# The STAPL Skeleton Framework *

Mani Zandifar, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{mazaninfardi,nthomas,amato,rwerger}@cs.tamu.edu

**Abstract.** This paper describes the STAPL Skeleton Framework, a high-level skeletal approach for parallel programming. This framework abstracts the underlying details of data distribution and parallelism from programmers and enables them to express parallel programs as a composition of existing elementary skeletons such as map, map-reduce, scan, zip, butterfly, allreduce, alltoall and user-defined custom skeletons.
Skeletons in this framework are defined as parametric data flow graphs, and their compositions are defined in terms of data flow graph compositions. Defining the composition in this manner allows dependencies between skeletons to be defined in terms of point-to-point dependencies, avoiding unnecessary global synchronizations. To show the ease of composability and expressivity, we implemented the NAS Integer Sort (IS) and Embarrassingly Parallel (EP) benchmarks using skeletons and demonstrate comparable performance to the hand-optimized reference implementations. To demonstrate scalable performance, we show a transformation which enables applications written in terms of skeletons to run on more than 100,000 cores.

## 1 Introduction

Facilitating the creation of parallel programs has been a concerted research effort for many years. Writing efficient and scalable algorithms usually requires programmers to be aware of the underlying parallelism details and data-distribution. There have been many efforts in the past to address this issue by providing higher-level data structures [29,6], higher-level parallel algorithms [8,19,21], higher-level abstract languages [5,27,19], and graphical parallel programming languages [25]. However, most of these studies focus on a single paradigm and are limited to specific programming models.

Algorithmic skeletons [9], on the other hand, address the issue of parallel programming in a portable and implementation-independent way. Skeletons are

* This research supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, Chevron, IBM, Intel, Oracle/Sun 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.

defined as polymorphic higher-order functions, that can be composed using function composition and serve as the building blocks of parallel programs. The higher-level representation of skeletons provides opportunities for formal analysis and transformations [26] while hiding underlying implementation details from end users. The implementation of each skeleton in a parallel system is left to skeleton library developers, separating algorithm specification from execution. A very well-known example of skeletons used in distributed programming is the *map-reduce* skeleton, used for generating and processing large data sets [10].

There are many frameworks and libraries based on the idea of algorithmic skeletons [14]. The most recent ones include Müesli [23], FastFlow [2], SkeTo [20], and the Paraphrase Project [15] that provide implementations for several skeletons listed in [26], such as *map*, *zip*, *reduce*, *scan*, *farm*. However, there are two major issues with existing methods that prevent them from scaling on large systems. First, most existing libraries provide skeleton implementations only for shared-memory systems. Porting such codes to distributed memory systems usually requires a reimplementation of each skeleton. Therefore, the work in this area, such as [1], is still very preliminary. Second, in these libraries, composition of skeletons is not projected into the implementation level, requiring skeleton library developers to provide either new implementations for composed skeletons [23] or insert global synchronizations between skeleton invocations resulting in a Bulk Synchronous Parallel (BSP) model, as in [20] which generally cannot achieve optimal performance.

In this work, we introduce the STAPL Skeleton Framework, a framework that enables algorithmic skeletons to scale on distributed memory systems. Skeletons in this framework are represented as parametric data flow graphs that allow parallelism to be expressed explicitly. Therefore, skeletons specified this way are inherently ready for parallel execution regardless of the underlying runtime execution model. These parametric data flow graphs are expanded over the input data and are executed in the STAPL data flow engine known as the `PARAGRAPH`. We show that parallel programs written this way can scale on more than 100,000 cores.

Our contributions in this paper are as follows:

- a skeleton framework based on parametric data flow graphs that can easily be used in both shared and distributed memory systems.
- a direct mapping of skeleton composition as the composition of parametric data flow graphs, allowing skeletons to scale on large supercomputers without the need for global synchronizations.
- an extensible framework to which new skeletons can be easily added through composition of existing skeletons or adding new ones.
- a portable framework that can be used with data flow engines other than the STAPL `PARAGRAPH` by implementing an execution environment interface.

This paper is organized as follows: In Section 2, we present the related work in the area of algorithmic skeletons. In Section 3, we provide an overview of the STAPL Skeleton framework where we show how to break the task of writing

parallel programs into algorithm specification and execution. In Section 4, we show a transformation that allows fine-grain skeletons to execute and perform well in a parallel environment. In Section 5, we present a case study showing expressivity and composability using the NAS EP and IS benchmarks. We evaluate our framework using experiments over a wide set of skeletons in Section 6. Conclusions and future work are presented in Section 7.

