science, the discussion on how to prioritize what computational tasks should get access to computational resources will be mentioned here as an important discussion, but left as out-of-scope for this report.

Modern compute resources are complex and hard to use efficiently, needing considerable know-how and development time to develop and optimize applications. In many cases, the need for detailed knowledge on computational resources is not only limited to application developers. For some users, the computational power is only a tool to reach results within some field of research, only tangentially related with computer science. The need for efficient use of resources is in active competition with both wall time and development resources, requiring us to make some tradeoffs. But how much efficiency can we sacrifice?

Access to high level frameworks, such as what SkePU or StarPU aim to provide, give more application programmers and users the tools needed for high performance compute. In raising the abstractions to such a level, plenty of important details can be hidden below. Giving users the ability to work on a high level of abstraction should also require telling them when not to, when they should drop to lower level tools in order to allow for reasonable efficiency.

7.9 Future Work

Several areas related to SkePU, StarPU and SkePU 2 StarPU-MPI are relevant for future work.

Support for Accelerators The implementation presented in this report lacks any support for accelerators, but the implementation is built to be extended with support for both OpenCL- and CUDA-enabled accelerators. This was within the original scope of this project, but was not possible due to time constraints. Implementing such support would show if the design presented in this report is able to support such accelerators, and could unlock a lot of potential performance, as many modern HPC systems rely on accelerators for the bulk of their performance.

Auto Tuning StarPU has support for supplying multiple implementations to a codelet, dynamically tuning which implementation should be used in certain situations. This includes support for accelerators. Currently, the codelets generated by this SkePU implementation only supply a single OpenMP-based CPU-implementation. A logical next step could be to generate multiple different codelets for both CPU and accelerators and let StarPU auto tune the execution.
Escape Hatches  As mentioned in Section 7.6, the current implementation lacks any escape hatch for the application programmer. If the available skeletons are not sufficient for expressing an algorithm, an application programmer using the current implementation is out of luck and would need to extend the SkePU library itself. Identifying and implementing a safe and easy to use method of sidestepping SkePU and/or StarPU could prove beneficial for preventing any potential “showstoppers”.

API-modifications  The SkePU API could be further modified, probably with both extensions and deprecations, in order to try and provide a consistent environment for both shared and distributed memory situations. Some changes were made for this project, but some additional modifications should probably be done to ensure a good consistent experience across supported platforms.

The task graph that is built and executed with StarPU is, in some cases, limited by premature synchronization caused by the SkePU API. Any user code running outside of user functions that has a data dependency to data within a SkePU-container will most likely require large parts of the task graph to finish, generating what is essentially a pipeline stall. The API could be extended to make the underlying task graph more transparent to the user, for example with explicit synchronization mechanisms. Any such changes would unfortunately probably make the framework harder to use. Another option would be to allow for execution of serial user code within the task graph, maybe with something like a “Serial skeleton” or even an “IO-skeleton”. That way, data dependencies could be fully managed by StarPU. SkePU contains some developments towards this, using skepu::external.[12]

Alternative Metaprogramming Methods  All implementations of SkePU, and many other implementations of skeleton programming, rely heavily on some kind of metaprogramming. In the case of Skepu2, we use C++-templates. While templates are very powerful and flexible, they are unfortunately not very ergonomic and challenging to debug. For C++, templates and preprocessor macros are probably the only options. However, other languages and runtimes with more powerful metaprogramming facilities could be explored. Languages such as Zig [46] or Rust [35] could provide an environment where the metaprogramming required for powerful skeletons become easier and more flexible. Staying within C++, it may be beneficial to use features in later C++-standards.

Different Back-Ends  While it was very much possible to create a StarPU back-end to SkePU, the programming models of these two tools do have some differences. Using “raw” MPI may provide lower overhead compared to StarPU, as demonstrated by [45]. However, the author would like to suggest using OpenSHMEM [27] or NVSHMEM [22], as the memory model provided by these could prove more compatible with SkePU. This is similar to work Almqvist presented on a GPI back-end [1].

Deeper Integration With StarPU  The implemented integration with StarPU uses a fairly small part of the functionality on offer by StarPU. Modules for out-of-core storage and FFT exist, as does support for heterogenous scheduling. The integration of these into SkePU could enable a wider range of skeletons to be implemented, or just increase computational efficiency.

