Composing Algorithmic Skeletons to Express High-Performance Scientific Applications

Mani Zandifar
Texas A&M University
College Station, Texas 77843
mazaninfardi@cse.tamu.edu

Mustafa Abdul Jabbar
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
mustafa.abduljabbar@kaust.edu.sa

Alireza Majidi
Texas A&M University
College Station, Texas 77843
majidi@cse.tamu.edu

David Keyes
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
david.keyes@kaust.edu.sa

Nancy M. Amato
Texas A&M University
College Station, Texas 77843
amato@cse.tamu.edu

Lawrence Rauchwerger
Texas A&M University
College Station, Texas 77843
rwerger@cse.tamu.edu

ABSTRACT

Algorithmic skeletons are high-level representations for parallel programs that hide the underlying parallelism details from program specification. These skeletons are defined in terms of higher-order functions that can be composed to build larger programs. Many skeleton frameworks support efficient implementations for stand-alone skeletons such as `map`, `reduce`, and `zip` for both shared-memory systems and small clusters. However, in these frameworks, expressing complex skeletons that are constructed through composition of fundamental skeletons either requires complete reimplementation or suffers from limited scalability due to required global synchronization. In the STAPL Skeleton Framework, we represent skeletons as parametric data flow graphs and describe composition of skeletons by point-to-point dependencies of their data flow graph representations. As a result, we eliminate the need for reimplementation and global synchronizations in composed skeletons. In this work, we describe the process of translating skeleton-based programs to data flow graphs and define rules for skeleton composition. To show the expressivity and ease of use of our framework, we show skeleton-based representations of the NAS EP, IS, and FT benchmarks. To show reusability and applicability of our framework on real-world applications we show an N-Body application using the FMM (Fast Multipole Method) hierarchical algorithm. Our results show that expressivity can be achieved without loss of performance even in complex real-world applications.

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

DOI: http://dx.doi.org/10.1145/2751205.2751241.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable Libraries*

Keywords

algorithmic skeletons, patterns, high-performance computing, distributed systems, data flow programming

1. INTRODUCTION

The increase in availability of high-performance computing systems has made the need for expressive and efficient methods of parallel program specification more evident. Many parallel libraries and frameworks currently provide easier methods for expressing parallel applications [6, 3, 4, 25, 15, 22]. However, most of these studies are tied to specific programming models, and programs expressed using such methods are not reusable in other libraries and frameworks.

Algorithmic skeletons [9], on the other hand, tackle this issue by separating program specification from parallel implementation. These higher-level representations, which are usually defined using polymorphic higher-order functions, represent common interaction patterns prevalent in parallel programs [24]. `map`, `zip`, and `reduce` are examples of these patterns. The functional representation of skeletons provides opportunities for formal analysis and transformations to be applied on parallel programs, regardless of their parallel implementations [24]. Providing efficient parallel implementations for these fundamental skeletons in a skeleton framework can simplify parallel programming. Algorithm developers focus on the computation of an application and leave the parallelism details to be handled by the skeleton framework developers. Moreover, the functional representation of skeletons allows large and complex applications to be composed from these building blocks.

Several skeleton frameworks provide efficient implementations of fundamental skeletons for both shared-memory systems and small clusters [12]. However, in these frameworks, skeleton composition is either not considered at all, requiring reimplementation of composed skeletons, or is a BSP (Bulk Synchronous Parallel) execution of each skeleton in a composition followed by a global synchronization

Operators	elem, repeat, compose
Skeletons	allgather, allreduce, alltoall, allgather, bitreversal, broadcast, butterfly, copy, fft, gather, inner-product, map, reduce, pointer-jumping, reverse-butterfly, tree<k>, zip<k>, zip-reduce<k> reverse-tree<k>, scan, scatter, transpose-2d, transpose-3d, wavefront-2d, wavefront-3d,

Table 1: Provides skeletons and compositional operators.

at each step [18, 20]. Even if these synchronizations do not play a large role in the performance of applications on lower core counts, they can degrade performance at a larger scale.

The STAPL Skeleton Framework resolves these outstanding issues. In this framework, we use parametric data flow graphs as our internal representation of skeletons. This representation allows programs to run efficiently on parallel systems, regardless of their size. We provide efficient implementations for various fundamental skeletons, as listed in Table 1, from which larger programs can be composed. We translate composition of skeletons by point-to-point dependencies between their corresponding data flow graphs. Using point-to-point dependencies, instead of global synchronizations, allows programs written in terms of skeletons to scale better. To show expressivity we present skeleton-based implementation of the NAS EP, IS, and FT benchmarks. An N-Body application using the FMM hierarchical algorithm shows the reusability of skeletons in a real-world application. Our framework can be easily extended either through composition of skeletons or by adding data flow graph representations for new skeletons. Furthermore, our framework can be ported to other libraries with data flow engines.
Our contributions in this paper are as follows:

- a novel representation for skeleton composition as point-to-point dependencies of data flow graphs that avoids global synchronization
- a scalable and extensible skeleton framework that is closed under composition, and respects the formal definition of skeleton composition
- experimental results that show expressivity can be achieved without loss of performance

2. RELATED WORK

Since the the first appearance of skeleton-based programming in [9], several skeleton libraries have been introduced. Some of these libraries support skeleton composition either at the specification level or at the implementation level. Examples are SkeTo [18], Müesli [16], JaSkel [10], SKiPPER [26], and Eden [17].

SkeTo [18] is a C++ skeleton framework implemented on top of MPI that provides parallel skeletons for parallel data structures. SkeTo allows successive invocation of elementary skeletons, but requires global synchronization between skeleton invocations due to the Bulk Synchronization Parallel model it implements. Our framework defines composition in terms of point-to-point dependencies, avoiding global synchronization costs.

Muesli [16] is a C++ skeleton library that is built for both shared and distributed memory, implemented on top of OpenMP and MPI. It provides a variety of task parallel and data parallel skeletons, but the composition of skeletons is restricted. The creation of a new composition requires reimplementation, as it cannot be defined directly from existing skeletons.

JaSkel [10] is a Java skeleton framework that provides `farm`, `pipe` and `heartbeat` skeletons. JaSkel provides separate implementations of each skeleton for shared and distributed memory systems, making it difficult to extend. Moreover, composition of skeletons in this framework requires reimplementation, limiting its generality.