## 2 Related Work

Since the the first appearance of skeleton-based programming in [9], several skeleton libraries have been introduced. The most recent efforts related to our approach are Müesli [23], FastFlow [2], Quaff [11], and SkeTo [20].

The Münster skeleton library (Müesli) is a C++ library that supports polymorphic task parallel skeletons such as *pipeline*, *farm*, and data parallel skeletons such as *map*, *zip*, *reduce*, and *scan* on array and matrix containers. Müesli can work both in shared and distributed memory systems on top of OpenMP and MPI, respectively. Skeleton composition in Müesli is limited in the sense that composed skeletons require redefinitions and cannot be defined directly as a composition of elementary skeletons.

FastFlow is a C++ skeleton framework targeting cache-coherent shared-memory multi-cores [2]. FastFlow is based on efficient Single-Producer-Single-Consumer (SPSC) and Multiple-Producer-Multiple-Consumer (MPMC) FIFO queues which are both lock-free and wait-free. In [1] the design and the implementation of the extension of FastFlow to distributed systems has been proposed and evaluated. However, the extension is evaluated on only limited core counts (a $2 \times 16$ core cluster). In addition, the composition is limited to task parallel skeletons with intermediate buffers, which limits their scalability.

Quaff is a skeleton library based on C++ template meta-programming techniques. Quaff reduces the runtime overhead of programs by applying transformations on skeletons at compile time. The skeletons provided in this library are *seq*, *pipe*, *farm*, *scm* (split-compute-merge), and *pardo*. Programs can be written as composition of the above patterns. However, Quaff only supports task parallel skeletons and is limited to shared-memory systems.

SkeTo is another C++ skeleton library built on top of MPI that provides parallel data structures:*list*, *matrix*, and *trees*, and a set of skeletons *map*, *reduce*, *scan*, *zip*. SkeTo allows new skeletons to be defined in terms of successive invocations of the existing skeletons. Therefore, the approach is based on a Bulk Synchronous Parallel model and requires global synchronization in between skeleton invocations in a skeleton composition. In our framework, we avoid global synchronizations by describing skeleton compositions as point-to-point dependencies between their data flow graph representations.

## 3 STAPL Skeleton Framework

The STAPL Skeleton Framework is built on top of the Standard Template Adaptive Parallel Library (STAPL) [6,7,16,29] and is an interface for algorithm de-

velopers as depicted in Fig. 7. STAPL is a framework for parallel C++ code development with interfaces similar to the (sequential) ISO C++ standard library (STL) [24]. STAPL hides the notion of processing elements and allows asynchronous communication through remote method invocations (RMIs) on shared objects. In addition, STAPL provides a data flow engine called the `PARAGRAPH` , which allows parallelism to be expressed explicitly using data flow graphs (a.k.a. task graphs). The runtime system of STAPL is the only platform specific component of STAPL, making STAPL programs portable to different platforms and architectures without modification.

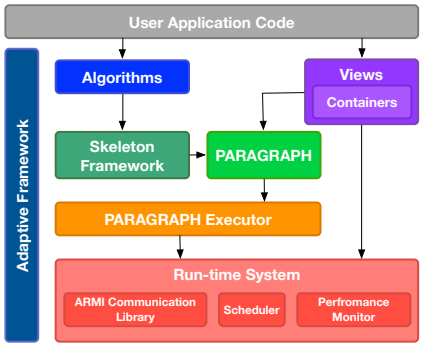



Fig. 1: The STAPL Library component diagram.

Using the STAPL Skeleton Framework, algorithm developers only focus on defining their computation in terms of skeletons. As we will see in this section, each skeleton is translated to a parametric data flow graph and is expanded upon the presence of input data. The data flow representation of skeletons allows programs to run on distributed and shared memory systems. In addition, this representation formulates skeleton composition as point-to-point dependencies between parametric data flow graphs, allowing programs to execute without the need for global synchronization.

### 3.1 Algorithm Specification

**Parametric Dependencies.** In our framework, skeletons are defined in terms of parametric data flow graphs. We name the finest-grain node in a parametric data flow graph a *parametric dependency (pd)*. A parametric dependency defines the relation between the input and output elements of a skeleton as a parametric coordinate mapping and an operation.