---

Either at all or just not efficiently enough.
Chapter 8

Conclusions

We managed to show that we can support distributed execution of SkePU skeletons across a cluster using StarPU, by implementing a StarPU back-end. While we needed to make some changes to the SkePU API, we did not need to make any drastic changes. As for performance, we can show reasonable scaling properties in some cases, but we also demonstrate large overheads.

8.1 Implementing a StarPU back-end to SkePU

The skeleton execution in the SkePU framework can be distributed across a cluster using StarPU. This required a substantial rewrite of the SkePU library, but no changes to the SkePU compiler were required. Adding support for StarPU as another back-end to SkePU, similarly to how support for CUDA and OpenCL are implemented, was not possible due to legitimate historical assumptions on the use of shared memory. We also suggest that the SkePU StarPU-MPI implementation can be extended with support for multivariant implementations, CUDA, or other environments the StarPU runtime support.

To reduce code complexity, some method to abstract over the skeletons should be used. In the implemented solution, multivariant_task as well as the unification of arrays and matrices serve this purpose. Using StarPU to shard skepu::Matrix onto participating ranks to then run the user functions distributed works. It does result in a fairly complex implementation despite the additional abstraction layers added.

The SkePU and StarPU frameworks complement each other to a great extent. The SkePU API provides a high level abstraction in front of StarPU, allowing much greater ease-of-use than using StarPU through pure C. However, at a technical level there are some discrepancies increasing implementation complexity.

8.2 Changes to the SkePU API

We could preserve much of the SkePU API in our distributed model, running on top of StarPU. The change in the semantics of iterators is a fairly small change, but will change how some existing code works. While this change is not strictly needed, it greatly reduces complexity and allows for more flexibility. The larger change was the omission of references to elements in SkePU containers. Within standard C++, we did not deem it reasonable to allow for such accesses with distributed memory. Additionally, such access patterns are expensive and would affect performance negatively.

While we preserved most of the semantics of the API, the performance characteristics of some operations will change drastically when moving to distributed memory. As such, in order to get
reasonable performance, application programmers should be prepared to make some changes to allow for high performance in a distributed system. Future changes in the API, such as the `skepu::external` added in SkePU 3 [12], could allow for a more coherent experience for application programmers, but exactly what changes would be beneficial are still unclear and should be explored further.

8.3 Overall Performance Characteristics

While we show some of the performance characteristics of a StarPU based SkePU in this report, this is far from complete and needs more research on a more fully fledged implementation. Implementing more skeletons, as well as optimizing and profiling existing code would give more complete results. Many of the benefits StarPU could provide are also left unused, for example automatic tuning and multiple implementations in a single codelet. However, the results are clear in that the general level of performance is underwhelming. Future work must overcome this performance gap. Zeijlon [45], Almqvist [1] and EXA2PRO [28] have started looking at this issue, but no large performance improvements within the StarPU-back-end to SkePU have emerged.

The benchmarks used show reasonable scaling for the n-body problem. Reductions, on the other hand, would benefit from some optimization work, as they show underwhelming performance. When comparing performance to the reference implementation, it looks like there may be some, fairly significant, overhead when applying user functions in the skeletons.

From the results we present, it is unclear how much the performance of SkePU 2 StarPU-MPI can be improved by optimizing the code and how much the performance characteristics are intrinsic to the use of StarPU along with the SkePU API and semantics.

8.4 Final Remarks

With the SkePU API providing high level abstractions for application programmers and StarPU providing a runtime for distributing computation across a cluster, the combination of them looks promising. The differences in programming models make the implementation somewhat complex, and may cause some performance overhead. The performance is, in its current shape, not ready and requires further work. A StarPU back-end to SkePU may prove a viable solution for providing high level abstractions for application programmers, given additional work on extending the implementation to the full SkePU API along with enough improvements to performance.
Bibliography


Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda, “Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs,” in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 80–89. doi: 10.1109/ICPP.2013.17

[31] Ingemar Ragnemalm, *When They Attack in packs: A course book in GPU computing*. Ragne-


Appendix A

Benchmark Code

Here we list the codes used for performance benchmarking. We have omitted some implementational details for brevity, but we show all performance sensitive code as is. We omitted or simplified the details on how we orchestrated the benchmarks, but left the hot parts of the code unchanged. For the SkePU-implementations, we based the code on the examples provided with the SkePU projects source code, available online [37].