SKiPPER [26] is a skeleton library developed in Caml. SKiPPER allows composition of its limited set of skeletons, e.g., `map`, `reduce`, etc. Theses skeletons are tailored towards vision applications and can work only for shared-memory systems. The compositions in this framework are tailored for specific purposes and cannot be generalized and extended.

Eden [17] is another skeleton framework which is intended to be an extension of Haskell. Similar to other functional languages, Eden allows non-strict functions, demand driven evaluation and monads. Eden provides methods for composing and nesting existing skeletons, and allows new skeletons to be added. The main drawback of Eden is that it cannot compete with high performance computing skeleton frameworks that are developed in languages like C and C++. In our framework we present the same expressivity level as Eden, while preserving performance.

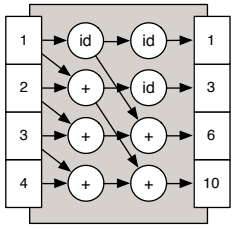
3. THE STAPL SKELETON FRAMEWORK

We have developed the STAPL Skeleton Framework, a C++ skeleton framework, which provides a wide variety of skeletons (Table 1) and allows new skeletons to be composed from these skeletons. In addition, new fundamental skeletons can be quickly created if needed. We provide multiple implementations for many skeletons, such as the `alltoall` and `allreduce` skeletons, allowing tunability of programs.

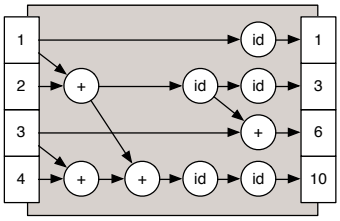
In our framework, skeletons are represented as parametric data flow graphs that are expanded when input data is provided. Since data flow graphs express available parallelism explicitly, our skeleton implementations can, by definition, run in parallel. This abstraction allows us to hide parallelism details from the specification level of skeletons. In our previous work [29], we showed how these fine-grain representations of skeletons are transformed to coarser-grain data flow graphs to run efficiently on distributes systems. In this work, we focus on skeleton composition and its translation to point-to-point dependencies. We show examples of applications that are composed from many skeletons without requiring global synchronization or reimplementation of the composed skeletons.

To facilitate the composition of data flow graphs, we encapsulate the internal representation of skeletons by defining input and output ports. Ports only expose the external interface of skeletons, as seen in skeletons formal definition, hiding internal complexity. For example, the 1D ports of a `scan` skeleton allows `scan` to be implemented in many ways (Fig. 1(a) and 1(b)), while keeping its external interface unique.

In our framework, skeleton compositions are defined statically, allowing the strong typing system of C++ to ensure the correctness of skeleton compositions. The type inference rules of these operators are given in Table 2.



(a) `scan` using a simultaneous binomial tree.



(b) `scan` using a binomial tree.

Figure 1: Ports of the `scan` skeleton (prefix sum) hide its internal implementation and expose only its 1D input and output interface. $scan(+)[1,2,3,4] = [1,3,6,10]$.

$$\frac{S : P \to Q \quad p : P}{S(p) : Q}\ \text{SKELETON} \qquad \frac{S : P \to Q}{elem_{span}(S) : \{P\} \to \{Q\}}\ \text{ELEM}$$

$$\frac{size : integer \quad S : P \to P}{repeat_{size}(S) : P \to P}\ \text{REPEAT}$$

$$\frac{S_1 : P \to R \quad S_2 : R \to Q}{compose(S_1, S_2) : P \to Q}\ \text{COMPOSE}$$

Table 2: The inference rules for the composition operators.

4. SKELETON COMPOSITION

Since skeletons are defined in terms of higher-order functions, they are by definition composable. Many applications can be expressed as a composition of fundamental skeletons. For example, `map-reduce`, a frequently used skeleton, can easily be expressed as a composition of the `map` and the `reduce` skeletons. Although defining skeletons composition is relatively simple at the specification level, translating them to efficient parallel implementations is not as straightforward. This issue has limited the usability of many skeleton frameworks [12] to very small applications.

In the STAPL Skeleton Framework we use data flow graphs as our internal representation of skeletons. This representation allows composition to be defined in terms of point-to-point dependencies between the data flow graph representation of skeletons. Therefore, we eliminate the need for global synchronizations in algorithms that are defined as compositions of other skeletons, and allow communication and computation to be overlapped.

In this section, we describe the process of generating such data flow graphs from their fine-grain dependence relations using our composition operators: `elem`, `repeat`, and `compose`. The interfaces for these operators are given Table 3.

We use `inner-product` as our example. Formally, this skeleton is defined as a composition of a `zip` and a `reduce` operators ($\texttt{reduce}(+) \circ \texttt{zip}(\times)$). First, we explain how these two fundamental skeletons are created using the `elem` and `repeat` operators from their *parametric dependence relations* which we refer to as *parametric dependencies* for brevity. We then show how these two skeletons are composed to represent the data flow graph representation of the `inner-product` skeleton (Fig. 2).

4.1 Parametric Dependencies

To build data flow graph representations for fundamental skeletons, we use their finest-grain dependence relations, which we refer to as *parametric dependencies*. A parametric dependency defines the relation between input elements of a skeleton and its output as a parametric coordinate mapping and an operation. For example, in a `zip` skeleton with a binary operator, this relation is defined as:

$$\begin{aligned} &zip_{<2>}(\oplus)[a_1 \dots a_n][b_1 \dots b_n] = [a_1 \oplus b_1 \dots, a_n \oplus b_n] \\ &zip\_pd_{<2>}(\oplus) \equiv \{(i,i) \mapsto (i), \oplus\} \end{aligned} \quad (1)$$

In other words, the value of element i of the output of this skeleton is computed by applying the binary $\oplus$ operator on the ith element of each of the inputs. At runtime, parametric dependencies are translated to data flow graph nodes in the execution process.

$S ::=$	$elem < span > (S \mid PD)$
	$\mid repeat(S, size)$
	$\mid compose < portmaps > (S')$
$S' ::=$	$S\ S' \mid \epsilon$

Table 3: The grammar for the compositional operators.

