The Journal of Systems and Software 213 (2024) 112042

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Collection skeletons: Declarative abstractions for data collections ☆

Björn Franke [a], Zhibo Li [a,*], Magnus Morton [b], Michel Steuwer [c]

[a] School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, Scotland, United Kingdom
[b] Huawei Central Software Institute, 2 Semple Street, Edinburgh, EH3 8BL, Scotland, United Kingdom
[c] Institute of Software Engineering and Theoretical Computer Science, TU Berlin, Room E-N 367, Einsteinufer 17, Berlin, 10587, Germany

ARTICLE INFO

Keywords:
Containers
Collections
Data structures
Properties

ABSTRACT

Modern programming languages provide programmers with rich abstractions for data collections as part of their standard libraries, e.g., Containers in the C++ STL, the Java Collections Framework, or the Scala Collections API. Typically, these collections frameworks are organised as hierarchies that provide programmers with common abstract data types (ADTs) like lists, queues, and stacks. While convenient, this approach introduces problems which ultimately affect application performance due to users over-specifying collection data types limiting implementation flexibility. In this article, we develop Collection Skeletons which provide a novel, declarative approach to data collections. Using our framework, programmers explicitly select properties for their collections, thereby truly decoupling specification from implementation. By making collection properties explicit, immediate benefits materialise in forms of reduced risk of over-specification and increased implementation flexibility. We have prototyped our declarative abstractions for collections as a C++ library, and demonstrate that benchmark applications rewritten to use Collection Skeletons incur little or no overhead. We also show how Collection Skeletons help shielding the application developer from parallel implementation details, either by encapsulating implicit parallelism or through explicit properties that capture the requirements of parallel algorithmic skeletons. We observe performance improvements across most of the 17 benchmarks resulting from the use of Collection Skeletons before trying to parallelise those benchmarks, while also enhancing performance portability across three different hardware platforms.

1. Introduction

Collections of data items are central to many fundamental algorithms, and find use in all applications storing, processing and retrieving data. Therefore, it is not surprising that collections have been at the heart of Computer Science research since the inception of the discipline. From lists, stacks, and queues to trees, maps, and union-find data structures, a vast number of *abstract data types (ADTs)* (Liskov and Zilles, 1974) and concrete data structure implementations along with efficient algorithms for the organisation and efficient retrieval of data have been developed over the years (Cormen et al., 2001).

While individual data structures and algorithms for general data collections are well understood, there is less agreement on the relationship, or more specifically, the *hierarchy*, of different kinds of collections. The designers of collection abstractions for different programming languages have taken different approaches to organising collections in object-oriented class hierarchies. For example, the *C++ Standard Template Library (STL)* (Stepanov and Meng, 1995), the *Java Collections Framework (JCF)* (Naftalin and Wadler, 2006), and the *Scala Collections Framework (SCF)* (Odersky and Spoon, 2019) capture essentially the same collections, but in different ways. This is because existing class hierarchies and design patterns for collections are *operation-centric*, where inheritance relationships dictate the structure. Furthermore, specific non-functional requirements like the prescribed algorithmic complexity of certain operations in the C++ STL leave little choice when implementing STL containers. Despite superficial similarities between the collection hierarchies in the C++ STL, the JCF and the SCF, there are major differences between the frameworks and any programmer familiar with the fundamental concepts of one of the frameworks would need to spend significant effort to familiarise themselves with the other frameworks before becoming confident in their efficient use (Chen et al., 2020). To illustrate this, we refer to the JCF, where a *Stack* extends a *Vector*, which in turn implements a *List*, while a *LinkedList* implements both a *List* and a *Deque* interface. Fig. 1 illustrates the hierarchies for both data collections, which appear non-intuitive in

☆ Editor: Laurence Duchien.

* Corresponding author.
E-mail addresses: bfranke@inf.ed.ac.uk (B. Franke), zhibo.li@ed.ac.uk (Z. Li), magnus.morton@huawei.com (M. Morton), michel.steuwer@tu-berlin.de (M. Steuwer).

https://doi.org/10.1016/j.jss.2024.112042
Received 10 March 2023; Received in revised form 23 February 2024; Accepted 30 March 2024
Available online 1 April 2024

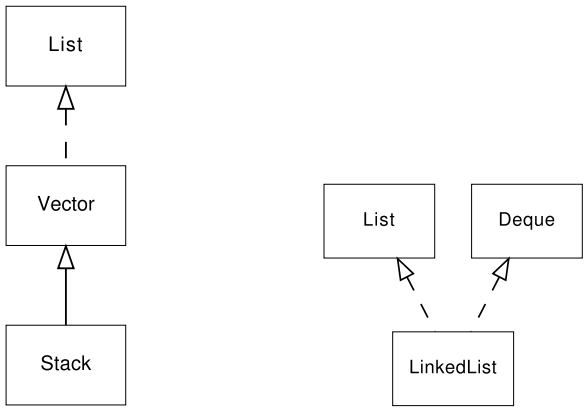


Fig. 1. Excerpt from the Java Collections Framework, showing the inheritance hierarchy for a Java Stack (left) and LinkedList (right). Motivated by object-oriented class inheritance where subclasses inherit methods from their parents, the resulting inheritance relationship fails to capture the *intuitive* collection properties. Typically, a programmer would not see a *stack* as a specialised *list*, but as a collection with the Last-In-First-Out (*LIFO*) property, and *push* and *pop* methods for access.

terms of semantic properties of the collections represented. Why would a stack be a vector or a list?

In this article, we develop a novel approach to providing user-facing abstractions for data collections. In our *Collection Skeletons* framework, programmers do not instantiate a collection of a certain class (e.g., `std::list<int>` for a list of integers), but instead, they explicitly specify properties that the collection must provide. The **key idea** is to entirely decouple specification from implementation of collections. Instead of abstract or concrete data types for their data collections, programmers *only* specify the properties their collections are relying on, thus giving the Collection Skeletons framework the flexibility to select *any* concrete implementation that provides the requested properties. For this, we distinguish between two kinds of properties concerned with (1) the *semantics* of collections, i.e., the expected behaviour, and (2) their *interfaces*, i.e., the available methods to access functionality. Although there are properties that do not belong to either semantics or interfaces, e.g., memory efficiency of a data collection, those *non-functional* properties are beyond the scope of this article and are subject of our future work. Unlike existing collection frameworks, Collection Skeletons do not attempt to fit different kinds of collections into a hierarchy, but instead we provide a single, versatile and parameterisable collection type.

Semantic properties refer to the expected behaviour of collections. For example, whether or not a collection is allowed to store duplicate entries, or if it is restricted to unique data items, is a semantic property. *Interface properties* relate to the provision of specific functions to interact with the collection, e.g., whether or not a *split-by-value* operation for splitting collections around a pivot element is provided. We discuss these properties in detail in Section 3.

Shielding collections from implementation details enables us to transparently provide *parallel* collection implementations, which do not require any changes to the application's source code. This is because the parallelism is *implicit* to the collection implementation, whereas its external, declarative interface remains untouched. Another level of *explicit* parallelism is enabled by applying parallel algorithmic skeletons (Cole, 1989) such as `map` and `reduce` to collections. In Section 5 we show how both implicit parallelism and parallel algorithmic skeletons interact with Collection Skeletons, and characterise necessary and sufficient collection properties required for a selection of parallel algorithmic skeletons.

We have prototyped our novel Collection Skeletons framework as a C++ library (see Section 4). For its evaluation, we have rewritten several benchmark applications exercising collections of different kind and nature, and executed them on three different hardware platforms (6-core Intel NUC 10, 72-core Intel Gold 6154 and 4-core Oracle Cloud Arm server). This enables us to measure any potential overheads introduced by our abstractions. In fact, we demonstrate that our Collection Skeletons introduce only negligible runtime overhead, and deliver performance improvements on several occasions. This is because our framework enables us to select an implementation different from the collection used in the original code base, which results in higher performance. We also show that the best possible implementation choice for a given collection is program- and platform-dependent. This is where the flexibility of Collection Skeletons is of clear benefit: concrete implementations can be flexibly swapped in and out without modification of the user's application code.

This article is based on an earlier conference publication (Franke et al., 2022), suitably extended to cover the interaction of Collection Skeletons and parallelism in Section 5.

1.1. Motivating example

Consider the linked-list example in Fig. 2. Listing 1 shows a code snippet with a loop traversing a user-defined linked list as it would be commonly found in C code. The user specifies a concrete implementation, i.e., a single-linked dynamically allocated list using a user-defined data type. The traversal loop performs pointer-chasing to move from element to element. Implementation choices *implicitly* define semantic properties, e.g., the order in which elements are stored and the possibility for duplicate elements. Using the C++ STL as shown in Listing 2, the user utilises a `std::list` collection data type, and the explicit pointer-chasing from Listing 1 is replaced by a range-based for loop. This notionally introduces a *list* ADT, where the concrete list implementation is hidden behind an operational list interface. The user relies on the properties implicitly provided by the *list* ADT as defined in the C++ STL. In contrast, using the Collection Skeletons approach shown in Listing 3, the user makes the properties explicit (e.g., storage order, sequentially accessible) that they request for their collection. We request the *Iterable* access property since we subsequently iterate over the elements one by one in an order specified by a separate property (*Ordered*, for insertion order). We also choose *Resizable* length as a property, since the number of elements contained in the collection is variable, i.e., not known at the time of instantiation. We also specify the collection to permit *Duplicate* entries.

We refer to the *Iterable* property as an *interface* property, since it is linked with the provision of operational facility through interface functions to iterate one by one over the collection. Furthermore, the *Ordered* property indicates that the collection is ordered in insertion order.

Our motivating example shows that Collection Skeletons provide a convenient abstraction to collection data types. They avoid tedious and non-intuitive hierarchies of collections, but instead equip the users to specify exactly the properties they need for their application. We show in our evaluation in Section 6 how this abstraction improves performance and incurs only negligible runtime overhead.

1.2. Contributions

Overall, this article makes the following contributions:

1. We develop a novel declarative approach to specifying data collections, exposing individual properties to be specified (rather than a pre-packaged sets of properties like in ADTs).
2. We identify a set of useful semantic and interface properties, which capture the key aspects of data collections that programmers care about.
3. We demonstrate how parallelism can be exploited either through transparent parallel operations provided by Collection Skeletons, or be exposed through data parallel algorithmic skeletons operating on Collection Skeletons characterised by suitable properties.

```
typedef struct nodes{
  nodes *next;
  int data;
};

for(; p; p=p->next) {
  p->data += 1;
}
```

Listing 1: User-defined list in C.

```
std::list<int> nodes;

for(auto& i : nodes) {
  i += 1;
}
```

Listing 2: List using the C++ STL.

```
Collection<int, Iterable,
    Duplicate, Resizable,
    Ordered> c;

for(auto& i : c) {
  i += 1;
}
```

Listing 3: Collection Skeletons.

Fig. 2. Motivating example showing the evolution of a linked list collection and its traversal expressed in C, C++ and the STL, and eventually using Collection Skeletons. While the C programmer employs a user-defined linked list data structure, the C++ STL offers a predefined list collection. In contrast, using Collection Skeletons a programmer requests a collection by explicitly specifying the properties they rely on for maximal implementation flexibility.