Listing A.1: SkePU 2 StarPU-MPI benchmark code for the n-body problem.

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float m;
};

constexpr float G [[ skepu::userconstant ]] = 1;
constexpr float delta_t [[ skepu::userconstant ]] = 0.1;

// Note the random access container argument 'parr'
Particle move(skepu::Index1D index, Particle pi, const skepu::Vec<Particle> parr) {
    size_t i = index.i;
    float ax = 0.0, ay = 0.0, az = 0.0;
    size_t np = parr.size;
    for (size_t j = 0; j < np; ++j) {
        if (i != j) {
            Particle pj = parr[j];
            float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x)
                + (pi.y - pj.y) * (pi.y - pj.y)
                + (pi.z - pj.z) * (pi.z - pj.z));
            float dum = G * pi.m * pj.m / pow(rij, 3);
            ax += dum * (pi.x - pj.x);
            ay += dum * (pi.y - pj.y);
            az += dum * (pi.z - pj.z);
        }
    }
    pi.x += delta_t * pi.vx + delta_t * delta_t / 2 * ax;
    pi.y += delta_t * pi.vy + delta_t * delta_t / 2 * ay;
    pi.z += delta_t * pi.vz + delta_t * delta_t / 2 * az;
    pi.vx += delta_t * ax;
    pi.vy += delta_t * ay;
    pi.vz += delta_t * az;
    return pi;
}

Particle init(skepu::Index1D index, size_t np) {
    int s = index.i;
    int d = np / 2 + 1;
    int i = s % np;
    int j = ((s - 1) / np) % np;
```
```c
int k = ((((s - 1) / np) - j) / np;

Particle p;
p.x = i - d + 1;
p.y = j - d + 1;
p.z = k - d + 1;
p.vx = 0.0;
p.vy = 0.0;
p.vz = 0.0;
p.m = 1;

return p;
}

auto nbody_init = skepu::Map<0>(init);
auto nbody_simulate_step = skepu::Map<1>(move);

void nbody(skepu::Vector<Particle> &particles, size_t iterations) {
  size_t np = particles.size();
  skepu::Vector<Particle> doublebuffer(particles.size());

  // particle vectors initialization
  nbody_init(particles, np);

  // TIMER STARTS HERE (after proper barriers)
  for (size_t i = 0; i < iterations; i += 2) {
    nbody_simulate_step(doublebuffer, particles, particles);
    nbody_simulate_step(particles, doublebuffer, doublebuffer);
  }

  // TIMER STOPS HERE

  {/*...*/} // Store the resulting time
}
Listing A.2: SkePU 2 StarPU-MPI benchmark code for a reduction.

```cpp
size_t seq_init_impl(skepu::Index1D index) {
    return index.i;
}
auto seq_init = skepu::Map<0>(seq_init_impl);
size_t sum_impl(size_t x, size_t y) {
    return x + y;
}
auto sum_reduce = skepu::Reduce(sum_impl);
void reduce(size_t n, size_t iterations) {
    size_t n = 1024;
    const size_t iterations = 4096;
    size_t res {};
    skepu::Vector<size_t> v(n);
    seq_init(v);
    // TIMER STARTS HERE (after proper barriers)
    for(size_t i {}; i < iterations; ++i) {
        auto iter_res = sum_reduce(v);
        res += iter_res;
    }
    // TIMER STOPS HERE
    assert ( res == iterations*((n*(n-1))/2) );
    {/*...*/} // Store the resulting time
```