4.2 elem

The `elem` operator (Table 3) is an operator that expands a parametric dependency over a subdomain of the given inputs, called the `span`. By default, `span` is defined as the full domain of the inputs. We use a *tree-span* later in this section as an example of a non-default case.

The `elem` operator is used to describe the data flow graph representation of the $\texttt{zip}_{<k>}$ skeleton from its parametric dependency (Fig.2 (a)):

$$zip_{<k>}(\oplus) = elem(zip\_pd_{<k>}(\oplus)) \quad (2)$$

The `map` skeleton, which is a specialization of the $\texttt{zip}_{<k>}$ skeleton with k equals 1, can be expressed similarly.

4.2.1 Ports

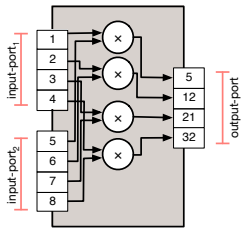
As mentioned earlier, ports encapsulate the internal implementation that a skeleton represents and only expose the external interface of a skeleton. For a skeleton implemented using the `elem` operator, we define a default port that allows access to every element in the domain of the given `span`. As can be seen in Fig. 2(a), in a $\texttt{zip}_{<2>}$ skeleton the two input ports (*input-port*$_1$ and *input-port*$_2$), indicate that the `zip` skeleton can accept two inputs with 1D domains. Similarly, its output port allows access to the result of applying multiplication on every pair of input with 1D domains. The values we are showing in Fig. 2 are given for better readability and are not stored in ports.

4.3 repeat

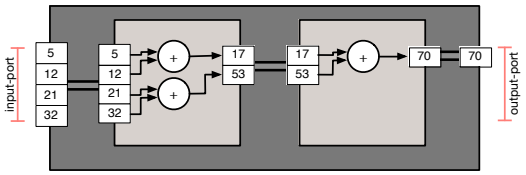
Many data-parallel skeletons, such as `reduce` and `scan`, can be defined as tree-based or multilevel data flow graphs. Our `repeat` operator (Table 3) allows these skeletons to be expressed simply as such. The `repeat` operator successively applies a given skeleton on a given input, a given number of times (determined at runtime) to produce the results.

We use this operator to define the binary-tree representation of the `reduce` skeleton. First, we define the levels of the tree using the `elem` operator and the parametric dependency of the `reduce` skeleton:

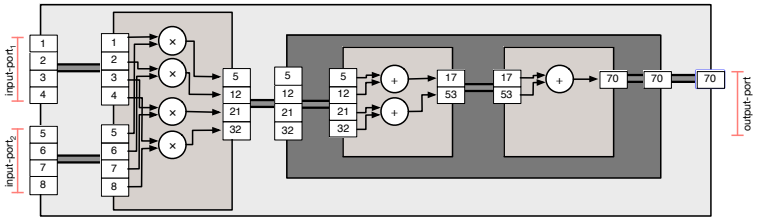
$$reduce\_pd(\otimes) \equiv \{((2i, j-1), (2i+1, j-1)) \mapsto (i,j), \otimes\} \quad (3)$$



(a) $zip_{<2>}(\times) = elem(zip\text{-}pd_{<2>}(\times))$



(b) $reduce(+) = repeat_{<log_2>}(elem(reduce\text{-}pd(+), tree\text{-}span))$



(c) $inner\text{-}product = compose(zip_{<2>}(\times), reduce(+))$

Figure 2: An example for the composition in the STAPL Skeleton Framework: inner-product.

This parametric dependency says that for every element at index i, we apply the $\otimes$ operator on the two inputs at $2i$ and $2i+1$ from the previous iteration.

Using both the `elem` and the `repeat` operators, we define the data flow graph for the reduce skeleton as:

$$reduce(\otimes) = repeat_{<log_2>}(elem_{<tree_{span}>}(reduce\_pd(\otimes))) \quad (4)$$

The *tree-span* in Eq. 4 is a customized `span` in which its domain is halved at each iteration of the `repeat`(Fig. 2(b)). The log_2 size indicates that the number of iterations for this skeleton is determined by computing log_2 of the input size. Many skeletons listed in Table 1, such as `scan`, `butterfly`, and `broadcast`, are defined in the same way.

4.3.1 Ports

The ports for the data flow graph representation of a skeleton created using the `repeat` skeleton allow access to only the input port of the first iteration and to the output port of the last iteration. As an example, we have shown the input and output ports of a `reduce` skeleton in Fig. 2 (b). In this example, the *input-port* allows access to all elements in the domain of the input, while the $output-port$ only allows access to the final result.

4.4 compose - Functional Composition

In the STAPL Skeleton Framework, we allow skeletons to be composed using the `compose` operator. These compositions can be defined in two ways. In the first method, which we describe here, skeletons are composed using the simple *mathematical* notion of function composition:

$$\begin{aligned} compose(S_1, S_2, \ldots, S_n)\ x &= S_n \circ \ldots \circ S_2 \circ S_1 x \\ &= S_n(\ldots(S_1 x))) \end{aligned} \quad (5)$$

This operator can be used to compose many skeletons, such as the `inner-product` skeleton shown in Fig. 2(c) and the ones listed in [29]. The simplified C++ code required to express these skeletons in our framework is shown in Listing. 1. Each of these skeletons is defined using functions which return the result of composition.

```
auto zip_reduce<2>(*, +) = compose(zip<2>(*), reduce(+))
auto inner_product = compose(zip<2>(*), reduce(+));
auto allreduce(+) = compose(reduce(+), broadcast);
auto fft_dit = compose(bitreversal(), reverse_butterfly(fft_dit_op));
auto fft_dif = compose(butterfly(fft_dif_op), bitreversal());
```

Listing 1: Examples of functional composition.

4.4.1 Ports

The input port for the result of this type of composition is defined as the input port of the first skeleton in the composition. Similarly, the output port is defined as the output port of the last skeleton. As shown in Fig. 2(c), in an `inner-product` skeleton, the two input ports are connected to the input ports of the `zip` skeleton, and the output port is connected to the output port of the `reduce` skeleton.

These port mappings are used in the data flow graph generation phase, explained in Section 5, to define compositions in terms of point-to-point dependencies.