The simplest parametric dependency is defined for the *map* skeleton:

$$\begin{aligned} &map(\oplus)[a_1 \dots a_n] = [\oplus(a_1) \cdots \oplus (a_n)] \\ &map\text{-}pd(\oplus) \equiv \{< i > \mapsto < i >, \oplus\} \end{aligned} \tag{1}$$

In other words, the element at index $i$ of the output is computed by applying $\oplus$ on the element at index $i$ of the input. This representation carries spatial

information about the input element. As we will see later, it is used to build data flow graphs from parametric dependencies.

The $zip_k$ skeleton is a generalization of the *map* skeleton over $k$ lists:

$$\begin{aligned} &zip_k(\oslash)[a_1^1,\ldots,a_n^1]\ldots[a_1^k,\ldots,a_n^k] = [\oslash(a_1^1,\ldots,a_1^k),\ldots,\oslash(a_n^1,\ldots,a_n^k)] \\ &zip\text{-}pd_k(\oslash) \equiv \{< \underbrace{i,\ldots,i}_{\text{k}} > \mapsto < i >, \oslash)\} \end{aligned} \tag{2}$$

**elem operator.** Parametric dependencies are expanded over the input size with the data parallel *elem* compositional operator. An *elem* operator receives a parametric dependency (of type $\delta$) and expands it over the given input, with the help of *span* (of type $\psi$), to form a list of nodes in a data flow graph:

$$\begin{aligned} &elem :: \psi \to \delta \to [\delta] \\ &elem_{span}(parametric\text{-}dependency) \end{aligned} \tag{3}$$

For ease of readability in Eq. 3, we show *span* as a subscript and the *parametric dependency* in parenthesis.

A *span* is defined as a subdomain of the input. Intuitively, the default *span* is defined over the full domain of the input and is omitted for brevity in the default cases. As we will see later in this section, there are other *spans*, such as *tree-span* and *rev-tree-span*, used to define skeletons with tree-based data flow graphs.

With the help of the *elem* operator, *map* and *zip* skeletons are defined as:

$$\begin{aligned} &map(\oplus) = elem(map\text{-}pd(\oplus)) \\ &zip_k(\oslash) = elem(zip\text{-}pd_k(\oslash)) \end{aligned} \tag{4}$$

Given an input, these parametric definitions are instantiated as task graphs in the STAPL Skeleton Framework.

**repeat operator.** Many skeletons can be defined as tree-based or multilevel data flow graphs. Our *repeat* operator allows such skeletons to be expressed simply as such. The *repeat* operator is a function receiving a skeleton and applying it to a given input successively for a given number of times specified by a unary operator of type $\beta \to \beta$ called $\xi$:

$$\begin{aligned} &repeat :: [\alpha] \to [\alpha] \to (\beta \to \beta) \to [\alpha] \to [\alpha] \\ &repeat(S,\xi)[a_1,\ldots,a_n] = (\underbrace{S\ldots(S(S}_{\xi(n)\ times}[a_1,\ldots,a_n])))) \end{aligned} \tag{5}$$

An example of the *repeat* operator is a tree-based data flow graph definition of the *reduce* skeleton (Fig. 2). In a tree-based *reduce*, each element at level $j$ depends on two elements at level $j-1$. Therefore, the parametric dependency for each level of this skeleton can be specified as:

$$reduce\text{-}pd(\otimes) \equiv \{(< 2i, j-1 >, < 2i+1, j-1 >) \mapsto < i, j >, \otimes\} \tag{6}$$

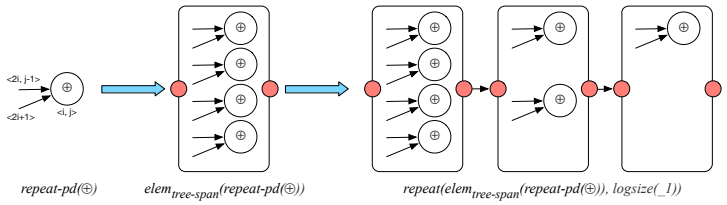
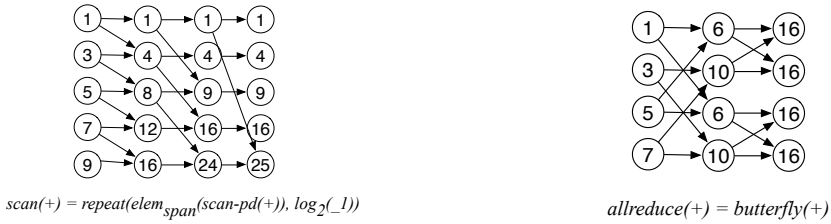



Fig. 2: The process of creating the tree-based representation of the *reduce* skeleton.