Listing A.3: Reference benchmark code for the n-body problem.

```c
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float m, padding;
};

constexpr float G = 1;
constexpr float delta_t = 0.1;

Particle move(size_t i, Particle pi, const Particle * parr, size_t size) {
    float ax = 0.0, ay = 0.0, az = 0.0;
    size_t np = size;
    for (size_t j = 0; j < np; ++j) {
        if (i != j) {
            Particle pj = parr[j];
            float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x) + (pi.y - pj.y) * (pi.y - pj.y) + (pi.z - pj.z) * (pi.z - pj.z));
            float dum = G * pi.m * pj.m / pow(rij, 3);
            ax += dum * (pi.x - pj.x);
            ay += dum * (pi.y - pj.y);
            az += dum * (pi.z - pj.z);
        }
    }
    pi.x += delta_t * pi.vx + delta_t * delta_t / 2 * ax;
    pi.y += delta_t * pi.vy + delta_t * delta_t / 2 * ay;
    pi.z += delta_t * pi.vz + delta_t * delta_t / 2 * az;
    pi.vx += delta_t * ax;
    pi.vy += delta_t * ay;
    pi.vz += delta_t * az;
    return pi;
}

Particle init(size_t index, size_t np) {
    int s = index;
    int d = np / 2 + 1;
    int i = s % np;
    int j = ((s - i) / np) % np;
    int k = (((s - i) / np) - j) / np;
    Particle p;
    p.x = i - d + 1;
    p.y = j - d + 1;
    p.z = k - d + 1;
    p.vx = 0.0;
    p.vy = 0.0;
    p.vz = 0.0;
    p.m = 1;
    return p;
}

void nbody(size_t np, size_t iterations) {
    int err = MPI_SUCCESS;
    MPI_Type_contiguous(8, MPI_FLOAT, &mpi_particle_type);
    MPI_Type_commit(&mpi_particle_type);
    int rank {};
    int world_size {};
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    size_t my_np = np / world_size;
    np = my_np * world_size;
    Particle * all_particles = nullptr;
    Particle * my_particles = nullptr;
    assert( MPI_Alloc_mem(sizeof(Particle) * np, MPI_INFO_NULL, &all_particles) == MPI_SUCCESS );
    assert( MPI_Alloc_mem(sizeof(Particle)*my_np, MPI_INFO_NULL, &my_particles) == MPI_SUCCESS );
```
// Initialize data
#pragma omp parallel for
for (size_t i = 0; i < my_np; ++i) {
    my_particles[i] = init(i + (rank*my_np), np);
}
err = MPI_Barrier(MPI_COMM_WORLD);
assert(err == MPI_SUCCESS);

// TIMER STARTS HERE
const auto starttime = MPI_Wtime();
for (size_t iter = 0; iter < iterations; iter++) {
    err = MPI_Allgather(my_particles,
        my_np,
        mpi_particle_type, 
        all_particles,
        my_np,
        mpi_particle_type, 
        MPI_COMM_WORLD);
    #pragma omp parallel for
    for (size_t i = 0; i < my_np; ++i) {
        size_t global_index = i + (rank*my_np);
        auto p = my_particles[i];
        my_particles[i] = move(global_index, p, all_particles, np);
    }
    err = MPI_Barrier(MPI_COMM_WORLD);
    assert(err == MPI_SUCCESS);
}
// TIMER STOPS HERE
const auto endtime = MPI_Wtime();

{/* ... */} // Store the resulting time
MPI_Free_mem(my_particles);
MPI_Free_mem(all_particles);
void reduce(size_t n, size_t iterations) {
    int rank {};
    int world_size {};
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    if (n % world_size != 0) {
        std::cout << "n % world_size != 0, refusing to continue!" << std::endl;
    }
    size_t res {};
    size_t my_size = n / world_size;
    std::vector<size_t> data(my_size);

    // Initialize data
    #pragma omp parallel for
    for (size_t i = 0; i < my_size; ++i) {
        data[i] = i + (rank * my_size);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // TIMER STARTS HERE
    const auto starttime = MPI_Wtime();
    for (size_t iter {}; iter < iterations; ++iter) {
        size_t local_sum = 0;
        #pragma omp parallel for reduction(+: local_sum)
        for (size_t i = 0; i < my_size; ++i) {
            local_sum += data[i];
        }
        size_t x;
        MPI_Allreduce(&local_sum, &x, 1, MPI_UINT64_T, MPI_SUM, MPI_COMM_WORLD);
        res += x;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    // TIMER STOPS HERE
    const auto endtime = MPI_Wtime();
    assert (res == iterations *((n*(n-1))/2));
    { /* ... */} // Store the resulting time
}
Appendix B

Raw Benchmark Measurements

Here, we present the raw measured times for our benchmark runs. All values are in seconds, and we present the values exactly as measured.

Table B.1: Measurements of the n-body problem running on SkePU 2 StarPU-MPI.