4. We evaluate a prototype C++ library implementation of our Collection Skeletons framework against legacy benchmarks rewritten to make use of our new abstraction, and demonstrate negligible performance impact across three different hardware platforms.

2. Background – algorithmic skeletons

Algorithmic skeletons are predefined components for structured parallel programming (Poldner and Kuchen, 2008), which we will later on combine with our novel Collection Skeletons. In Section 2.1 we first present the ideas underpinning structured parallel programming models and the concepts of algorithmic skeletons and parallel design patterns. We then discuss attempts at capturing and formally defining algorithmic skeletons in Section 2.2, before we give an overview of existing algorithmic skeleton programming frameworks in Section 2.3. We build on this background in Section 5 of this article where we show how algorithmic skeletons and collection skeletons naturally complement each other to intuitively capture parallel processing of data stored in collections.

2.1. Structured parallel programming, parallel design patterns and algorithmic skeletons

In software engineering, design patterns define general and reusable solutions to common programming scenarios; and parallel design patterns provide parallel versions for their corresponding design patterns (Danelutto et al., 2021). Research on parallel design patterns largely focuses structured parallel programming and parallel algorithmic skeletons.

Structured parallel programming models provide abstractions for commonly used patterns of parallel computation and interaction as parameterisable compositions. Existing frameworks, e.g., those discussed in Section 2.3, provide concrete, parallel implementations, which programmers can directly use in their applications to immediately benefit to develop parallel programs McCool et al. (2012), Loidl and Jones (1998). Structured Parallel Programming involves the selection and application algorithmic skeletons and parallel design patterns, thus hiding low-level parallel implementation details such as thread creation, communication and synchronisation primitives. The structured programming model results in easier to understand, thus more maintainable parallel code, which often also results in better parallel performance due to its pattern-specific optimisation opportunities.

Algorithmic skeletons are a high-level abstract programming paradigm that encapsulates common sequential and parallel programming patterns. This includes sequential skeletons such as *N-times repeated task* and *conditional split of task*, but also parallel skeletons such as *map* or *reduce* patterns (Cole, 1989). Typically, these high-level patterns are parameterised with code fragments, sometimes referred to as *muscle* functions, that implement the specific functionality resulting in the overall algorithm when applied according to the structure prescribed by the skeleton.

Algorithmic skeletons are reusable components, and their runtime performance can be optimised by expert programmers exploiting their well-defined behaviour. On the other hand, application programmers using algorithmic skeletons can entirely focus on the application logic and reap the performance benefits without delving into the depths of parallel programming.

Based on the data access and data flow patterns they encapsulate, parallel algorithmic skeletons can be divided into three categories (González-Vélez and Leyton, 2010),

- Data-parallel algorithmic skeletons, which capture data parallelism where the same operation is performed concurrently on a collection of data items.
- Task-parallel algorithmic skeletons, which capture a mode of computation where many different tasks are executed at the same time on the same data. *Task farms* and *pipelines* are examples of task-parallel algorithmic skeletons.
- Resolution skeletons, which use a combination of methods to solve a specified problem, e.g., *divide-and-conquer* or *branch-and-bound*.

Consider the example in Fig. 3, where a data-parallel *map* skeleton applies a function f to each element of a data collection independently. A *map* skeleton may access and compute elements within the data collection either sequentially or concurrently, but this is an implementation detail hidden from the user, who will only specify $map(f)(C)$ to invoke the provided *map* skeleton with a user-specified function f and a data collection C.

2.2. Formal definitions of algorithmic skeletons

While often introduced as an informal description or paradigm, there have been some attempts to provide more formal definitions of algorithmic skeletons.

For example, von Koch et al. (2018) defines a *map* skeleton as a simple, non-nested loop L (or *reduce* skeleton, respectively) if and only if

1. L is a data parallel loop (or parallel reduction, respectively) and
2. it does not modify its input.

There exist alternative formal definitions of algorithmic skeletons, e.g., based on a concept of commutativity (von Koch et al., 2018). Although each of the presented formalisations have some limitations,

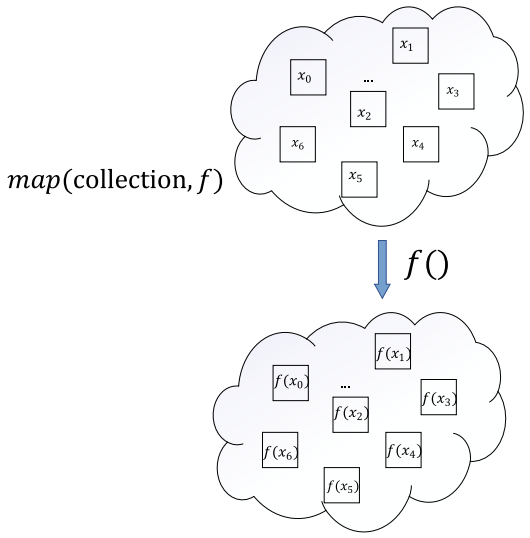


Fig. 3. A graphical representation of the data-parallel *map* skeleton.

these attempts of providing formal definitions to some of the known algorithmic skeletons aim to capture the "essence" of widely agreed informal intuitions of parallel algorithmic skeletons.

2.3. Algorithmic skeleton frameworks

There exist a wide and flexible range of programming frameworks implementing the concept of algorithmic skeletons in different ways and providing users with different interfaces. For example, algorithmic skeletons can be provided as a Domain-specific language, or a third-party library that can be externally used. Programming languages that offer algorithmic skeletons typically implement only a selected subset of these skeletons. Providing a comprehensive set could render the language too cumbersome for general-purpose computing. For a more exhaustive collection of algorithmic skeletons, numerous frameworks and libraries are available, which can be integrated as third-party libraries.

2.3.1. eSkel

The Edinburgh Skeleton Library (eSkel) is a structured parallel programming library developed with C programming language as an implementation to the original algorithmic skeletons (Benoit et al., 2005). It offers a range of algorithmic skeletons that leverage the messaging passing parallel computing model MPI (Walker and Dongarra, 1996).

2.3.2. SkePU

Compared to eSkel, SkePU is a more recent algorithmic skeletons framework (Ernstsson et al., 2021). SkePU is an open-source algorithmic skeleton framework for heterogeneous platforms, including multi-core CPUs and multi-GPU systems. The current SkePU release provides eight data-centric (data-parallel) algorithmic skeletons that can be applied to four of the generic SkePU containers. Additionally, SkePU has introduced a new backend based on MPI and StarPU (Augonnet et al., 2009) for distributed computing in cluster environments and thus broadening its applicability to diverse scenarios.

2.3.3. Munich skeleton library

The Muenster Skeleton Library (Muesli) is another implementation for algorithmic skeletons (Ciechanowicz and Kuchen, 2010). Similar to SkePU, Muesli also offers parallelism for heterogeneous clusters. Apart from data-centric algorithmic skeletons, Muesli provides implementation for task-based algorithmic skeletons such as a task farm.

Beyond these there exist several other algorithmic skeletons frameworks, e.g., Grelck (2005), Danelutto and Teti (2002), Leyton and Piquer (2010).

3. Collection skeletons

Collection Skeletons provide an alternative to the concept of ADTs (Liskov and Zilles, 1974) for the purpose of specifying data collections. ADTs are mathematical models for data types, which formalise user-facing signatures and semantics of operations on a data type. Formally, ADTs are defined by either axiomatic semantics or operational semantics of an abstract machine. Specific properties of ADTs are thus expressed through the semantics of the operations they provide, e.g., for a *stack* we might expect a definition that captures that a *pop* operation following immediately after a *push x* returns the value *x*, and that the state of the *stack* is the same as it was before. In this sense, ADTs *implicitly* define properties through the semantics of their operations. In addition, there is no notion of relationships between different ADTs. For example, users might perceive *stacks* and *queues* as related collection data types that only differ in their data retrieval order, but their respective ADTs do not attempt to establish this similarity.

Collection Skeletons follow a different approach, where all collections are derived from a single, versatile *Collection* type by means of parameterisation. The parameters of this *Collection* archetype are semantic and interface *properties*, respectively. Semantic properties relate to, e.g., whether data elements are unique or whether duplicate entries are allowed, whereas interface properties specify available access functions for the programmer to interact with the collection.

We propose eight preliminary groups of properties to help model the Collection Skeletons in our prototype library. Properties belonging to the same group exhibit similar features or a twofold symmetrical dichotomy. By combining the properties following proper rules, different kinds of data collections can be defined. Table 1 presents the eight groups of properties, their definitions, and the API parameters which are later used as abbreviations to the properties for convenient use in our prototype library.

Using these *declarative* abstractions for data collections, the end users of our framework, i.e., application programmers, define data collections only by specifying their required properties. We integrate the property-based model with modern C++ programming practice, making the learning curve for end users as smooth as possible. Our implementation employs C++ template metaprogramming, in which a collection can be defined as:

$$\texttt{Collection<T,P}_1\texttt{,P}_2\texttt{,...P}_n\texttt{,(F...)>}$$

where `T` is the elementary data type of the collection and $\texttt{P}_1$ to $\texttt{P}_n$ are property parameters from the API column of Table 1. `F...` are additional optional parameters, which we will discuss later in Section 4. Using rule-based combinations of properties, different kinds of data collections – including the standard collection ADTs found in other collection frameworks – can be specified.

3.1. Semantic properties

Semantic properties manipulate the behaviour of the methods with which collections are accessed or modified. For our minimal prototype library we have implemented the following two semantic properties:

Uniqueness. A collection only contains unique elements, i.e., there are no two elements with equal value contained in the collection. Attempts to insert a duplicate element have no effect. In Table 1, function `val()` represents the value of an element. Thus, a collection with property *Unique* does not contain two distinct elements x and y with the same value $val(x) = val(y)$. If a collection is specified with the

Table 1
Groups of properties, their definitions and API usage.

Semantic property	Definitions	API parameters
Uniqueness	$\forall x \in \mathbb{C} : \nexists y \in \mathbb{C} : x \neq y \Rightarrow val(x) = val(y)$ where x, y are elements of collection C and $val(x), val(y)$ are their values otherwise	Unique Duplicate
Circularity	$index(next(last)) = index(first)$ where $last = iterator(pos(\|C\| - 1))$ and $first = iterator(pos(0))$ otherwise	Circular Noncircular
Interface property	Definitions	API Parameters
Size	$\|\mathbb{C}\| = size$ where $size$ is a constant Otherwise (provides e.g., functions **append**, **insert**)	Fixed Resizable
Iterable	provides function **iterator** to return a forward iterator provides function **backward_iterator** to return a backward iterator provides both forward and backward iterator otherwise	Iterable ReverseIterable Bidirectional NonIterable
Accessibility	provides random access functions **get(i)**, **set(i,value)**, **c[i]** provides associative access functions **get(key)**, **set(key,value)**, **c[key]**	Random Associative
Splitability	provides functions **splitAt**, **splitBy** with $splitAt(pos(x_1), pos(x_2), ...pos(x_n)), x_n \in \mathbb{C}$ $splitBy(x_1, x_2, ...x_n), x_n \in \mathbb{C}$	NonSplit, SplitN, Splitable
UnionFind	provides function **union(c1,c2)** and **find(value)** for disjoint collection of collections	UnionFind
Hybrid property	Definitions	API Parameters
Order	Elements can be retrieved in defined insertion order $push(x) \preceq push(y) \Rightarrow y = pop() \preceq x = pop()$ $enqueue(x) \preceq enqueue(y) \Rightarrow x = dequeue() \preceq y = dequeue()$ $priority(c[0]) > priority(y) \forall y \in \mathbb{C}$ and etc. $x < y \Rightarrow pos(x) < pos(y) \forall x, y \in \mathbb{C}$ for Order by value otherwise	Ordered LIFO FIFO PriorityOrder OrderByValue Unordered

Duplicate property, then this collection can contain duplicate elements, i.e., distinct elements x and y that have the same value.