4.5 compose - Arbitrary Composition

Functional composition is very straightforward for defining simple compositions. However, this form of expressing composition becomes less expressive in arbitrary and complex compositions. In functional programming languages this problem is resolved using *let* expressions (*letrec*). A *let* expression simplifies compositions by potentially avoiding repetition and, at the same time, improving readability [23]. In a *let* expression, subexpressions are given names and can be reused; we use a mini-language to express the same type of compositions. The grammar for this mini-language is shown in Table 4.

4.5.1 Ports

We use our mini-language descriptions to define the mappings of the input and output of the skeletons in this type of composition. In Listing 2 we show how the mini-language can be used to express the functional composition used for the `inner-product` skeleton.

$Composition ::=$	$input = [Vars]$
	$Stmts$
	$output = [Vars]$
$Stmts ::=$	$Vars = Vars[Vars] \mid \epsilon$
$Vars ::=$	$ID\ Vars \mid \epsilon$

Table 4: The mini-language grammar used by the `compose` operator.

The first line after the name in Listing 2 indicates that this skeleton receives two inputs. In the second line, these inputs are passed to the **zip** skeleton. The result of this computation is then passed to the **reduce** skeleton. Finally, the output of the **reduce** skeleton is used as the output of the **inner-product** skeleton. This simple example defines the simple functional composition using our mini-languages and is given here for ease of understandability. In the STAPL Skeleton Framework, we provide generic port-mappings for simple functional compositions of arbitrary size and do not require users to provide port-mappings for such cases.

```
simple-port-mapping:
  input = [in1 in2]
  v1 = zip<2>(*) [in1 in2]
  v2 = reduce(+) [v1]
  output = [v2]

// C++ code for this composition
auto inner_product = compose<portmaps::simple-port-mapping>(
                       zip<2>(*), reduce(+))
```

Listing 2: `inner-product` composition using the mini-language.

We use the mini-language for complex compositions (such as the **bucket-sort** example in Listing 3) to express the port-mappings. We provide a Python-based tool which translates these descriptions to strongly-typed port-mappings in C++. The generated files can be used in compositions by simply adding the port-mapping name to the **compose** operator (**portmaps::simple-port-mapping** in Listing 2). Our tool also generates a GraphViz [11] dot file for the given composition description. As an example, the graphical representation of the **bucket-sort** skeleton is shown in Fig. 3.

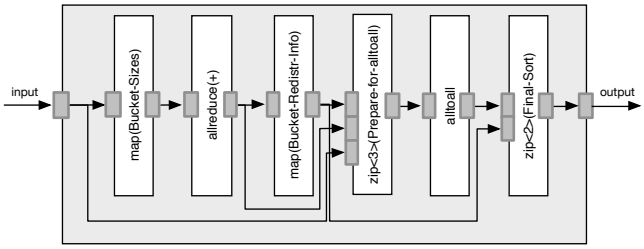


Figure 3: `bucket-sort` input/output port-mappings.

We use this method of composition in this paper to show the compositions used in our case studies in Section 6. The C++ code for these compositions is similar to the one line listed in Listing 2. Therefore, we omit it in the rest of the examples for better readability.

5. SKELETON EXECUTION

In the previous sections we explained the process of expressing parametric data flow graphs for skeletons. To execute these parametric definition in the STAPL Skeleton Framework, we need to first expand them based on the given inputs and generate data flow nodes for them. These data flow graphs need to be passed to a data flow engine for execution; this process is orchestrated by an entity called the *Skeleton Manager*. Upon the presence of input, the Skeleton Manager starts traversing the expression tree of a skeleton composition in a pre-order, depth-first traversal (Fig. 4). We refer to this step as the *spawning process*. The spawning process populates the data flow graph representation of a skeleton composition to an *execution environment*.

The action taken by the spawning process at each step of traversal varies based on the type of the node visited. For both **compose** and **repeat** nodes, the spawning process spawns children in order. The inputs and output for each child are determined by the given port-mapping. For **repeat** and **compose** in their simple functional composition forms, the input of each child is defined as the output of its preceding child. For the **compose** nodes with arbitrary compositions, these mappings are determined based on the description given in the mini-language description of the composition.

At **elem** nodes, the spawning process is called for all the points in the domain of the **span**. All children of **elem** node have access to the input and output of the **elem** node. Finally, for every parametric dependency reached in the expression tree, a data flow graph node is generated. The parametric dependency issues point-to-point consumption requests to the input port provided. Each port receiving these requests is either mapped to another port, which in turn passes the request on, or is directly defined over a set of tasks, i.e; the output port of an **elem** operator, in which case the dependency definition is finished. The spawning process terminates after all the tasks are generated.

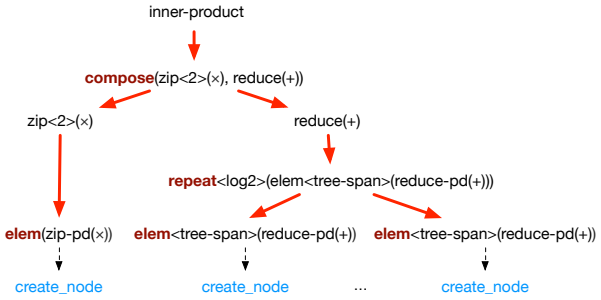


Figure 4: The spawning process of the `inner-product`skeleton.

We use STAPL PARAGRAPH as the execution environment for our spawning process. STAPL (Standard Template Adaptive Parallel Library) [14, 5, 6, 13, 27] is a parallel library which simplifies parallel programming by providing interfaces similar to the (sequential) ISO C++ standard library (STL) [21]. With the help of the STAPL PARAGRAPH we are able to run our data flow graph representation of skeletons efficiently, even on large scale systems.

In addition to the STAPL PARAGRAPH environment, we can populate our data flow graphs on other execution environments, making our framework potentially portable to other data flow engines. An execution environment defines the meaning of nodes and edges in a data flow graph. Some examples of the execution environments that we provide are: (1) a GraphViz environment which allows the data flow graphs to be stored as GraphViz [11] dot files, (2) a debug

environment which prints out the data flow graph specifications to standard output, (3) a graph environment which allows the data flow graphs to be stored in a STAPL parallel graph container [13].