Each level of this tree representation is then expanded using the *elem* operator. However, the expansion is done in a different way than the default case used in Eq. 1 and 2. In a tree, the *span* of the *elem* operator at each level is half of its previous level, starting from the *span* over the domain of the input at level 0. We name this *span* a *tree-span* and we use it to define a tree-based *reduce*:

$$reduce(\otimes) = repeat(elem_{tree\text{-}span}(reduce\text{-}pd(\otimes)), log_2(size)) \tag{7}$$

Similarly, other skeletons can be defined using the *elem* and *repeat* operators




(a) Simultaneous binomial tree *scan(+)*. (b) Butterfly-based *allreduce*.

Fig. 3: Data flow representation of the binomial *scan* and *butterfly* skeletons.

such as *scan*, *butterfly*, *reverse-butterfly*, and *broadcast* as shown in Fig. 4. For brevity, we show only the simultaneous binomial tree implementation of the *scan* skeleton in Fig. 3(a) and 4. However, we support two other scan implementations in our framework, namely the exclusive scan implementation introduced in [4] and the binomial tree scan [28], which are expressed in a similar way.

**compose operator.** In addition to the data flow graph composition operators presented above, we provide a skeleton composition operator called *compose*. The *compose* operator, in its simplest form, serves as the functional composition used in the literature for skeleton composition and is defined as:

$$compose(S_1, S_2, \ldots, S_n)\ x\ = S_n \circ ... \circ S_2 \circ S_1 x\ =\ S_n(\ldots(S_1 x))) \tag{8}$$

The *map-reduce* and the *allreduce* skeleton are skeletons which can be built from the existing skeletons (Fig. 4) using the *compose* operator.

**The binomial tree representation of the *scan* skeleton**

$$scan\text{-}pd(\oplus) \equiv \begin{cases} \{(< i, level-1 >, < i-2^{level}, level-1 >) \mapsto < i, level >, \oplus\} & \text{if } i \geq 2^{level} \\ \{(< i, level-1 > \mapsto < i, level >), id\} & \text{if } i < 2^{level} \end{cases}$$

$$scan(\oplus) = repeat(elem(scan\text{-}pd(\oplus)), log_2(size))$$

**The k-ary *tree* and *reverse-tree* skeletons:**

$$tree_k(pd) = repeat(elem_{tree-span_k}(pd), log_k(size))$$

$$rev\text{-}tree_k(pd) = repeat(elem_{rev\text{-}tree\text{-}span_k}(pd), log_k(size))$$

**The *broadcast* skeleton:**

$$broadcast\text{-}pd \equiv \{(< i/2 > \mapsto < i >), \lambda x.x\}$$

$$broadcast = rev\text{-}tree(broadcast\text{-}pd)$$

**The *butterfly* and *reverse-butterfly* skeletons:**

$$butterfly\text{-}pd(\otimes) \equiv \{(< i, level-1 >, < i \pm 2^{n-level-1}, level-1 >) \mapsto < i, level >, \otimes)\}$$

$$rev\text{-}butterfly\text{-}pd(\otimes) \equiv \{(< i, level-1 >, < i \pm 2^{level}, level-1 >) \mapsto < i, level >, \otimes\}$$

$$butterfly(\otimes) = repeat(elem(butterfly\text{-}pd(\otimes)), log_2(size))$$

$$rev\text{-}butterfly(\otimes) = repeat(elem(rev-butterfly\text{-}pd(\otimes)), log_2(size))$$

**Skeletons composed from the existing skeletons:**

$$allreduce_1(\oplus) = butterfly(\oplus)$$

$$allreduce_2(\oplus) = broadcast \circ reduce(\oplus)$$

$$alltoall(\star) = butterfly(\star)$$

$$fft\text{-}DIT = butterfly(fft\text{-}DIT\text{-}op)$$

$$fft\text{-}DIF = ref\text{-}butterfly(fft\text{-}DIF\text{-}op)$$

$$map\text{-}reduce(\oplus, \otimes) = reduce(\otimes) \circ map(\oplus)$$

**The *do-while* skeletons:**

$$\begin{aligned} &while\ p\ S\ x = \ if\ p\ x \\ &\ \ then\ while\ p\ S\ (S\ x) \\ &\ \ else\ x \end{aligned}$$