<table>
<thead>
<tr>
<th>N</th>
<th>Ranks</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>1,024</td>
<td>1</td>
<td>0.244041</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.123911</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.0767861</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0495494</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0578326</td>
</tr>
<tr>
<td>2,048</td>
<td>1</td>
<td>0.973993</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.491653</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.253659</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.142296</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.102811</td>
</tr>
<tr>
<td>4,096</td>
<td>1</td>
<td>3.89528</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.97308</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.00019</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.517224</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.281652</td>
</tr>
<tr>
<td>8,192</td>
<td>1</td>
<td>15.5674</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7.88908</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.93494</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.96966</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.02151</td>
</tr>
<tr>
<td>16,384</td>
<td>1</td>
<td>62.3743</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>31.3115</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15.6351</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.8322</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3.95084</td>
</tr>
</tbody>
</table>
Table B.2: Measurements of the reduction running on SkePU 2 StarPU-MPI.

<table>
<thead>
<tr>
<th>N</th>
<th>Ranks</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>1</td>
<td>0.0645624</td>
<td>0.0668966</td>
<td>0.0672311</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.167154</td>
<td>0.168173</td>
<td>0.170266</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.416271</td>
<td>0.435741</td>
<td>0.436364</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.914774</td>
<td>0.921579</td>
<td>0.930114</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.04096</td>
<td>2.06282</td>
<td>2.09298</td>
</tr>
<tr>
<td>16,384</td>
<td>1</td>
<td>0.096877</td>
<td>0.09957</td>
<td>0.0998933</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.176019</td>
<td>0.179411</td>
<td>0.180247</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.41</td>
<td>0.411704</td>
<td>0.427854</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.872965</td>
<td>0.90654</td>
<td>0.915595</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.97009</td>
<td>2.05819</td>
<td>2.11038</td>
</tr>
<tr>
<td>131,072</td>
<td>1</td>
<td>0.319625</td>
<td>0.320327</td>
<td>0.333137</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.282472</td>
<td>0.284214</td>
<td>0.286027</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.381673</td>
<td>0.391524</td>
<td>0.404013</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.890558</td>
<td>0.91223</td>
<td>0.917793</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.98283</td>
<td>1.98838</td>
<td>2.01614</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1</td>
<td>2.05008</td>
<td>2.05997</td>
<td>2.06333</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.15839</td>
<td>1.16412</td>
<td>1.16478</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.756398</td>
<td>0.759121</td>
<td>0.774406</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.777648</td>
<td>0.82732</td>
<td>0.858124</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.07047</td>
<td>2.13178</td>
<td>2.16576</td>
</tr>
<tr>
<td>8,388,608</td>
<td>1</td>
<td>21.4141</td>
<td>21.5212</td>
<td>21.5425</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10.9784</td>
<td>11.0138</td>
<td>11.0627</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.28265</td>
<td>5.33075</td>
<td>5.345</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.54707</td>
<td>2.60283</td>
<td>2.63674</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2.06384</td>
<td>2.06864</td>
<td>2.07215</td>
</tr>
<tr>
<td>67,108,864</td>
<td>1</td>
<td>169.672</td>
<td>169.857</td>
<td>170.233</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>85.5123</td>
<td>85.6603</td>
<td>85.7377</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>43.4026</td>
<td>43.4178</td>
<td>43.4203</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>22.6829</td>
<td>22.7557</td>
<td>22.8389</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12.5664</td>
<td>12.7634</td>
<td>12.7646</td>
</tr>
</tbody>
</table>
Table B.3: Measurements the n-body problem running on the reference implementation.