6. CASE STUDIES

In this section, we show examples of using skeleton compositions in various applications. We start from the NAS EP (Embarrassingly Parallel) benchmark, which is defined as a single `map-reduce` skeleton. We show the NAS IS (Integer Sort) benchmark as a more complex example in which we use the `bucket-sort` skeleton. The `bucket-sort` skeleton is composed from many other provided skeletons, which is the first showcase for our composition operators. We then present the skeleton-based implementation of the 3D FFT computation used in the NAS FT (Fourier Transform). In this benchmark we use the `transpose-3D` skeleton, which is also composed from other skeletons. Finally, we show a real-world example, an N-Body application using the FMM algorithm, which uses the `bucket-sort` skeleton, as an example of reusability of skeletons, in addition to several other skeletons.

6.1 NAS Embarrassingly Parallel (EP)

The NAS EP benchmark is designed to evaluate an application with relatively little interprocessor communication. The only communications in this benchmark are the ones used in the pseudo-random number generation in the beginning and the collection of the results in the end. This benchmark determines an upper bound for machine floating point performance. The goal is to tabulate a set of Gaussian random deviates in successive square annuli.

The skeleton-based representation of this benchmark is specified with the `map-reduce` skeleton. The `map` skeleton generates pairs of uniform pseudo-random deviates in the range of $(0, 1)$. The pairs that satisfy the given predicate are summed to compute the global sums and compute the ten counts of deviates in square annuli.

6.2 NAS Integer Sort (IS)

In the NAS IS benchmark, an input with N keys is sorted [2]. The keys are uniformly distributed in memory and are generated using a predefined sequential key generator. The IS benchmark is expressed using the `bucket-sort` skeleton.

```
bucket_sort(bucket-sizes, bucket-redistr-info, final-sort):
  input = [in]
  v1 = map(bucket-sizes) [input]
  v2 = allreduce() [v1]
  v3 = map(bucket-redistr-info) [v2]
  v4 = zip<3>(prepare-for-alltoall) [v3 v2 input]
  v5 = alltoall [v4]
  v6 = zip<2>(final-sort) [v3 v5]
  output = [v6]
```

Listing 3: The NAS IS benchmark defined as a bucket-sort skeleton.

In the `bucket-sort` skeleton (Listing 3), the number of input values in each range are put into buckets. This information is then summed using the `allreduce` skeleton. The summation is used to determine a well-balanced repartitioning scheme for the values in `map`(*bucket-redistr-info*). For this benchmark, we use the partitioning strategy mentioned in the NAS Benchmarks reference manual [2]. Users can provide a customized partitioning strategy to the `bucket-sort` skeleton instead. In the next step, input values are grouped and prepared for a global exchange (`zip`<3>(*prepare-for-alltoall*)). The keys are then redistributed with the help of the `alltoall` skeleton. In the last step, the received values are locally sorted based on the given local sorting method (*final-sort*). We provide several implementations for the `alltoall` skeleton which can be selected by users. Currently, we support four methods for the `alltoall` skeleton; *flat*, *butterfly*, *hybrid*, and *pairwise-exchange*.

6.3 NAS Fourier Transform (FT)

In the NAS FT benchmark, a partial differential equation is solved by using a 3D FFT (Fast Fourier Transform) on a 3D matrix [2]. A 3D FFT algorithm is defined as three steps of applying 1D FFTs on a given matrix across each dimension. In the literature, two methods are generally used for this computation [7]: distributed and transpose-based. In the first method, an efficient distributed 1D FFT is used. Although this method is very expressive, the long-range accesses in the algorithm prevent this method from scaling beyond very small clusters [7]. In the second method, which relies on efficient implementation of the sequential 1D FFT, 1D FFTs are applied across the dimensions for which data is locally available. The matrix is transposed whenever necessary to make the values across a dimension available locally. We use the transpose-based method in our framework due to the lack of scaling in the distributed method (Listing 4).

There are also two methods for matrix partitioning which are used for 3D FFT in the literature [7, 1]: the *slab decomposition*, in which each processor gets a 2D slab of the input, and the *pencil decomposition*, in which each processor gets a pencil (column) of the input. In this paper, we show the expressive skeleton-based representation for the slab decomposition, although the pencil decomposition can be expressed in a similar way. As can be seen in Listing 4, in the forward FFT, we compute 1D FFTs across the x and y axes first, and then transpose the matrix from (x, y, z) to (z, x, y). We use `transpose-3D` for this step. This skeleton is defined in term of the `zip` and `alltoall` skeletons, which is omitted for the sake of brevity. After the matrix is transposed, 1D FFTs are computed across the z dimension. The reverse 3D FFT is defined in a similar way.

```
FFT-3D(forward):
  input = [in]
  v1 = map(cfft1) [in]
  v2 = map(cfft2) [v1]
  v3 = transpose_3d<2,0,1>() [v2]
  v4 = map(cfft1) [v3]
  output = [v4]

FFT-3D(reverse):
  input = [in]
  v1 = map(cfft1) [in]
  v2 = transpose_3d<1,0,2>() [v1]
  v3 = map(cfft1) [v2]
  v4 = map(cfft2) [v3]
  output = [v4]
```

Listing 4: The NAS FFT benchmark.

6.4 Fast Multipole Solver (FMM)

FMM is a fast algorithm designed to allow N-body applications to work at exascale. This algorithm consists of several hierarchical kernels, as shown in Fig. 5: P2M, M2M, M2L, L2L, and L2P [28]. Using a Taylor expansion, the

P2M (particle-to-multipole) kernel propagates the coordinates and potential information into a multipole expansion. The mathematical derivation method used for this step are described in [8]. The next expansion happens at the center of bigger cells using the M2M (multipole-to-multipole) kernel. The M2L (multipole-to-local) kernel then creates a local expansion, out of all multipoles in the tree, based on cell size to distance ratio. The L2L (local-to-local) kernel passes the received information downwards in the tree, until it reaches the particles, for the L2P (local-to-particle) kernel to be applied.