Fig. 4: A list of skeletons compositions

**do-while operator.** The compositional operators we mentioned so far cover skeletons that are static by definition. A *do-while* skeleton is intended to be used in dynamic computations which are bounded by a predicate $p$ as in [17]. The *do-while* skeleton applies the same skeleton $S$ to a given input until the predicate is satisfied. It is defined as shown in Fig. 4.

The execution of the *do-while* skeleton requires its corresponding data flow graph to be dynamic, as the number of iterations are not known a priori. This functionality is allowed in our framework with the help of the *memento* design pattern [12], as we will see later in Section 3.2.

**Flows.** So far, we have only showed the skeletons that are composed using *repeat* and *compose* using simple functional composition. In these compositions, a skeleton's output is passed as the input to the subsequent skeleton. Similar to the *let* construct in functional programming languages, input/output dependencies between skeletons can be defined arbitrarily as well (e.g., Fig. 5). To represent such compositions in our internal representation of skeletons, we define one input

and one output port (depicted as red filled circles in Fig. 2 and 5) for each skeleton. We formulate the skeleton composition as the connections between these ports and refer to them as *flows*. Flows are similar to the notion of flows in flow-based programming [22].

With the help of *ports* and *flows*, skeleton composition is directly mapped to point-to-point dependencies in data flow graphs, avoiding unnecessary global synchronizations commonly used in BSP models. As a concrete example, Fig 5 shows a customized flow used for NAS IS skeleton-based representation.

### 3.2 Algorithm Execution

In the previous section we looked at algorithm specification. In this section, we explain how an input-size independent algorithm specification is converted to a data flow graph through the *spawning* process.

**Skeleton Manager.** The Skeleton Manager orchestrates the spawning process in which a skeleton composition is traversed in a pre-order depth-first-traversal, in order to generate its corresponding data flow graph. The nodes of this data flow graph correspond to the parametric dependency instances in a composition. If a `PARAGRAPH` environment is used, these data flow graphs will represent a taskgraph and will be executed by the data flow engine of STAPL called the `PARAGRAPH`. The creation and execution of taskgraphs in a STAPL `PARAGRAPH` can progress at the same time, allowing overlap of computation and communication.

**Environments.** An environment defines the meaning of data flow graph nodes generated during the spawning process. As we saw earlier, in a `PARAGRAPH` environment each data flow graph node represents a task in a taskgraph. Similarly, other environments can be defined for execution or additional purposes, making our skeleton framework portable to other libraries and parallel frameworks.

For example, we used other environments in addition to the `PARAGRAPH` environment for debugging purposes such as (1) a GraphViz environment which allows the data flow graphs to be stored as GraphViz dot files [13], (2) a debug environment which prints out the data flow graph specifications on screen, and (3) a graph environment which allows the data flow graphs to be stored in a STAPL parallel graph container [16]. Other environments can also be easily defined by implementing the environment interface.

**Memento Queue.** The Skeleton Manager uses the memento design pattern [12] to record, pause, and resume the spawning process allowing the incremental creation of task graphs, and execution of dynamic skeletons. For example, the continuation and the next iteration of a *do-while* skeleton are stored in the back and the front of the memento queue, respectively, in order to allow input-dependent execution.

## 4 Skeleton Transformations

As mentioned earlier, various algorithms can be specified as compositions of skeletons. Since skeletons are specified using high-level abstractions, algorithms

written using skeletons can be simply analyzed and transformed for various purposes, including performance improvement.

In this section, we define the coarsening transformation operator $\mathcal{C}$ which enables efficient execution of skeletons using hybrid (a.k.a. macro) data flow graphs [18] instead of fine-grained data flow graphs.

### 4.1 Definitions

Before explaining the coarsening transformation, we need to explain a few terms that are used later in this section.