Circularity. This property has an impact on the behaviour of an iterator when used to traverse the elements of a collection. If the *Circular* property is specified, then an iterator will "wrap around" from the last position of the collection back to its first position, or vice versa. On the other hand, a collection specified with the *Noncircular* property will result in an iterator traversal to signal when it has reached the final element of the collection.

3.2. Interface properties

Interface properties are used to specify the desired availability of certain access methods for a collection. In this work we consider the following interface properties:

Size. This interface property determines whether access methods that result in a resizable collection will be provided. A collection is resizable if its size is not statically known at compile time or fixed at the time of its instantiation, i.e., its size can be changed after construction. For example, functions such as `insert` or `append` are provided for a resizable collection, whereas these data access functions are absent for fixed-size collections.

Iterable. Elements of an iterable collection can be traversed through means of an iterator, where each element is visited exactly once. This property lets the user specify whether a collection is iterable, and if so, whether forward, backward or bidirectional iterators are provided. *NonIterable* collections do not provide iterators of either kind.

Accessibility. This property specifies how elements of a collection can be accessed. We support two access modes, including random access and associative access via the *Random* and *Associative* properties, respectively. If *Random* access is specified, the collection will provide both *get* and *set* functions in addition to the `[]` operator that either takes an integer index or a key value as a parameter.

Splitability. A collection is splitable if it can be decomposed into two or more collections whose union are the elements of the original collection. We support splitting by index, i.e., elements before and after a given index position, and splitting by value, i.e., elements less or greater than a pivot value. This is encapsulated in the behaviours of functions `splitAt()` and `splitBy()`, which are provided if either *Splitable* (for two partitions) or *SplitN* (for N partitions) is requested.

UnionFind. This property is defined for collection of collections and provides `union` and `find` operations. Assuming set-like collections, this property provides a collection abstraction for disjoint sets and a function for merging collections or identifying the collection containing a specific element (Doyle and Rivest, 1976).

3.3. Hybrid properties

Some properties are of hybrid nature, i.e., they both specify access methods and also change the way operations behave semantically. An example of such a hybrid property is *order*:

Order. This property is an *interface property* because Last-in-First-Out (LIFO) or First-In-First-Out (FIFO) order provide `pop` & `push` and `enqueue` & `dequeue` operation pairs, respectively. If an operation *push(x)* immediately precedes ($\preceq$) an operation *push(y)*, then we expect two subsequent *pop()* operations to first return y and then x. However, order is also a semantic property because the specified order influences the behaviour of the iterator function `next`. Iterator-based traversals of the collection will yield different traversal orders depending on the specified order (insertion order, ordered by value, LIFO, FIFO, priority queue order, or unordered), thus changing the *semantics* of the collection's iterator access.

3.4. From concept to prototype implementation

Collection Skeletons provide a convenient and flexible way to define data collections using combinations of the desired properties in a purely declarative way. To integrate the Collection Skeletons with modern C++ programming practice and enable users to transition legacy C/C++ code, we implemented the API of the Collection Skeletons using C++ template metaprogramming, where a collection can be defined by a series of template parameters. In Section 4 we document our effort in

developing a prototype Collection Skeletons library, which allows us to evaluate the flexibility and performance impact of our new abstraction for concrete benchmark applications.

4. Library design principles

Our prototype Collection Skeletons library consists of two essential components: (i) A programming API that provides the user with a collection template class and types that correspond to collection properties, and (ii) a multi-staged pattern matching mechanism that maps declarative collections abstractions onto concrete data structure implementations.

4.1. Programming API

We use C++ template metaprogramming for the implementation of our Collection Skeletons library. However, unlike the C++ STL, which equally utilises template metaprogramming, we do not expose the underlying metaprogramming to the application programmer. We have designed the Collection Skeletons API so that it enables programmer to provide properties to the template class as type parameters. We consider it an advantage that users of our API are only exposed to limited complexity of the C++ language despite the use of C++ template metaprogramming "behind the scenes".

4.1.1. Collection properties

Fig. 4 provides an overview of the Collection interface as provided by our prototype library. We denote the property parameters enclosed in the angle brackets as the *property list*. In this API, the first type parameter `T` is the type of the elementary data to be stored by the data collection. Following `T`, P_1, P_2 to P_n are different properties of the desired data collection. These parameters of properties can be found in the column *API Parameters* in Table 1.

A special case occurs when the parameter *Fixed* is provided, i.e., the requested collection is *Fixed*. An unsigned integer constant must be provided through a non-type template parameter of type alias `size` as the fixed size of the collection.

Variadic type parameters (`F...`) are optional type parameters that may need to be applied when declaring a data collection, e.g., an *OrderByValue* data collection where the user requests a collection ordered by values as specified by a user-defined comparison function. Such a user-defined comparison function can be passed as input through the variadic type parameters `F...`. Or when an *Associative* collection is requested, the *Value* type should be declared at `F...` where the *Key* type is defined by `T`.

Table 2 summarises interface properties and their corresponding member functions. For programming convenience, we have defined a set of default methods available to all collections, which are described in Table 3.

4.1.2. Type aliases for convenience

Besides directly declaring the data collection with a given property list, users can also introduce a type alias to the property-based representation to simplify further usage such as,

```
using C1 = Collection<T,P1,P2,...Pn>
```

where `C1` is a type alias. With the type alias, the user can declare a data collection in the following program without repeatedly writing down the same complex Collection type. This would, e.g., allow the user to introduce a *set* alias for a collection with the properties of a set.

4.2. Default settings for further convenience

For ease of use, we assume in our prototype library that all the data collections are by default *Duplicate*, *Noncircular*, *Iterable*, and *Ordered*, as collections with these properties are used more frequently comparing to others with different properties, unless explicitly specified otherwise.

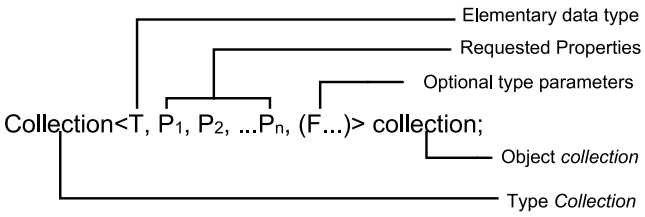


Fig. 4. Overview of the Collection interface.

4.3. Multi-staged pattern matching and mapping algorithm

Selected combinations of properties need to be checked for consistency, and mapped onto available concrete data structures for implementation. This is the purpose of the multi-staged pattern matching algorithm described in this section.

With a list of property parameters as an input according to Fig. 4, our library employs multi-staged pattern matching on the property list and eventually selects a concrete data structure providing those properties. If none of the available implementation data structures satisfies the requested properties, the compilation will be aborted with an error message. When a combination of property parameters can be satisfied by at least one concrete implementation, we call this combination of properties *eligible* and the possible implementation data structures become *eligible* implementation candidates. An overview of this process is shown in Fig. 5.

Conceptually, this process works akin to a database query: the user-provided input property list is transformed into a *query*, and our library performs a lookup over available implementation data structures and their provided properties through pattern matching. If a query can be matched, i.e., the input property list is eligible, the library returns a concrete data structure candidate and resume compilation; otherwise, if the requested properties are found to be internally inconsistent or no suitable implementation data structure that satisfies all of the requested properties can be found, compilation will be interrupted with an error message.

More details of the pattern matching process is provided in Algorithm 1. In fact, rather than directly returning the implementation data structure, we introduce another level of indirection through a dispatcher. This is for situations where more than one eligible implementation data structure can be found. Implementation candidates are wrapped in an *Intermediate Class*, which ultimately returns one of the eligible implementations. Incidentally, this intermediate class concept also provides us a convenient way to extend the Collection Skeletons, as new data structure implementations can be added without changing the programming model or any user application code.

For example, a user might request an *Ordered* and *Iterable* collection of integers as follows:

```
Collection<int, Ordered, Iterable> c
```

Both a double-linked and single-linked list meet the requirement, among many other data structures. In fact, in our prototype library, there are three concrete data structure candidates wrapped from `std::list`, `std::forward_list` and `boost::slist` available, which satisfy the requested properties. The prototype library can be easily extended for more candidate concrete data structures as long as they satisfy the properties *Ordered* and *Iterable*. The corresponding structure of the relevant intermediate class is shown in Fig. 6. This class acts as a type alias for the three template classes associated with macros. With predefined macro configurations passed through the function `FindOptimal`, a single concrete data structure is selected and returned.

In our current prototype, we resolve the selection of the implementation data structure through macros and C++ template metaprogramming at compile time, but this step could be automated and even performed adaptively at runtime, e.g., using the CollectionSwitch framework (Costa and Andrzejak, 2018).

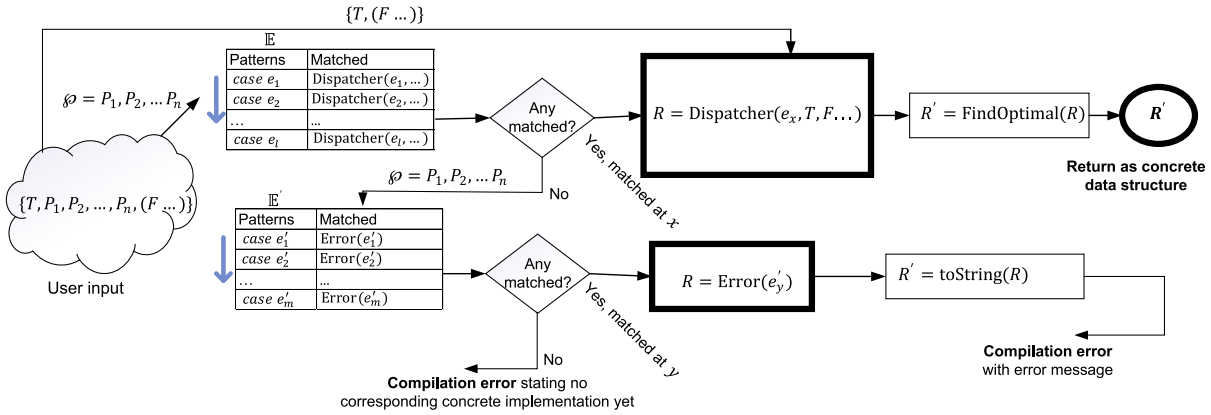


Fig. 5. Resolving properties to concrete data structure implementations through a multi-staged pattern matching and mapping algorithm.

Table 2
Interface properties and their corresponding member functions.