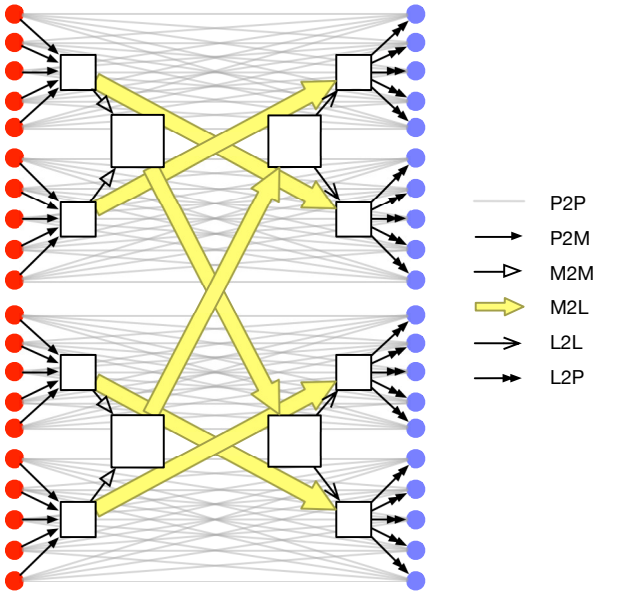


Figure 5: FMM hierarchical kernels.

Using the set of skeletons provided in our framework, we have implemented FMM in terms of skeletons (Listing 5). As we will see in the evaluation section, our implementation not only increases the expressivity of this algorithm, but also improves its overall performance in comparison to the reference implementations. In addition, by providing algorithm selection at user's level (switching between various implementations of `alltoall`, `allreduce`, and `allgather` skeletons in our framework), we allow algorithm developers to adapt their algorithm based on their heuristics. This is an advantage over the MPI implementation of such algorithms.

Moreover, our implementation is reusable for other PDE problems that can be solved using Green's function of the form listed in Eq. 6 (δ represents the Dirac delta function). The Green's function, in these problems, provides a visual interpretation of an impulse response, which is caused by a force or a charge at the center of mass.

$$\pounds G(x,s) = \delta(x-s) \tag{6}$$

To achieve acceptable performance in an N-Body problem, an efficient partitioning scheme should be used. This partitioning scheme has to consider three factors: number of particles per partition, workload, and input distribution. For a uniform distribution, the global geometric ORB (Orthogonal Recursive Bisection) is sufficient for achieving a suitable load-balancing. For a non-uniform distributions the plummer distribution can be used. In both cases, it is important to keep geometrically close points in the same partition, to reduce the intra-node interaction list sizes. Additionally, the previous workload and the amount of communication have to be balanced simultaneously to achieve load-balance across time-steps.

To fine-tune the partitioning algorithm based on the uniformity of input, we use the `bucket-sort` skeleton, listed in the IS benchmark (Listing. 3). Using this skeleton, we sort the splitters for our partitioning selection criteria and determine the physical bounds of each partition by applying an `allreduce` skeleton on the results. We then use the `build-tree` to build an octree at each partition, based on the geometric positions of particles. In the `upward_pass`, we recursively calls the `P2M` kernel for terminal cells or M2M (if a cell is within the multipole acceptance criteria). When octrees are created, we recalculate the bounds and populate them with the help of an `allreduce` skeleton followed by an `allgather` skeleton. Afterwards, to reduce communication in the next steps, we create the Local Essential Trees (LETs) in each partition and make this information available to other partitions using the `alltoall` skeleton. LETs contain a subset of the local tree cells that are required by other partition for their computations. Since the number of LETs required by a partition is inversely proportional to distance, the complexity of this step is $O(log(P))$. Once LETs are received, we combine them to form the global octree. In the `upward_pass` kernel, we invoke the `M2L`, `L2P` and `P2P` kernels on the trees, to calculate potential and acceleration of target particles. Finally, we use the `repeat` operator to calculate the subsequent snapshots. Since particles at boundaries might move across nearby partitions, we do not require a global repartitioning.

```
FMM:
  input = [bodies tree]
  x1 = bucket-sort(selection-citeria) input
  x2 = alltoall [x1]
  x3 = allreduce(<,>) [x2]
  x4 = map(build-tree) [x3]
  repeat
  {
    x5 = map(upward_pass) [x4 x2]
    x6 = allreduce(<,>) [x3 x4]
    x7 = allgather() [x6]
    x8 = alltoall() [x7]
    x9 = map(DTT) [x5]
    x10= map(Traverse and DP) [x5]
  }(max_steps)
  output = [x10]
```

Listing 5: N-Body using the FMM hierarchical algorithm.

7. EVALUATION

We have evaluated our framework on three massively parallel systems: a 24,576 node BG/Q system (Vulcan from LLNL), each node containing a 16-core IBM PowerPC A2 for a total of 393,216 cores; a 5,272 node Cray XC30 supercomputer (Piz Daint from CSCS Switzerland), each node containing a hyperthreaded 8-core Intel SandyBridge with 32GB per node, for a total of 42,176 cores (84,352 hyperthreads); and a 1,266 node Cray XC40 supercomputer (a Piz Daint extension from CSCS Switzerland), each node containing two 12-core Intel Haswell CPUs, for a toal of 30,144 cores.

7.1 Fundamental Skeletons

To show the efficiency of our fundamental skeletons we show scalability results for the `map`, `reduce`, and `scan` skeletons in Fig. 6, 8, and 7. Our results indicate that these fundamental skeletons can easily scale beyond 100,000 cores. The jump at 64k processes in Fig. 7 is due to the implicit change of algorithms in the MPI implementation of MPI_Scan. As you can see in this figure, the implicit algorithm selection is not always favorable. For this reason, we not only provide default efficient implementations for various fundamental skeletons, but also allow algorithm developers to choose their desired algorithm in their specification.

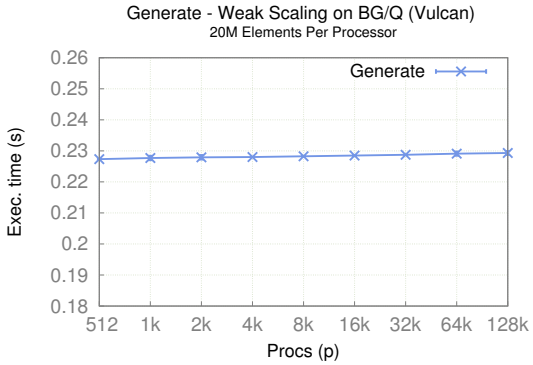


Figure 6: Weak scalability of Generate as an example of *map*.