**dist and flatten skeletons.** A *dist* skeleton [17] partitions the input data and a *flatten* (*projection*) skeleton unpartitions the input data. They are defined as:

$$\begin{aligned} &dist\,[a_1,\ldots,a_n] \;=\; [[a_1,\ldots,a_k],\ldots,[a_j,\ldots,a_n]] \\ &flatten\,[[a_1,\ldots,a_k],\ldots,[a_j,\ldots,a_n]] \;=\; [a_1,\ldots,a_n] \end{aligned} \tag{9}$$

**Homomorphism.** A function on a list is a homomorphism with respect to a binary operator $\oplus$ iff on lists $x$ and $y$ we have [26]:

$$f(x +\!\!+\; y) = f(x) \oplus f(y) \tag{10}$$

in which $+\!\!+$ is the list concatenation operator.

The skeletons that are list homomorphisms can be defined as a composition of the *map* and the *reduce* skeletons, making them suitable for execution in parallel systems. However, as mentioned in [26], finding the correct operators for the *map* and *reduce* can be difficult even for very simple computations. Therefore, in our transformations of skeletons which are list homomorphisms, we use their *map-reduce* representation only when the operators can be devised simply, and in other cases we define a new transformation.

### 4.2 Coarsening Transformations ($\mathcal{C}$)

As we saw earlier, skeletons are defined in terms of parametric data flow graphs. Although fine-grained data flow graphs expose maximum parallelism, research has shown [18] that running fined-grained data flow graphs can have significant overhead on program execution on Von Neumann machines. This is due to the lack of spatial and temporal locality and the overhead of task creation, execution, and pre/post-processing. In fact, the optimum granularity of data flow graphs depends on many factors, one of the most important being hardware characteristics. Therefore, we define the coarsening transformation in this section as a transformation which is parametric on the input size where granularity can be tuned per application and machine.

The coarsening transformations, listed in Eq. 11, use *dist* skeleton to make coarser chunks of data (similar to the approach used in [17]). Then they apply an operation on each chunk of data (e.g., $map(map(\oplus))$ in $\mathcal{C}(map(\oplus))$). Subsequently,

they might apply a different skeleton on the result of the previous phase to combine the intermediate results (e.g., $reduce(\otimes)$ in $\mathcal{C}(reduce(\otimes))$). Finally, they might apply a $flatten$ skeleton to put the result in its original fine-grain format:

$$\begin{aligned}
&\mathcal{C}(map(\oplus)) \;=\; flatten \circ map(map(\oplus)) \circ dist \\
&\mathcal{C}(zip(\oslash)) \;=\; flatten \circ zip(zip(\oslash)) \circ dist \\
&\mathcal{C}(reduce(\otimes)) \;=\; reduce(\otimes) \circ map(reduce(\otimes)) \circ dist \\
&\mathcal{C}(butterfly(\otimes)) = flatten \circ map(butterfly(\otimes)) \circ butterfly(zip(\otimes)) \circ dist \\
&\mathcal{C}(\textit{rev-}butterfly(\otimes)) = flatten \circ butterfly(zip(\otimes)) \circ map(\textit{rev-}butterfly(\otimes)) \circ dist
\end{aligned} \tag{11}$$

The coarsening transformation of the *map-reduce* can be defined in two ways:

$$\begin{aligned}
&\mathcal{C}(\textit{map-}reduce(\oplus,\otimes)) = \mathcal{C}(reduce(\otimes) \circ map(\oplus)) = \mathcal{C}(reduce(\otimes)) \circ \mathcal{C}(map(\oplus)) \\
&\mathcal{C}(\textit{map-}reduce(\oplus,\otimes)) = reduce(\otimes) \circ map(\textit{map-}reduce(\oplus,\otimes)) \circ dist
\end{aligned} \tag{12}$$

For performance reasons, it is desirable to choose the second method in Eq. 12 as the first one might require intermediate storage for the result of $\mathcal{C}(map(\oplus))$.

Similarly, the coarsening transformation for *scan* can be defined as:

$$\begin{aligned}
\mathcal{C}(scan(\oplus)) = let\ \ & r_1 \leftarrow map(scan(\oplus)) \circ dist \\
& r_2 \leftarrow scan_{exclusive} \circ map(last)\ r_1 \\
in\ & flatten \circ zip(\Psi(\oplus))\ r_2\ r_1
\end{aligned} \tag{13}$$

In Eq. 13 $\Psi$ is a function of type $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$ and is defined as:

$$\Psi(\oplus)\ c\ [a_1, \ldots, a_n] = [a_1 \oplus c, \ldots, a_n \oplus c] \tag{14}$$

The coarsening transformation in Eq. 13 is more desirable than the *map-reduce* transformation listed in [26]. The reason is that the *reduce* operation used in [26] is defined in an inherently sequential form while in Eq. 13 we define the transformation in terms of other parallel skeletons.