Properties		Guaranteed functions	Explanations
Size	Resizable	`void insert(Iter iter,T elem)`	Insert `elem` at position `iter` of a collection Not applicable for Fixed or `const` collections once initialised. "Overloaded" for Collections with different interface properties

Iterable	Iterable/
	Bidirectional
		`Iter begin()`
		`Iter end()`
		`Iter next(Iter iter)`
			`begin()` returns an iterator to the beginning of a collection
			`end()` returns an iterator to the end of a collection
			`next()` returns the (immediately) next iterator or element of the current iterator (iter)

	ReverseIterable/
	Bidirectional
		`Iter prev(Iter iter)`
		`Iter rbegin()`
		`Iter rend()`
			`prev()` returns the (immediately) previous iterator/element of the current iterator (iter)
			`rbegin()` returns a backward iterator to the beginning
			`rend()` returns a backward iterator to the end

Accessibility	Random Associative	`Iter operator[](size_t i)` `Iter operator[](Key key)`	Access the element through an index Access the element through a key
Order	FIFO	`void enqueue(T elem)` `T dequeue()`	Enqueue element Dequeue element

	LIFO	`void push(T elem)`
		`void T pop()`
		`void T top()`
			Push element
			Pop element
			Access top element without popping

	PriorityOrder	`void T front()`	Access element with highest priority
UnionFind	UnionFind	`void union(C<T> c1, c2)` `C<T> find(T elem)`	Merge disjoint collections c1 and c2 in a collection of collections Find collection **elem** is element of
Splitability	Splitable	`tuple<Iter> splitAt(Iter iter)` `tuple<Iter> splitBy(T elem)`	Split a collection around iterator `iter` Split a collection around pivot element `elem`

Table 3
Default member functions.

Default methods	Explanations
`size_t size()`	Return the size of a collection
`bool isEmpty()`	Check if a collection is empty, `true` for empty, otherwise not empty
Collection() Collection(size_t size) Collection(const Collection& c)	Default Constructor Constructor by size Copy constructor

4.3.1. Detailed discussion of the matching algorithm

The algorithm accepts the declarative properties as well as the elementary data type and other optional type parameters as input. In Algorithm 1, $\mathscr{P}$ is the set of properties **parameters** declared; T is the elementary data type to be stored in the data collection, and F... are optional type parameters that might be requested by the user. The properties in set $\mathscr{P}$ are unique, as redundant properties will be merged at an early stage properties check. From line 1 to line 5, pattern matching of the eligible combinations of property parameters is performed, where $\mathbb{E}$ is the set of all eligible patterns of eligible combinations. Each time a pattern e is checked against $\mathscr{P}$. The algorithm then decides whether a pattern matches, i.e., a matching case can be found.

After a case has been found, function `Dispatcher` in line 3 returns an **intermediate class** R based on the combination e and type parameters T, and F (if any). R stores all the information regarding the properties, types, type traits, and macros that can be used by function `FindOptimal` to help deduce the final concrete data structure. R stores templates for many concrete data structures, and more data structures can be integrated without much work. Thus, the Collection Skeletons are flexible, as one exact declaration can map to different concrete data structures, and with function `FindOptimal` we can get the optimal underlying data structure. If, however, no match is found in line 1 to line 5, then Algorithm 1 will go to the next stage — pattern matching for erroneous declarations.

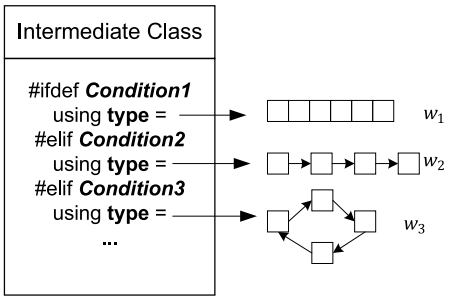


Fig. 6. Example overview of an intermediate class.

Algorithm 1: Property-based data collection deduction

Input : Properties $\mathscr{P}$=P_1,P_2...P_n,
Elementary data type T,
Other template arguments, $(F...)$
Output: The implementation or error information

```
1  for e ⟵ 𝔼 do
2      if 𝒫 ≡ e then
3          R ⟵ Dispatcher(e, T, F...)
4          R′ ⟵ FindOptimal(R)
5          return R′
6  for e′ ⟵ 𝔼′ do
7      if 𝒫 ≡ e′ then
8          R ⟵ Error (e′)
9          return compilation error toString(R)
10 return Combination toString(𝒫) not yet supported
```

Expected common user errors, such as, putting a pair of *twofold symmetrical dichotomy* properties into the same property list, make those declarations ineligible. We have encoded the error combinations in $\mathbb{E}'$ and, if a match is found, it means the input declaration cannot be resolved to a concrete data structure. Then, a function `Error` generates error information based on the pattern and stores the error information in an intermediate class for error usage, also denoted as R. After that, a compilation error is triggered with some human-readable information generated by the function `toString` called on R. After pattern matching in the error stage, if neither a concrete data structure is returned nor a compilation error is triggered in line 6 to line 9, that means the combination of properties is eligible, but no concrete data structure can be found. For example, in our prototype we have yet to support a collection that is of fixed size, ordered, and associative. A compilation error is triggered with a message stating that the input properties are not supported.

4.4. Rules of properties and API design

We have deliberately kept the programming API simple and user-friendly. However, some rules are applied to prevent non-deterministic behaviour at code generation time, or more specifically during the stage of mapping to concrete data structures.

4.4.1. The property list is order-free

Sequences of property parameters in the property list are order-free, for example,

```
Collection<T, P1, P2, P3>
```

functions exactly as

```
Collection<T, P2, P3, P1>
```

where P_1, P_2, and P_3 are different properties that together with T form an effective combination. Any permutation on properties P_1, P_2, and P_3, resolves to the same result, i.e., the same concrete data structure or the same error message.

4.4.2. Mutually exclusive properties cannot co-exist in a property list

Some properties are mutually exclusive, e.g., twofold symmetrical dichotomy properties, therefore a property list cannot have both at the same time. For example, a collection cannot be *Ordered* and *Unordered* at the same time, nor can a collection allow *Duplicate* and *Unique* elements at the same time. For example, a collection declared as

```
Collection<int, Duplicate, Unique> c
```

is not permitted. As a result, if mutually exclusive properties are declared in the property list, e.g., the above ineligible declaration, compilation terminates with an error message. These mutually exclusive cases are encoded in the error patterns as stated in line 6 to line 9 of Algorithm 1.

4.4.3. No guarantee on algebraic operations for properties

Users might expect to perform algebraic operations on properties or property lists. However, our library does not provide this facility at this stage. It is the user's responsibility to ensure consistent use of collections when extending functions or declaring their own functions that operate on the collections. For example, given two collections,

```
Collection<T, P1, P2, P3, P4> c1;
```

and

```
Collection<T1, P5, P6, F> c2;
```

where both declarations are eligible, `T` and T_1 are elementary data types, `F` is an optional template parameter. A user might want to define a `merge` function that merges the two collections `c1` and `c2`. At this stage, we do not regulate the resulting type of the merged collection when the original collections comprise different property lists. Eventually, we feel the semantic effects of such operations on distinct collection types should be resolved by the user who must specify the intended behaviour of such an operation.

4.5. Implementation of the pattern matching algorithm

As shown in Section 4.3, after receiving the declarative representation through the frontend, the pattern matching algorithm operates on the properties to determine the resulting concrete data structure. We also implemented the pattern matching algorithm based on C++ templates and type traits, thus no modification has been done on the compiler, making the prototype library more flexible.

```
template<typename T, typename... Ts>
struct contains: std::disjunction<std::is_same<T
    , Ts>...> {};
```

Listing 4: Check if a type is contained in a variadic type list.

Listing 4 presents a template class that operates on type parameters to decide if a given type `T` is contained in the variadic type list `Ts...`. Thus, a pattern matching on the type parameters can be implemented based on the helper functions in Listing 4.

```
/*type selects*/
template<typename... Args >
using selects = typename std::disjunction<Args
    ...>::type;

/*type when to decide the conditions*/
template<bool V, typename T>
struct when {
    static constexpr bool value = V;
    using type = T;
};
```

Listing 5: Compile time structures for pattern matching.

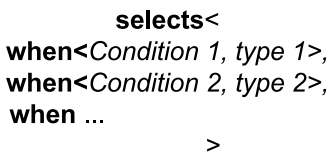


Fig. 7. Grammar of the pattern matching implementation.

Fig. 7 presents the grammar of the pattern matching implementation. In Listing 5, `selects` and `when` are the implementation of the basic pattern matching grammar - *when* the case matches, the corresponding class, i.e., the intermediate class, will be *selected*.

```
struct CollectionTypeDispatcher{
    using type = selects<
        when<contains<Random, F...>::value && !
            contains<Fixed, F...>::value,
            typename pRnd<T>::type >,
        when<contains<Associative, F...>::value
            && !contains<Random, F...>::value,
            typename pAsso<T>::type >,
        ...
        ...
    >
}
```

Listing 6: Implementation Principle of the Pattern Matching Approach.

Listing 6 shows some examples of the patterns and their corresponding intermediate classes. In Listing 6, for example, for the first `when` expression, if the condition is true, i.e., the template parameters contains *Random* and does not contain *Fixed*, then an intermediate class `pRnd` is returned as the result of the pattern matching.

```
template<typename T>
struct pRnd{
#ifdef WRAPPERSSTL
    using type = wvector;
#else
    ...
#endif
};
```

Listing 7: The Intermediate Class as a Wrapper around a Concrete Implementation Choice.

Listing 7 shows an example of the intermediate class, presenting more details for Fig. 6 mentioned before. Similarly, we implement the pattern matching for the incorrect combinations of properties in the same way except that error messages are enclosed in their corresponding intermediate classes. For the eligible combinations not yet supported by our library, we exploit `static_assert` to interrupt the compilation and return a message to the user as described in Algorithm 1.

4.5.1. Prototype implementation and extensibility

Our Collection Skeletons framework has been prototyped as C++ library, and much of the functionality is triggered at compile time, thus contributing to flexibility and runtime efficiency.

We draw on the C++ STL, Boost (Koranne, 2011) and other data structure libraries for our pool of concrete data structure implementations. Through the provided wrapping mechanism it is convenient to extend our library with additional implementations, whilst introducing only minimal runtime overhead.

In the next section of this article we extend the library with concurrent data structures and parallelism, demonstrating its potential for harnessing parallel processing in a user-friendly manner.

5. Collection skeletons and parallelism

We expect Collection Skeletons to play a role in exploiting parallelism. For this we distinguish between *implicit* and *explicit* parallelism. The key distinction here is that implicit parallelism is transparent to the end user and requires no change to the application's source code, whereas explicit parallelism is exposed to the application programmer.

We exploit implicit parallelism in operations on data collections, i.e., the application programmer specifies and uses data collections with the properties introduced earlier in this article, and our library selects concurrent data structures and parallel operations for the concrete implementation. This means that the user does not need to consider this level of parallelism at all. It appears like any other implementation scheme for Collection Skeletons, but with the difference that parallelism is exploited internally in data access functions.