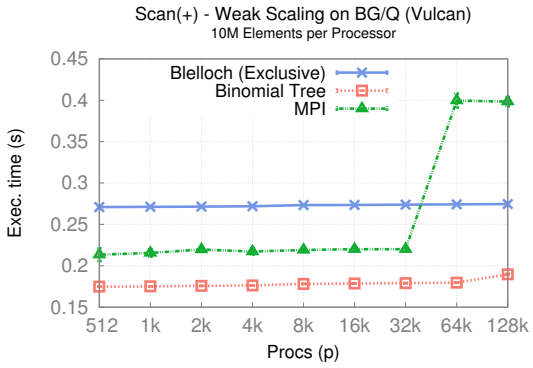


Figure 7: Weak scalability of the various *scan* skeletons vs. a hand-optimized MPI *scan*

7.2 NAS Benchmarks

7.2.1 NAS Embarrassingly Parallel

The EP benchmark shows an example of simple functional composition. This benchmark is expressed as a `map-reduce` skeleton. The `map-reduce` skeleton is in turn composed from our `map` and `reduce` skeletons. The obtained results in Fig. 9 indicate that the simple functional composition in our framework preserves the efficiency of the nested skeletons.

7.2.2 NAS Integer Sort

The IS benchmark shows an example of arbitrary compositions. The results for this benchmark are especially important; they are indicative of the efficiency of arbitrary compositions. Our results for class D of this application (Fig. 10)

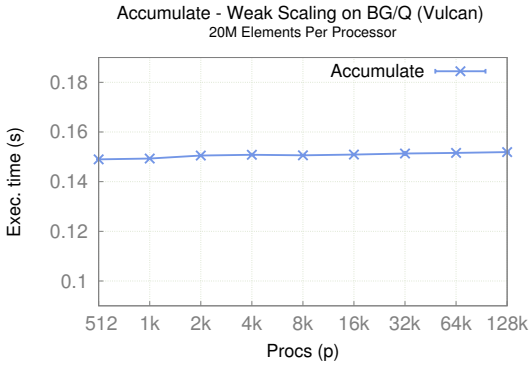


Figure 8: Weak scalability of accumulate(+) as an example of *reduce*.

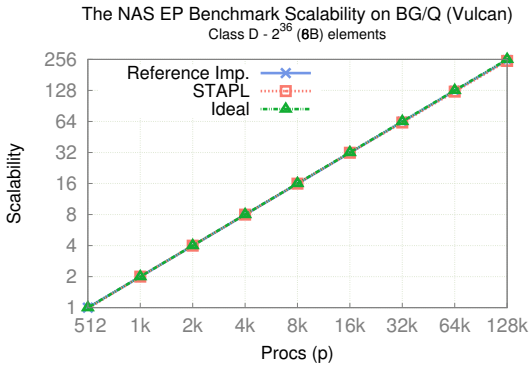


Figure 9: Performance comparison of class D NAS EP benchmark with the reference implementation.

show that our skeleton-based version of this benchmark performs better than the hand-optimized version of this benchmark. The reference MPI IS implementation consists of several blocking MPI calls, such as `alltoall`, and `allreduce`. These constructs require global synchronizations in MPI after each invocation. We avoid these synchronizations by representing the problem in our framework in terms of skeletons. As mentioned earlier, the data flow representation of skeletons, along with the composition operators, can represent the data dependencies between the steps of the problem as point-to-point dependencies of their corresponding data flow graphs. This provides our implementation with better performance than the reference implementation by avoiding global synchronizations (Fig. 10).

7.2.3 NAS Fourier Transform

As shown in Fig. 11, our implementation of the FT benchmark shows comparable performance to the hand-optimized reference implementation (Fig. 11). The FT benchmark stresses the system with interprocessor communication in its 3D matrix transpose, requiring efficient `alltoall` communication. Our overhead is due to the copy-semantics of the STAPL runtime system. The STAPL runtime system requires one copy between the user-level to the MPI level on both sender and receiver side. These extra copies result in only 30-40% overhead in our implementation, as some of the overhead is compensated for by not requiring global synchronizations.

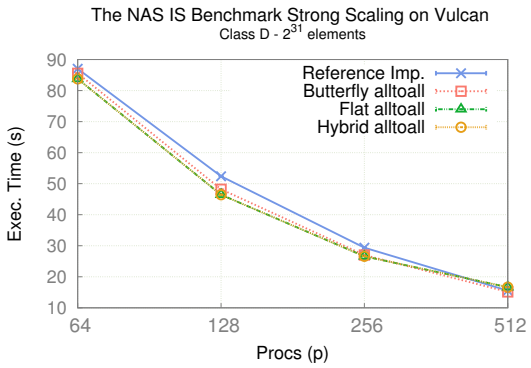


Figure 10: Performance comparison of class D NAS IS benchmark with the reference implementation.

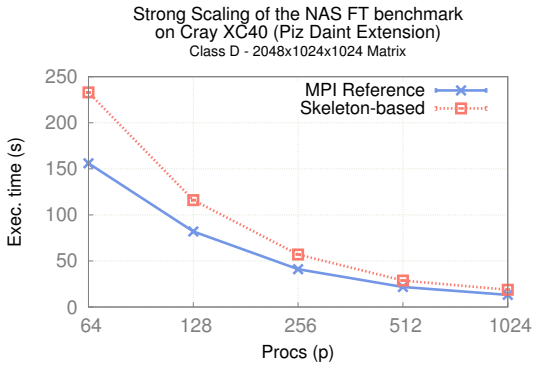


Figure 11: Performance comparison of class D of NAS EP benchmark with the reference implementation.

7.3 Skeleton-based FMM Algorithm

As can be seen in Fig. 12 and 13, our skeleton-based implementation of the FMM algorithm can scale better than the reference implementation. In both versions, we are exploiting distributed memory parallelism; no shared-memory or GPU optimizations are involved. This is attributable to the fact that, for this benchmark, we are making a direct comparison between a hand-optimized MPI implementation, given by Exa-FMM reference code, and a Skeleton-based distributed implementation that internally uses the same serial FMM kernels given by the reference implementation.