<table>
<thead>
<tr>
<th>N</th>
<th>Ranks</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>1</td>
<td>0.0420476</td>
<td>0.0420655</td>
<td>0.0423181</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.0272142</td>
<td>0.0272782</td>
<td>0.0273278</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.024661</td>
<td>0.0241967</td>
<td>0.0244407</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0122009</td>
<td>0.0123122</td>
<td>0.0132326</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0100626</td>
<td>0.0107485</td>
<td>0.0116374</td>
</tr>
<tr>
<td>2,048</td>
<td>1</td>
<td>0.167629</td>
<td>0.167908</td>
<td>0.167966</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.0902551</td>
<td>0.0904472</td>
<td>0.0905745</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.0559631</td>
<td>0.0563916</td>
<td>0.0572358</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.028785</td>
<td>0.0280727</td>
<td>0.0282414</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0183216</td>
<td>0.0185731</td>
<td>0.0193554</td>
</tr>
<tr>
<td>4,096</td>
<td>1</td>
<td>0.673212</td>
<td>0.675801</td>
<td>0.678153</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.337198</td>
<td>0.338176</td>
<td>0.341995</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.1725</td>
<td>0.173001</td>
<td>0.174348</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0857824</td>
<td>0.0882524</td>
<td>0.0909083</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.048829</td>
<td>0.0441278</td>
<td>0.0449972</td>
</tr>
<tr>
<td>8,192</td>
<td>1</td>
<td>2.68326</td>
<td>2.68478</td>
<td>2.69462</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.35039</td>
<td>1.35126</td>
<td>1.35452</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.677828</td>
<td>0.679052</td>
<td>0.679191</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.34217</td>
<td>0.341964</td>
<td>0.343746</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.174371</td>
<td>0.175299</td>
<td>0.186807</td>
</tr>
<tr>
<td>16,384</td>
<td>1</td>
<td>10.743</td>
<td>10.7532</td>
<td>10.7547</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.38002</td>
<td>5.42011</td>
<td>5.42285</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.70081</td>
<td>2.71496</td>
<td>2.72028</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.35181</td>
<td>1.35349</td>
<td>1.36084</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.676483</td>
<td>0.686419</td>
<td>0.700667</td>
</tr>
</tbody>
</table>
Table B.4: Measurements of the reduction running on the reference implementation.

<table>
<thead>
<tr>
<th>N</th>
<th>Ranks</th>
<th>Time (s)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
<td>Median</td>
<td>Max</td>
</tr>
<tr>
<td>1,024</td>
<td>1</td>
<td>0.0151276</td>
<td>0.0152729</td>
<td>0.0154843</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.021302</td>
<td>0.0207183</td>
<td>0.0211367</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.0251906</td>
<td>0.0254963</td>
<td>0.0276654</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.025804</td>
<td>0.0257727</td>
<td>0.0263852</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0289869</td>
<td>0.0292742</td>
<td>0.0293155</td>
</tr>
<tr>
<td>16,384</td>
<td>1</td>
<td>0.0354904</td>
<td>0.0355716</td>
<td>0.0356582</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.034681</td>
<td>0.0338566</td>
<td>0.0346233</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.0339025</td>
<td>0.0343782</td>
<td>0.0350125</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0311885</td>
<td>0.0322481</td>
<td>0.0351822</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0324633</td>
<td>0.0330578</td>
<td>0.0331785</td>
</tr>
<tr>
<td>131,072</td>
<td>1</td>
<td>0.18452</td>
<td>0.184574</td>
<td>0.187037</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.109759</td>
<td>0.109903</td>
<td>0.110469</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.0721972</td>
<td>0.0724244</td>
<td>0.0724386</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0541096</td>
<td>0.0543286</td>
<td>0.0571144</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.0449772</td>
<td>0.0452385</td>
<td>0.0499098</td>
</tr>
<tr>
<td>1,048,576</td>
<td>1</td>
<td>1.6082</td>
<td>1.57079</td>
<td>1.58396</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.787072</td>
<td>0.811727</td>
<td>0.812592</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.417428</td>
<td>0.422951</td>
<td>0.439343</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.222357</td>
<td>0.230027</td>
<td>0.235082</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.126414</td>
<td>0.132275</td>
<td>0.133013</td>
</tr>
<tr>
<td>8,388,608</td>
<td>1</td>
<td>12.7685</td>
<td>12.8256</td>
<td>12.8696</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6.45172</td>
<td>6.45548</td>
<td>6.47609</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.2694</td>
<td>3.27748</td>
<td>3.27759</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.65905</td>
<td>1.66964</td>
<td>1.80577</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.862201</td>
<td>0.873259</td>
<td>0.877769</td>
</tr>
<tr>
<td>67,108,864</td>
<td>1</td>
<td>102.31</td>
<td>102.076</td>
<td>103.057</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51.4731</td>
<td>51.8047</td>
<td>52.0868</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>25.886</td>
<td>25.9691</td>
<td>26.2816</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.965</td>
<td>13.0987</td>
<td>13.1154</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>6.65502</td>
<td>6.72181</td>
<td>6.73909</td>
</tr>
</tbody>
</table>