On the other hand, explicit parallelism requires the application programmer to express operations over collections as algorithmic skeletons such as `map` or `reduce` (Cole, 1989). The key challenge here is to make sure that such data parallel algorithmic skeletons can interact successfully with our Collection Skeletons. This means that we need to capture the properties required by each algorithmic skeleton and match these with the properties of the data collections they are being applied to.

5.1. Implicit parallelism

Implicit parallelism disguises itself from the user of Collection Skeletons as yet another concrete implementation of a data collection. Though, individual data access functions might be implemented internally using a parallel algorithm, but this internal implementation is transparent to the application programmer. When working with our library, programmers define a data collection with the same properties they would use for a sequential implementation. Under control of a global macro, the library selects a parallel implementation using a concurrent data structure and provides parallel access functions where available. In particular, no changes to the application source code are required. For example, an integer collection of resizable length and with random access functionality would be specified as:

```
using c = Collection<int, Resizable, Random>
```

The application programmer declares a collection `c` and initialises it like this:

```
c collection{...};
```

As part of the application, the programmer might want to find an element x in the collection using the *find* function:

```
collection.find(x);
```

The application is compiled as before, however, for an implicitly parallel implementation a concurrent array or vector might be chosen by the matching and selection algorithm of our library, and a parallel implementation of the `find` function is provided, which might rely on internal multi-threading.

Using implicit parallelism the programmer is not required to be familiar with parallel programming and, in fact, they do not even need to be aware that parallelism is being used at all — this fact is entirely hidden by the Collection Skeletons library. In its current state, our prototype library relies on Intel oneTBB (Anon, 2023b), Boost lock-free containers and concurrent containers from libcds (Anon, 2023a) to provide concrete data structures. As before, these parallel data structures and operations are chosen by the matching and selection algorithm described above.

5.2. Explicit parallelism

The provision of implicit parallelism in Collection Skeletons is entirely transparent to the user: Certain operations on collections simply

Table 4
List of the benchmarks, their data structures, extracted properties and replaced implementations.

Benchmark	Source	Original data structures	Extracted properties from problem domain	Replaced (concrete)
treeadd	Olden (Carlisle, 1996)	balanced binary tree	Split2	tree2
ising	Github (Vasiladiotis, 2020)	linked list	Iterable	list
set	Rosetta (Rosseta Code Contributors, 2022)	linked list	Unique	set
libactor	Github (Fisher, 2011)	linked list	Iterable	list
tinn	Github (Louw, 2020)	array	Random	vector
shor	Github (Anon, 2017)	linked list array	Random	vector
simpleHash	Github (Lattner and Venancio, 2021)	linked list (of linked lists)	Random, Iterable	vector (of lists)
mp3	Github (Whelan, 2002)	linked list	Unique	set
scheduler	Github (Marcell, 2009)	min heap	PriorityOrder, OrderByValue	priority queue
md5	Github (Jarek, 2015)	hashmap	OrderByValue, Associative, Unordered	map
bisort	Olden (Carlisle, 1996)	binary tree	Split2	tree2
lud	Rodinia (Che et al., 2009)	array	Random	vector
kmeans	Rodinia (Che et al., 2009)	array	Random	vector
mri-q	Parboil (Stratton et al., 2012)	array	Random	vector
joseph	Github (Bhojasia, 2013)	circular linked list	Circular, Iterable	circular list
infix	Github (Diego, 2021)	queue stack	NonIterable, FIFO NonIterable, LIFO	queue stack
kruskal	Geeksforgeeks (Agrawal, 2021)	map and array	UnionFind	disjoint_set

are performed in parallel without the need for any change in the application's source code and a concurrent implementation is just another data structure implementation from the user's perspective.

However, we cannot expect Collection Skeletons to provide tailored access functions (as part of the interface properties) for every possible operation on a collection. Instead, we support algorithmic skeletons (see Section 2) as building blocks and abstractions of computational patterns in our framework. In fact, Collection Skeletons have been inspired by and named after algorithmic skeletons (Cole, 1989), which hide the complexity of parallelism in a set of parallel patterns provided as a high-level parallel programming model for parallel and distributed computing.

In this section we integrate both Collection Skeletons and algorithmic skeletons by identifying what properties are required to connect these two abstractions.

Algorithmic skeletons expose, i.e., make explicit, the parallel nature of computation as a high-level pattern such as `map` or `reduce`. A key property of algorithmic skeletons is that parallel activities such as orchestration and synchronisation are implicitly defined in the skeleton pattern, whereas the user is only concerned with the specification of the computational *muscle* function. For example, the "`map`" skeleton applies a user-provided function to each element of a collection, but leaves thread creation, scheduling, and synchronisation to the underlying implementation of the *map* skeleton.

For example, consider the following simple loop iterating over the elements of a collection and summing up its values:

```
int sum = 0;
for(int i = 0; i < size; i++){
    sum += collection[i];
}
```

Using a `for expression`, this can be rewritten as:

```
int sum = 0;
for(auto& t : collection){
    sum += t;
}
```

Now replacing the `for loop` or `for expression` with a reduction skeleton, the example can be written as follows:

```
int sum = 0;
sum = reduce(collection, [](int i, int j){return
     i+j;});
```

where we use a lambda function to represent the user-defined muscle function provided to the `reduce` algorithmic skeleton.

Parallel algorithmic skeletons provide easy access to parallelism on data collections without the need for the programmer to delve into details of the implementation. However, the question we need to answer here is what properties do our data collections need to expose in order to satisfy the requirements of the algorithmic skeletons we want to apply?

In this article we focus primarily on data-parallel algorithmic skeletons such as `map`, `reduce`, `scan`, and `zip` as they naturally operate on data collections. Task- and resolution-based algorithmic skeletons (González-Vélez and Leyton, 2010) are subject to our future work.

5.2.1. Necessary and sufficient properties

Consider the `map` skeleton shown in Fig. 8 where `map` applies a function f to each element of a collection. For this operation, the collection needs to be at least *Iterable* — we need to be able to iterate over the elements of the collection to apply f to all of the elements. Thus, we denote *Iterable* as the **necessary** property for a collection to interact with the data-parallel `map` skeleton.

However, the *Iterable* property is not enough to define a parallel `map` operation. We require additionally random access for a parallel `map` operation for simultaneous access to elements. This is not provided by the *Iterable* property, which only provides basic iteration functionality. A **sufficient** property is *Random*, which not only implies *Iterable*, but additionally provides the required parallel access functionality.

5.2.2. Inference of collection properties resulting from the application of algorithmic skeletons

In general, it is not possible to infer the properties of the collection returned by an algorithmic skeleton. For example, it depends on the function f whether the output collection contains duplicate elements or not. While it is not possible to infer the complete set of properties of the resulting collection under an algorithmic skeleton, we can at least infer *some* of the properties of the collection resulting from the application of a `map` on another collection. We know that the resulting collection is of smaller or the same size as the original collection. As the function f maps each element of the original collection to exactly one value, the size of the resulting collection is limited by the size of the original collection. However, as duplicate values might be introduced by f, a collection that does not allow duplicates, e.g., a set, the resulting collection might end up with fewer elements.

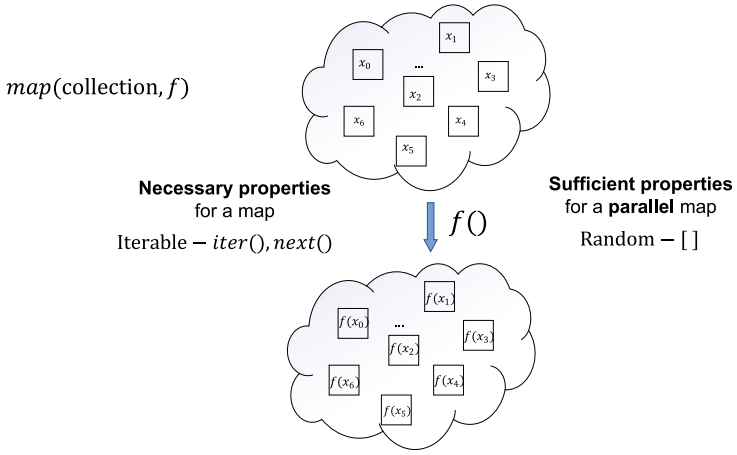


Fig. 8. Necessary and sufficient properties for a `map` skeleton applied to a collection.

Typically, we cannot make any statement on the order of the resulting collection or uniqueness of elements under a `map` without further information about the function f. Therefore, we require the user to specify the desired properties of the resulting collection, thereby following our declarative paradigm where the user is asked to state the properties they rely on.

6. Evaluation

We evaluate our Collection Skeletons library against benchmark suites including Olden (Carlisle, 1996) and other open-source projects. We have manually rewritten these legacy applications to make use of our Collection Skeletons thus allowing us to evaluate our Collection Skeletons in real-world scenarios. For this we measure any performance variation introduced by our novel abstractions in a set of before/after experiments. From Section 6.1 to Section 6.4, the evaluations are all performed in sequential mode before we configure the benchmarks and the prototype library for parallel evaluation in Section 6.5.

For the purpose of this evaluation we have manually rewritten the legacy benchmarks written in C, but have replaced the low-level user-defined data structures and their access functions with their equivalent collections from our framework. No other code changed have been performed and the same input data have been used to facilitate a like-for-like comparison. Details of the benchmarks can be found in Table 4, where we have manually identified the original data structures from the benchmarks and their properties from the problem domain. With the extracted properties, we have replaced the original data structures with the declarative counterparts from our library. Column *Replaced* are concrete data structures before wrapped and obtained by manually checking the declaration and the pattern matching rules. In column *Replaced (concrete)*, `list`, `set`, `vector`, `map`, `queue`, `stack`, and `priority_queue` are from the C++ STL, `circular list` and `disjoint_set` are from the Boost library.

We use Clang 12 and Clang++ 12 for compilation of the benchmarks, along with the "-O2" compiler flags. Besides, we enable C++ 17 compiling by setting "-std=c++17" for Clang++ 12. Experiments have been run on an Intel NUC 10 (Intel Desktop), a 72-core Intel Gold 6154 (Intel Server), and a 4-core Ampere Altra platform (Arm Server), respectively. The choice of three different target platforms serves a dual purpose: (a) we want to reassure the reader and ourselves that our approach is portable and absence of significant performance penalties is not an artefact of a specific choice of platform, and (b) we want to show that not all implementation options perform equally well on all platforms, but the flexibility of implementation choice offered by Collection Skeletons benefits performance portability across platforms.

6.1. Overall performance

Fig. 9 presents the results of the computational performance variation (speedup) for the 17 benchmarks. A speedup greater than 1.0 indicates that the programs rewritten using Collection Skeletons exhibit a performance improvement compared to the original versions; conversely, values less than 1.0 suggest a decline in computational performance. From Fig. 9, we can see that most of the replaced benchmarks have similar or better computational performance than the baseline. Some of the replaced benchmarks have much better performance than the baseline ones, suggesting that for some benchmarks, by simply replacing the data structures of the original benchmarks, the performance can be greatly improved. For some benchmarks, the speedup is not substantial, e.g., the replaced ising benchmark only ran a factor of 1.04 faster than the baseline. Since the ising is a small benchmark and does not consist of complex structures, the speedup is still important to be recognised. There are only a few benchmark programs reporting a performance decrease compared to the baseline. For example, in benchmark mri-q, where highly optimised arrays associated with manual memory management have been used in the baseline program, it is reasonable that our version using Collection Skeletons did not outperform the original one.