**Limitation.** Similar to the approach in [26], our coarsening transformation is currently limited to input-independent skeletons which are list homomorphisms.

## 5 Composition Examples

The goal in this section is to show the expressivity and ease of programmability of our skeleton framework. As a case study we show the implementation of two NAS benchmarks [3] (Embarrassingly Parallel (EP) and Integer Sort (IS)) in terms of skeletons.

### 5.1 NAS Embarrassingly Parallel (EP) Benchmark

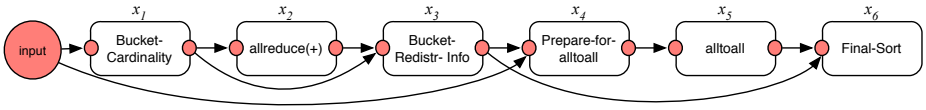
This benchmark is designed to evaluate an application with nearly no interprocessor communication. The only communication is used in the pseudo-random

number generation in the beginning and the collection of results in the end. This benchmark provides 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 help of the *map-reduce* skeleton. The *map* operator in this case generates $n$ pairs of uniform pseudo-random deviates $(x_j, y_j)$ in the range of $(0, 1)$, then checks if ${x_j}^2 + {y_j}^2 \leq 1$. If the check passes, the two numbers $X_k = x_j\sqrt{(-2log\ t_j)/t_j}$ $Y_k = y_j\sqrt{(-2log\ t_j)/t_j}$ are used in the sums $S_1 = \sum_k X_k$ and $S_2 = \sum_k Y_k$. The *reduce* operator computes the total sum for $S_1$ and $S_2$ and also accumulates the ten counts of deviates in square annuli.

### 5.2 NAS Integer Sort (IS) Benchmark

In this benchmark $N$ keys are sorted. The keys are uniformly distributed in memory and are generated using a predefined sequential key generator. This benchmark tests both computation speed and communication performance.




(a) Graphical representation of NAS IS flows.

```
integer-sort input =
  let
    x1 ← map(Bucket-Cardinality) input
    x2 ← allreduce(+) x1
    x3 ← zip2(Bucket-Redistr-Info) x1 x2
    x4 ← zip2(Prepare-for-alltoall) x3 input
    x5 ← alltoall x4
    x6 ← zip2(Final-Sort) x3 x5
  in
    x6
```

```
compose<flows::nas_is>(
  map(bucket_cardinality<int_t>()),
  allreduce<std::vector<int_t>>(),
  map(bucket_redistr_info()),
  zip<3>(prepare_for_alltoall<int_>(),
  alltoall<int_t>(),
  zip<3>(final_sort())
)
```

(b) The Nas IS composition.

Fig. 5: The NAS Integer Sort Benchmark

The IS benchmark is easily described in terms of skeletons as shown in Fig. 5. Similar to other skeleton compositions presented so far, the IS skeleton composition does not require any global synchronizations between the skeleton invocations and can overlap computation and communication easily. As we show later in the experimental results, avoiding global synchronizations results in better performance on higher core counts.

The IS benchmark skeleton composition as presented in Fig. 5 is based on the well-known counting sort. First, the range of possible input values are put into buckets. Each partition then starts counting the number of values in each bucket

(*map(Bucket-Cardinality)*). In the second phase, using an *allreduce* skeleton (defined in Fig. 4), the total number of elements in each bucket is computed and is globally known to all partitions. With the knowledge of the key distribution, a partitioning of buckets is devised (*map(Bucket-Redistr)*) and keys are prepared for a global exchange (*zip(Prepare-for-allToall)*). Then with the help of the *alltoall* skeleton (Fig. 4 and 7) the keys are redistributed to partitions. Finally, each partition sorts the keys received from any other partition (*zip(Final-Sort)*).