In Fig. 12, we show the weak-scaling results for 50k particles per process (100M particles on 2k processes). As can be seen in this figure, the execution time for the reference implementation diverges rapidly from the skeleton-based version, after 32 processes and runs out of memory after 128 processes. This is due to the overhead caused by the global communication of LETs in the reference implementation, using MPI_Alltoall. Our implementation continues to scale up to 2k processes. By replacing global synchronization with point-to-point dependency and using skeletons in our algorithm, we are able to scale by 2 more orders of magnitude than the reference implementation and solve a problem of 10^8 particles.

In Fig. 13, we show the strong scaling results for a problem of 25M particles. In this figure, the scalability of our implementation persists until 256 cores; at this point the theoretical amount of serial work assuming no communication is approximately 0.16s. This is considerably smaller than the actual amount of communication. The reference implementation crashes due of the same memory allocation issue, mentioned earlier, after 8 processes.

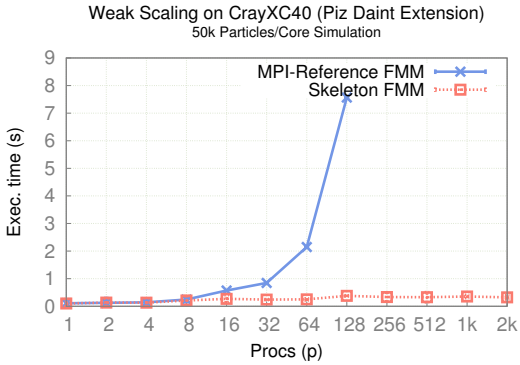


Figure 12: Weak scaling of FMM using MPI vs Skeleton-based Implementations

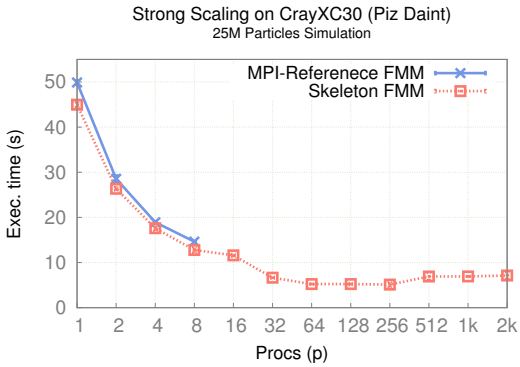


Figure 13: Strong scaling of FMM using MPI vs Skeleton-based Implementations

8. CONCLUSION

In this work, we presented a novel method for representing skeletons and their composition in the STAPL Skeleton Framework. We defined fundamental skeletons from their fine-grain dependence relations, e.g., `map`, `zip`, and `reduce`, and expanded them to form their data flow graph representations. We then composed these skeletons to build more complex real-world applications and benchmarks. With the abstraction that ports provided, we were able to define data dependencies between the skeletons as point-to-point dependencies of their corresponding data flow graphs, allowing programs to scale efficiently beyond 100,000 cores. Our framework addresses many concerns listed in [19] for a parallel programming framework: scalability, portability, extensibility, debuggablity, tunability, and maintainability.

9. ACKNOWLEDGMENTS

This research supported in part by NSF awards CNS-0551685, CCF 0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053, IIS-0917266, RI-1217991, EFRI-1240483, by NIH NCI R25 CA090301-11, by DOE

awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

10. REFERENCES

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. An efficient parallel algorithm for the 3D FFT NAS parallel benchmark. In *Proceedings of the Scalable High-Performance Computing Conf.*, pages 129–133. IEEE, 1994.

[2] D. Bailey and T. H. et al. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Dec. 1995.

[3] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the 11th ACM SIGPLAN symp. on Principles and practice of par. prog.*, pages 48–57. ACM, 2006.

[4] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Sci. Prog.*, 18(3):203–217, 2010.

[5] A. Buss, A. Fidel, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, L. Rauchwerger, et al. The STAPL pView. In *Int. Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 261–275. Springer, 2011.

[6] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. Stapl: Standard Template Adaptive Parallel Library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, 2010. ACM.

[7] A. Chan, P. Balaji, W. Gropp, and R. Thakur. Communication analysis of parallel 3D FFT for flat cartesian meshes on large Blue Gene systems. In *High Perf. Comp. (HiPC)*, pages 350–364. Springer, 2008.

[8] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Comp. Physics*, 155(2):468–498, 1999.

[9] M. I. Cole. *Algorithmic skeletons: structured management of parallel computing*. Pitman/MIT, London, 1989.

[10] J. F. Ferreira, J. L. Sobral, and A. J. ProenÃ§a. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *in CCGRID, IEEE Computer Society*, pages 301–304, 2006.

[11] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[12] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[13] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, pages 46–60, 2013.

[14] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[15] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[16] H. Kuchen and J. Striegnitz. Features from functional programming for a C++ Skeleton library: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):739–756, June 2005.

[17] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[18] K. Matsuzaki and H. Iwasaki. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st Int. Conf. on Scalable information systems*, page 13. ACM Press, 2006.

[19] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[20] U. Müller-Funk, U. Thonemann, and G. Vossen. The Münster skeleton library Muesli–a comprehensive overview. 2009.

[21] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, 2nd Edition*. Addison-Wesley, 2001.

[22] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming language. In *Proceedings of the 6th Int. Conf. on Supercomputing*, pages 167–177. ACM, 1992.

[23] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, 2002.

[24] F. Rabhi and S. Gorlatch. *Patterns and skeletons for Parallel and Distributed Computing*. Springer, 2003.

[25] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Comp. in Sci. and Eng.*, 15(2):0066–71, 2013.

[26] J. Sérot, J. S, J.-P. DERUTIN, D. GINHAC, and J. pierre D. SKiPPER - a skeleton-based parallel programming environment for real-time image processing applications, 1999.

[27] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. *SIGPLAN Not.*, 46(8):235–246, Feb. 2011.

[28] R. Yokota and L. Barba. Comparing the treecode with FMM on GPUs for vortex particle simulations of a leapfrogging vortex ring. *Computers & Fluids*, 45(1):155–161, 2011.

[29] M. Zandifar, N. Thomas, N. M. Amato, and L. Rauchwerger. The STAPL Skeleton Framework. In *Int. Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, Hillsboro, OR, 2014.