There are differences in speedup across the three architectures, e.g., for the *joseph* benchmark which represents a best case scenario, the speedup on the three architectures is 11.20, 16.37, and 10.36. Nearly a half of the benchmark programs report a speedup between a factor of 1.00 to 2.00, and others report greater performance boost. In general, the overall results show that the replaced versions have similar or better computational performance for almost every benchmark on the three architectures. This confirms that our Collection Skeletons library can increase the performance for almost all the benchmarks by replacing the original data structures with those hidden behind property-based declarations.

6.2. Implementation flexibility

As mentioned in Section 4, a single collection with a property list can lead to more than one concrete data structure as candidates. Here we have selected those benchmarks where this is the case, and we have evaluated the computational performance for each of the feasible concrete data structures. We have found that benchmark treeadd, bisort, ising, set, libactor, tinn, shor, simpleHash, mp3, lud, kmeans and mri-q can be implemented using different concrete data structures from our library. As the mapping and deduction process is transparent to the user, there is no need to modify any user code. We only need to modify some compile-time macros to alter the behaviour of the

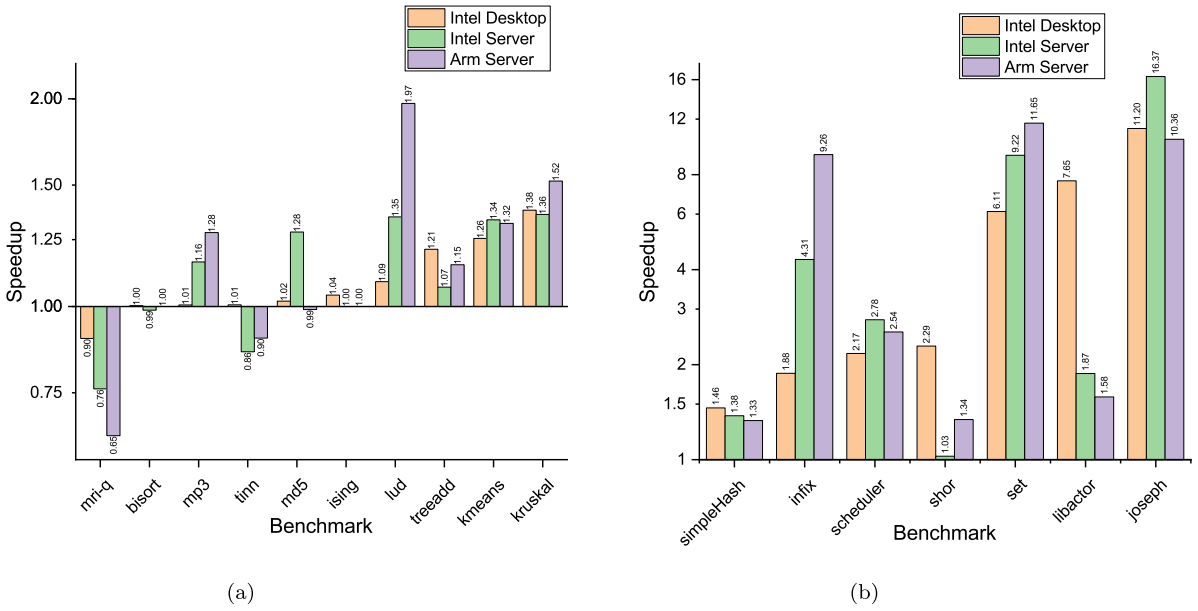


Fig. 9. Speedup of the benchmarks on three platforms.

Table 5
Optimal implementations for selected benchmarks.

Benchmark	Intel Desktop		Intel Server		Arm Server	
	Optimal	Speedup	Optimal	Speedup	Optimal	Speedup
treeadd	array_tree	1.61	array_tree	1.09	array_tree	6.37
bisort	array_tree	1.21	array_tree	1.33	array_tree	1.54
ising	slist	1.08	list	1.00	list	1.00
set	unordered_set	6.09	unordered_set	9.22	unordered_set	11.65
libactor	list	7.65	list	1.87	list	1.58
tinn	vector	1.01	vector	0.88	vector	0.9
shor	vector	2.29	vector	1.23	vector	1.34
simpleHash	forward_list	1.61	forward_list	1.44	forward_list	1.36
mp3	flat_set	1.01	set	1.16	set	1.28
lud	vector	1.09	vector	1.34	vector	1.97
kmeans	vector	1.26	vector	1.34	vector	1.32
mri-q	vector	0.90	vector	0.76	vector	0.65

function `FindOptimal` operating on intermediate classes to *swap out* the concrete implementations.

The experiments have been performed on the three platforms following the same method, and the experimental results are shown in Table 5. We can conclude that for different target platforms, the optimal concrete implementations can be different, e.g., slist from the Boost library is the optimal data structure on the Intel Desktop, while list from STL is the optimal data structure on the Intel Server. Furthermore, for some benchmarks, e.g., treeadd and bisort, the default chosen collection is a binary tree implemented through pointers, which is the de facto standard way; however, the array-based binary tree has better speedup according to Table 5.

6.3. Overhead

The prototype library has been implemented with C++ metaprogramming, meaning the process of concrete data structure deduction and substitution happens during compile time, which should introduce minimum-to-no extra overhead at runtime (Pataki, 2010). However, the compiler might optimise the code differently when it evaluates the encapsulated data structures than the concrete ones. Thus, to evaluate the overhead of the prototype library, we performed a back-to-back experiment to compare the computational time of the benchmark rewritten with the property-based declarations and that rewritten directly with the concrete implementations according to our prototype library. To replace the property-based declarations with their concrete

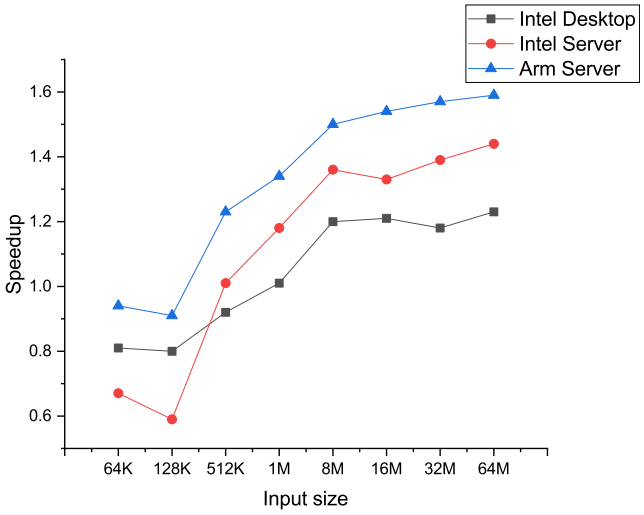
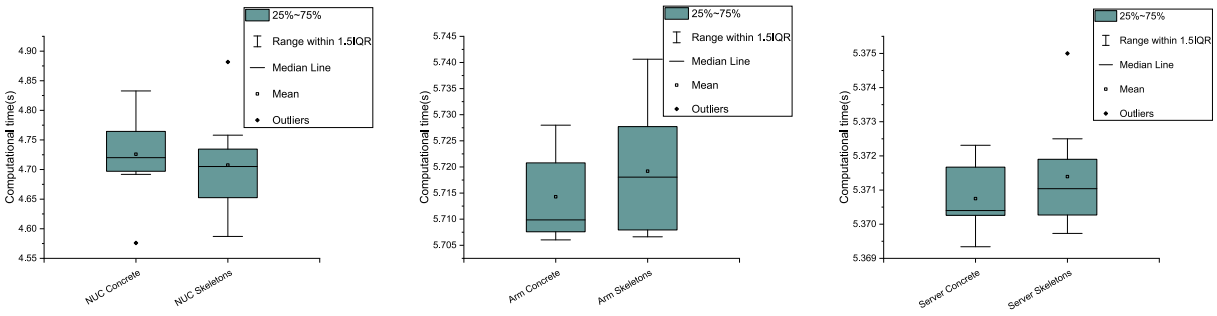


Fig. 10. Speedup by array-based binary tree for bisort regarding the input size.

implementations, we manually checked with the deduction rules to evaluate the concrete data structures. Based on the same software and hardware environment as described earlier this chapter, we ran the



(a) Intel Desktop (b) Arm Server (c) Intel Server

Fig. 11. Computational time for Collection Skeletons and concrete data structures on Intel Desktop, Arm Server and Intel Server.

ising benchmark 10 times for both configurations, and compared the mean and variance.

Fig. 11 presents the box graph of the computational time for the ising benchmark rewritten with the Collection Skeletons and with the concrete data structures, which are `std::list` in this case. From this figure it is clear that the computational time on the three platforms is close to identical, especially for the Arm Server and Intel Server where the mean, median and range are extremely close. The result should also hold for other benchmark programs as discussed earlier.

6.4. Factors influencing runtime performance

The size of the data collection, i.e., the number of elements stored in the collection, can have an impact on its performance and different architectures may offer different performance trade-offs. We further explore this for the *bisort* benchmark in Fig. 10. The performance for our three target platforms is shown for up to 64M elements, where we treat the original pointer-based implementation (`tree2`) as the baseline to evaluate the relative performance of an array-based binary tree (`array_tree2`). For up to approximately 512k elements we observe a slowdown of the array-based binary tree over the original pointer based implementation, but for larger collections, the array-based implementation outperforms the original implementation by up to a factor of 1.6. While we exercise in our experiments manual control over the final implementation through passing different macros to function `FindOptimal`, an automated technique such as "CollectionSwitch" (Costa and Andrzejak, 2018) could be employed to automate this step.

6.5. Parallelisation and performance

As discussed earlier, implicit and explicit parallelism can be applied to some benchmarks. We extended our prototype library to support concurrent data structures as candidates for concrete data structures. We further integrated the prototype library with parallelised algorithmic skeletons from third-party libraries. With the extended library, we investigated the benchmarks that have already been replaced by our serial version of Collection Skeletons whether they could benefit from parallelism. For implicit parallelism, there is no need to change the source code of any benchmark program. The property-based declarations are still the same. We simply amend the intermediate classes during code generation to allow the concrete implementation to use the concurrent data structures in the presence of the macro `ENABLE_PARALLEL`. Similarly, there is no need to alter member functions of the collections in the benchmark programs. In our extended prototype library not every property-based collection maps to a concurrent data structure, e.g., there is currently no concurrent circular data structure supported. For explicit parallelism, we firstly identified code sections that can be rewritten with a couple of data parallel algorithmic skeletons without altering the functionality and semantics of the program in the source code of the replaced sources. We focused on the for loop in every piece of source code from the benchmark programs and found that some of the for loops can be transformed to an algorithmic skeleton such as a `map` or a `reduce`.

Tables 6 and 7 summarise the performance results for our benchmarks using both implicit and explicit parallelism, respectively. In both tables, the baseline is the sequential runtime of the benchmarks expressed using Collection Skeletons as discussed before. For all benchmarks we have attempted parallelisation using both implicit and explicit parallelism at the same time. The concurrent concrete data structures used for the implicit parallelism scheme are shown in column *Details* in Table 6. The algorithmic skeletons used for explicit parallelism are show in column *Details* in Table 7.

In Table 6, `concurrent_vector`, `concurrent_priority_queue`, and `concurrent_unordered_map` are from the Intel oneAPI (Anon, 2023b). `lockfree stack` and `lockfree queue` are sourced from Boost. In Table 7, we have used skeletons based on FunctionalPlus (Hermann, 2018), Taskflow (Huang et al., 2022), and Intel oneAPI, which under the hood rely on multi-threading.

On occasion where the same program region has been parallelised using both schemes, this might result in similar or even identical speedup. For circumstance the parallelised version occur substantial performance decrease, i.e., decreasing more than a factor of 100, we do not present the speedup in the tables; instead, those results are denoted as < 0.01. Speedup results that are denoted as *N/A* mean those after-parallelised programs failed to compile, which are subject to our future research.

We observe that most of the benchmarks (13 out of 17) can be parallelised through either implicit or explicit parallelism, while three of the benchmarks directly achieve a performance gain on all of the three target platforms. Two of the benchmarks realise speedup only on the Arm Server; and one of the benchmarks reports speedup only on the Intel Desktop. While only relatively few benchmarks benefit from parallelisation, the benefit from implicit parallelism comes for free to the application programmer, where the use of algorithmic skeletons in the explicit parallelism case contributes to simpler, more declarative user code.

As part of our future work we will integrate additional concurrent data structures to provide additional scope for parallel implementation beyond the few options in our current prototype.

7. Related work

Data collections and their abstractions have received ample attention from researchers and applied software engineers alike. ADTs provide a mathematical model of data types and are defined through semantics of data access functions (Liskov and Zilles, 1974). In practice, there exists greatly different implementations of the ADTs as there are

Table 6
Parallelisation of the benchmarks (Implicit).

Benchmark	Details	Speedup Intel Desktop	Speedup Arm Server	Speedup Intel Server
ising	concurrent_vector	5.04	4.53	32.22
set	concurrent_vector	1.62	1.52	1.02
tinn	concurrent_vector	0.95	1.01	1.00
mp3	concurrent_vector	0.94	0.65	0.82
scheduler	concurrent_priority_queue	0.29	N/A	N/A
md5	concurrent_unordered_map	0.68	< 0.01	0.42
infix	lockfree stack lockfree queue	< 0.01	< 0.01	< 0.01
kruskal	concurrent_unordered_map	0.70	0.90	0.66

Table 7
Parallelisation of the benchmarks (Explicit).

Benchmark	Details	Speedup Intel Desktop	Speedup Arm Server	Speedup Intel Server
ising	for_each	5.04	4.53	32.22
libactor	for_each	1.03	0.95	1.13
tinn	map, reduce, zip	0.95	1.01	1.00
simpleHash	for_each	N/A	1.78	N/A
lud	map reduce	0.14	0.07	< 0.01
kmeans	map reduce	1.10	0.99	0.97
mri-q	map reduce zip	1.12	1.09	1.17

no universal definitions of even the most commonly used ADTs. To improve the computational or spatial performance in parallel computing, several abstractions for parallel computation based on ADTs have been proposed, e.g., Hornung and Keasler (2014), Edwards and Trott (2013), Buss et al. (2010), Ernstsson et al. (2021). However, all of these libraries require the user to acquire some deeper understanding of their detailed aspects before they can use them effectively. None of the libraries expose clearly defined properties of the data collections to the user, but these are being kept implicit.

In Marr and Daloze (2018), collection libraries from 14 programming languages are reviewed. In their work, the authors discuss how to design a collection library with multiple dimensions, including properties of collections, but without further formalising or exposing them to the user. We have taken inspiration from this work, which ultimately resulted in the approach developed in this article.

Transparently replacing and dynamically switching data collections has been proposed in several papers, e.g., Costa and Andrzejak (2018), Jung et al. (2011), Xu (2013), Costa et al. (2017), De Wael et al. (2015). These approaches generally assume that the user has already sufficiently specified their use of a data collection by making a concrete choice of a container from a collection framework, possibly subject to unspecified runtime behaviour. Then quantitative runtime data is used to select an alternative, but functionally equivalent collection with better runtime behaviour for the current scenario. These works are typically based on Java, where one data container is replaced by another.

In Bensoussan et al. (2016), a framework that uses Concern-Oriented Reuse (CORE) to capture different kinds of associations, their properties, behaviours and implementation solutions within a reusable artefact has been proposed. This work focuses on software reuse, which also partly inspired our work and, in particular, aspects on portability. An extension of this work can be found in Schöttle and Kienzle (2019), where a hierarchical Association Feature Model has been investigated. This hierarchy of data collections' features, or properties, has in inspired us considering a flatten model of the data collections through a single and versatile Collection type.

In Basios et al. (2018), data structures that share a common interface and enjoy multiple implementations as "Darwinian" data structures are proposed. Central to the concept is that data structures adjust their implementations while keeping their interfaces. A framework called ARTEMIS finds the optimal implementation of a data structure and then replaces the original one. In Wang et al. (2022) the authors propose to replace containers by analysing a synthesised complexity metric of the original containers from the source code.

Just-In-Time (JIT) data structures (De Wael et al., 2015) share some similarity with our Collection Skeletons, where data representations are decoupled from *data interfaces* as set of operations which define an ADT. These data interfaces resemble our *interface properties*, which define how data is being accessed. However, unlike our *semantic properties* the JIT data structures approach does not make explicit underlying semantic properties to the programmer. On the other hand, the framework developed in De Wael et al. (2015) allows for dynamic changes of the data representation at runtime, based on declarative input from a performance expert programmer. In contrast, our prototype implementation of Collection Skeletons finalises the implementation at compile time. This is not a fundamental restriction, though, and in principle it would be possible to support dynamic data structure changes at runtime. While JIT data structures have been evaluated specifically for matrices and lists, we aim for a larger variety of general data collections.

C++ "concepts" have been introduced in C++ 20, specifying the requirements on template arguments which can also be used to specify the properties of a template class (Thoman et al., 2022). However, in contrast with our collection skeletons, C++ concepts require the end-user to have a deep understanding of C++ metaprogramming, which is not the case for our novel abstraction library.

SkePU (Ernstsson et al., 2021) is a C++-based open-source skeleton programming framework, which aims at making parallel algorithmic skeletons easy to use. As part of this library, operands to skeleton instances are to be passed in data containers, which are STL-like, generic collection ADTs, e.g., Vector and Matrix, that encapsulate C++ array-type data. As such, SkePU containers are compatible with algorithmic skeletons, but there is no formalisation of their underlying properties. Similarly, STAPL (Majidi et al., 2019) provides customised STL-like containers (called pContainers) that distribute data across the system and provide data access operations that encapsulate the details of accessing distributed data.

To the best of our knowledge, no previous work has proposed to classify properties of data collections based on their semantics and interface functions at the programming language level. Additionally, no previous work has attempted to develop a truly declarative approach of providing the user with data collections.

8. Summary & conclusions

We have introduced a declarative approach to specifying data collections by exposing fundamental collection properties to the programmer. We show that our property-based approach to collections does not

introduce performance overhead, but instead opens up the opportunity for performance improvements gained through greater implementation flexibility, which is hidden from the application programmer. In addition, we show how parallelism can be exploited at two levels: implicit parallelism within data access operations on concurrent data structures transparent to the user, and explicit parallelism exposed through parallel algorithmic skeletons under control of the application programmer.

Benchmark applications rewritten to make use of our novel Collection Skeletons show favourable performance characteristics across different target platforms. We show that parallelism can be used to improve some of the benchmark applications. Seven of the parallel benchmark versions outperform their sequential counterparts on at least one of the three platforms.

In our future work we will continue to explore the interaction of our Collection Skeletons and algorithmic skeletons, opening up further performance benefits from parallelisation for more complex scenarios. We are planning to provide to support Collection Skeletons on heterogeneous platforms, e.g., GPUs to leverage the greater parallel performance of those platforms. We will also explore avenues to render the concrete data structure selection more adaptive to the target machine and application context, e.g., through the use of machine learning methods for the selection of an optimal implementation data structure at runtime.

CRediT authorship contribution statement

Björn Franke: Conceptualization, Funding acquisition, Supervision. **Zhibo Li:** Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Magnus Morton:** Conceptualization, Methodology, Writing – review & editing. **Michel Steuwer:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Björn Franke reports financial support was provided by Huawei-Edinburgh Joint Lab. Björn Franke reports a relationship with Huawei-Edinburgh Joint Lab that includes: funding grants.

Data availability

Data will be made available on request.

Acknowledgements

The authors would like to thank the reviewers for their constructive feedback and comments. They would also like to extend their gratitude to Chris Vasiladiotis and Murray Cole. This work was supported by a grant from Huawei-Edinburgh Joint Lab, United Kingdom. Additionally, the authors are grateful to Oracle for providing computational resources through the Cloud Starter Award, and to Intel for their support via the oneAPI Student Ambassador program.

References

Agrawal, C., 2021. Kruskal disjoint. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/, [Accessed 20-Feb-2023].

Anon, 2017. GitHub - rafa32/quantum-shor: Implementation of a quantum computer simulator together with shor's algorithm. — github.com. https://github.com/rafa32/Quantum-Shor, [Accessed 20-Feb-2023].

Anon, 2023a. GitHub - khizmax/libcds: A C++ library of concurrent data structures — github.com. https://github.com/khizmax/libcds, [Accessed 21-Feb-2023].

Anon, 2023b. Specifications: oneAPI. https://www.oneapi.io/spec/, [Accessed 20-Feb-2023].

Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A., 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (Eds.), Euro-Par 2009 Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 863–874. http://dx.doi.org/10.1007/978-3-642-03869-3_80.

Basios, M., Li, L., Wu, F., Kanthan, L., Barr, E.T., 2018. Darwinian data structure selection. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 118–128. http://dx.doi.org/10.1145/3236024.3236043.

Benoit, A., Cole, M., Gilmore, S., Hillston, J., 2005. Flexible skeletal programming with eskel. In: Cunha, J.C., Medeiros, P.D. (Eds.), Euro-Par 2005 Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 761–770.

Bensoussan, C., Schöttle, M., Kienzle, J., 2016. Associations in MDE: a concern-oriented, reusable solution. In: European Conference on Modelling Foundations and Applications. Springer, pp. 121–137. http://dx.doi.org/10.1007/978-3-319-42061-5_8.

Bhojasia, M., 2013. Joseph. https://www.sanfoundry.com/c-program-solve-josephus-problem-using-linked-list, [Accessed 20-Feb-2023].

Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N.M., Rauchwerger, L., 2010. STAPL: Standard template adaptive parallel library. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference. SYSTOR '10, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/1815695.1815713.

Carlisle, M.C., 1996. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines (Ph.D. thesis). Princeton University, USA, UMI Order No. GAX96-27387.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K., 2009. Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization. IISWC, IEEE, pp. 44–54. http://dx.doi.org/10.1109/iiswc.2009.5306797.

Chen, L., Wu, D., Ma, W., Zhou, Y., Xu, B., Leung, H., 2020. How C++ templates are used for generic programming: an empirical study on 50 open source systems. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (1), 1–49. http://dx.doi.org/10.1145/3356579.

Ciechanowicz, P., Kuchen, H., 2010. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In: 2010 IEEE 12th International Conference on High Performance Computing and Communications. HPCC, pp. 108–113. http://dx.doi.org/10.1109/HPCC.2010.23.

Cole, M.I., 1989. Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman London.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to Algorithms, second ed. The MIT Press.

Costa, D., Andrzejak, A., 2018. CollectionSwitch: A framework for efficient and dynamic collection selection. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 16–26. http://dx.doi.org/10.1145/3168825.

Costa, D., Andrzejak, A., Seboek, J., Lo, D., 2017. Empirical study of usage and performance of Java collections. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. pp. 389–400. http://dx.doi.org/10.1145/3030207.3030221.

Danelutto, M., Mencagli, G., Torquati, M., González-Vélez, H., Kilpatrick, P., 2021. Algorithmic skeletons and parallel design patterns in mainstream parallel programming. Int. J. Parallel Program. 49 (2), 177–198. http://dx.doi.org/10.1007/s10766-020-00684-w.

Danelutto, M., Teti, P., 2002. Lithium: A structured parallel programming environment in java. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (Eds.), Computational Science — ICCS 2002. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 844–853. http://dx.doi.org/10.1007/3-540-46080-2_89.

De Wael, M., Marr, S., De Koster, J., Sartor, J.B., De Meuter, W., 2015. Just-in-time data structures. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). In: Onward! 2015, Association for Computing Machinery, New York, NY, USA, pp. 61–75. http://dx.doi.org/10.1145/2814228.2814231.

Diego, 2021. GitHub - dlb04/infix-to-postfix ubinfix to postfix. — github.com. https://github.com/dlb04/infix-to-postfix, [Accessed 20-Feb-2023].

Doyle, J., Rivest, R.L., 1976. Linear expected time of a simple union-find algorithm. Inform. Process. Lett. 5 (5), 146–148. http://dx.doi.org/10.1016/0020-0190(76)90061-2, URL https://www.sciencedirect.com/science/article/pii/0020019076900612.

Edwards, H.C., Trott, C.R., 2013. Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (Xsw 2013). pp. 18–24. http://dx.doi.org/10.1109/XSW.2013.7.

Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C., 2021. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. Int. J. Parallel Program. 49 (6), 846–866. http://dx.doi.org/10.1007/s10766-021-00704-3.

Fisher, J., 2011. GitHub - airplug/libactor: Actor model library for C — github.com. https://github.com/airplug/libactor, [Accessed 20-Feb-2023].

Franke, B., Li, Z., Morton, M., Steuwer, M., 2022. Collection Skeletons: Declarative abstractions for data collections. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. In: SLE 2022, Association for Computing Machinery, New York, NY, USA, pp. 189–201. http://dx.doi.org/10.1145/3567512.3567528.

González-Vélez, H., Leyton, M., 2010. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. - Pract. Exp. 40 (12), 1135–1160. http://dx.doi.org/10.1002/spe.1026.

Grelck, C., 2005. Shared memory multiprocessor support for functional array processing in SAC. J. Funct. Programming 15 (3), 353–401. http://dx.doi.org/10.1017/S0956796805005538.

Hermann, T., 2018. GitHub - Dobiasd/FunctionalPlus: Functional Programming Library for C++. Write concise and readable C++ code. — github.com. https://github.com/Dobiasd/FunctionalPlus, [Accessed 20-Feb-2023].

Hornung, R.D., Keasler, J.A., 2014. The RAJA portability layer: Overview and status. http://dx.doi.org/10.2172/1169830.

Huang, T.-W., Lin, D.-L., Lin, C.-X., Lin, Y., 2022. Taskflow: A lightweight parallel and heterogeneous task graph computing system. IEEE Trans. Parallel Distrib. Syst. 33 (6), 1303–1320. http://dx.doi.org/10.1109/TPDS.2021.3104255.

Jarek, D., 2015. GitHub - djarek/md5lamacz: A multithreaded brute-force MD5 hashed password cracker. — github.com. https://github.com/djarek/md5lamacz, [Accessed 20-Feb-2023].

Jung, C., Rus, S., Railing, B.P., Clark, N., Pande, S., 2011. Brainy: Effective selection of data structures. ACM SIGPLAN Notices 46 (6), 86–97. http://dx.doi.org/10.1145/2345156.1993509.

Koranne, S., 2011. Boost C++ libraries. In: Handbook of Open Source Tools. Springer US, Boston, MA, pp. 127–143. http://dx.doi.org/10.1007/978-1-4419-7719-9_6.

Lattner, C., Venancio, L., 2021. Test-suite/SingleSource/Benchmarks/Shootout/hash.c at master · llvm-mirror/test-suite — github.com. https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/hash.c, [Accessed 20-Feb-2023].

Leyton, M., Piquer, J.M., 2010. Skandium: Multi-core programming with algorithmic skeletons. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing. pp. 289–296. http://dx.doi.org/10.1109/PDP.2010.26.

Liskov, B., Zilles, S., 1974. Programming with abstract data types. In: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. Association for Computing Machinery, New York, NY, USA, pp. 50–59. http://dx.doi.org/10.1145/800233.807045.

Loidl, H.-W., Jones, S.P., 1998. Algorithm+ strategy=parallelism. J. Functional Program. 8 (1), 23–60. http://dx.doi.org/10.1017/S0956796897002967.

Louw, G., 2020. GitHub - glouw/tinn: A tiny neural network library — github.com. https://github.com/glouw/tinn, [Accessed 20-Feb-2023].

Majidi, A., Thomas, N., Smith, T., Amato, N., Rauchwerger, L., 2019. Nested parallelism with Algorithmic Skeletons. In: Hall, M., Sundar, H. (Eds.), Languages and Compilers for Parallel Computing. Springer International Publishing, Cham, pp. 159–175. http://dx.doi.org/10.1007/978-3-030-34627-0_12.

Marcell, J., 2009. GitHub - jasmarc/scheduler: CPU scheduling simulator — github.com. https://github.com/jasmarc/scheduler.

Marr, S., Daloze, B., 2018. Few versatile vs. many specialized collections: how to design a collection library for exploratory programming? In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. pp. 135–143. http://dx.doi.org/10.1145/3191697.3214334.

McCool, M., Reinders, J., Robison, A., 2012. Structured Parallel Programming: Patterns for Efficient Computation, first ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Naftalin, M., Wadler, P., 2006. Java Generics and Collections: Speed Up the Java Development Process. "O'Reilly Media, Inc.".

Odersky, M., Spoon, L., 2019. The architecture of scala collections. https://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html.

Pataki, N., 2010. Testing by C++ template metaprograms. http://dx.doi.org/10.48550/arXiv.1012.0038, arXiv e-prints.

Poldner, M., Kuchen, H., 2008. Algorithmic skeletons for branch and bound. In: Filipe, J., Shishkov, B., Helfert, M. (Eds.), Software and Data Technologies. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 204–219. http://dx.doi.org/10.5220/0001315002910300.

Rosseta Code Contributors, 2022. Rosetta code — Rosetta code. https://rosettacode.org/wiki/Rosetta_Code, [Accessed 20-Feb-2023].

Schöttle, M., Kienzle, J., 2019. On the difficulties of raising the level of abstraction and facilitating reuse in software modelling: The case for signature extension. In: 2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE). pp. 71–77. http://dx.doi.org/10.1109/MiSE.2019.00018.

Stepanov, A., Meng, L., 1995. The standard template library. Tech. Rep. HPL-95-11(R.1), HP Laboratories.

Stratton, J.A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G.D., Hwu, W.-m.W., 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. In: Center for Reliable and High-Performance Computing. 127, p. 27.

Thoman, P., Tischler, F., Salzmann, P., Fahringer, T., 2022. The celerity high-level API: C++20 for accelerator clusters. Int. J. Parallel Program. 50 (3–4), 341–359. http://dx.doi.org/10.1007/s10766-022-00731-8.

Vasiladiotis, C., 2020. Mischung-suite/programs/ising/data/original_source/ising.c at master · compor/mischung-suite — github.com. https://github.com/compor/mischung-suite/blob/master/programs/ising/data/original_source/ising.c, [Accessed 20-Feb-2023].

von Koch, T.J.K.E., Manilov, S., Vasiladiotis, C., Cole, M., Franke, B., 2018. Towards a compiler analysis for parallel algorithmic skeletons. In: Proceedings of the 27th International Conference on Compiler Construction. In: CC 2018, Association for Computing Machinery, New York, NY, USA, pp. 174–184. http://dx.doi.org/10.1145/3178372.3179513.

Walker, D.W., Dongarra, J.J., 1996. MPI: a standard message passing interface. Supercomputer 12, 56–68.

Wang, C., Yao, P., Tang, W., Shi, Q., Zhang, C., 2022. Complexity-guided container replacement synthesis. Proc. ACM Program. Lang. 6 (OOPSLA1), http://dx.doi.org/10.1145/3527312.

Whelan, T., 2002. Mp3. https://sourceforge.net/projects/mp3reorg/files/mp3reorg/, [Accessed 20-Feb-2023].

Xu, G., 2013. CoCo: Sound and adaptive replacement of Java collections. In: European Conference on Object-Oriented Programming. Springer, pp. 1–26. http://dx.doi.org/10.1007/978-3-642-39038-8_1.

Björn Franke is a Professor in the School of Informatics at the University of Edinburgh, where he is a member of the Institute for Computing Systems Architecture (ICSA), the Compiler and Architecture Design (CArD) group, the ARM Centre of Excellence and the Edinburgh Huawei Lab. Björn is broadly interested in software transformation, mostly for performance optimisation and parallelisation, but he has also looked at various techniques to, e.g., reduce code size on embedded processors.

Zhibo Li is a Research Assistant in the School of Informatics at the University of Edinburgh. He just submitted his Ph.D. thesis and is waiting for the viva. His research interests include System & Architecture, especially in Data-Centric Parallelism, Compilers, and Programming Models. He is currently working on a Property-based Collection Skeletons library under the supervision of Prof. Björn Franke and Dr. Michel Steuwer. Before his Ph.D. studies, Zhibo earned both his Master's and Bachelor's degrees in Computer Science and Technology from South China University of Technology, where he was supervised by Prof. Kejing He.

Magnus Morton is a Principal Engineer of Programming Languages Research at Huawei Edinburgh Research Centre. He is interested in Compilers, Parallelisation, dynamic program analysis and transformation, and accelerators. Before joining Huawei, he worked as a postdoctoral researcher in the School of Informatics at the University of Edinburgh. He received his Ph.D. in Computer Science from the University of Glasgow in 2018.

Michel Steuwer is a Professor at Technische Universität Berlin and leads the chair of Compilers and Programming Languages. He is a member of the Institute of Software Engineering and Theoretical Computer Science in the Faculty IV – Electrical Engineering and Computer Science at TU Berlin. Before joining TU Berlin, Michel was a lecturer (assistant professor) in the School of Informatics at the University of Edinburgh, a lecturer at the School of Computing Science at the University of Glasgow and a postdoctoral researcher at the University of Edinburgh. He received his Ph.D. from the University of Münster in Germany.