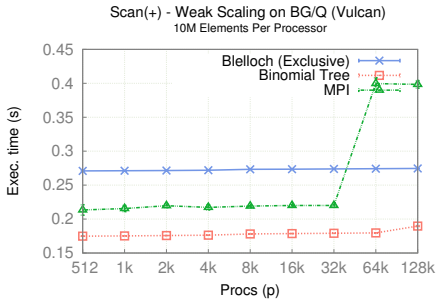
## 6 Performance Evaluation

We have evaluated our framework on two massively parallel systems: a 153,216 core Cray XE6 (HOPPER) and a 24,576 node BG/Q system (VULCAN). Each node contains a 16-core IBM PowerPC A2, for a total of 393,216 cores. Our results in Fig. 6 show excellent scalability for the *map*, *reduce*, and *scan* skeletons and also the NAS EP benchmark on up to 128k cores. These results show that the ease of programmability in the STAPL Skeleton Framework does not result in performance degradation and our skeleton-based implementation can perform as well as the hand-optimized implementations.

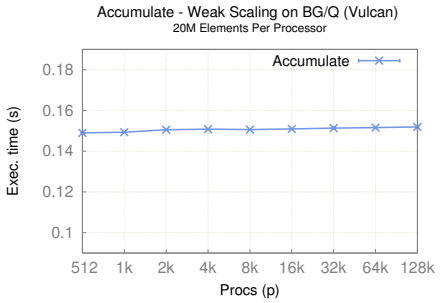
To show a more involved example, we present the result for the NAS IS benchmark. The IS benchmark is a communication intensive application and is composed out of many elementary skeletons such as *map*, *zip*, and *alltoall*. It is therefore a good example for the evaluation of composability in our framework. We compare our implementation of the IS benchmark to the reference implementation in Fig. 8(b).

Having an efficient *alltoall* skeleton is the key to success in the implementation of the IS benchmark. We have implemented *alltoall* in three ways (Fig. 7). Our first implementation uses the *butterfly* skeleton. The second is a flat *alltoall* in which all communication happen at the same level. The third method is a hybrid of *butterfly* and flat *alltoall*s. In both the flat and hybrid implementations, we use a permutation on the dependencies to avoid a node suffering from network congestion.

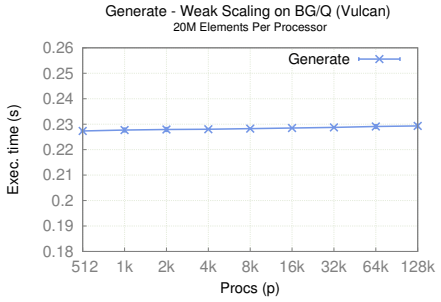
Our experiments show that the best performance for our skeleton-based implementation of the IS benchmark is achieved using the hybrid version in both C and D classes of the benchmark. Our implementation of the IS benchmark using the hybrid *alltoall* shows comparable performance to the hand-optimized reference implementation (Fig. 8(b)). We have an overhead on lower core counts which is due to the copy-semantics of the STAPL runtime system. The STAPL runtime system at this moment requires one copy between the user-level to the MPI level on both sender and receiver side. These extra copies result in a 30-40% overhead on lower core counts. However, this overhead is overlapped with computation on higher core counts. In fact, in the class D of the problem, our implementation is faster than the reference implementation. This improvement is made possible by avoiding global synchronizations and describing skeleton composition as point-to-point dependencies.



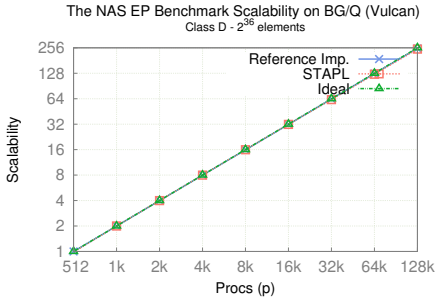

(a) Our inclusive binomial and exclusive *scans* (Section 3) vs. a hand-optimized MPI scan.




(b) Weak scalability of accumulate(+) as an example of the *reduce* skeleton.




(c) Weak scalability of Generate as an example of the *map* skeleton.




(d) Comparison of the scalability of the NAS EP benchmark with the reference implementation.

Fig. 6: Experimental results for elementary skeletons and the NAS EP benchmarks.

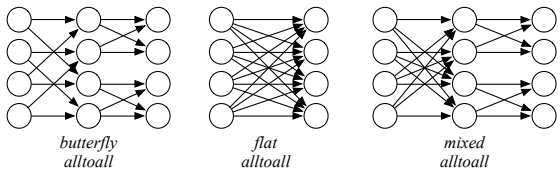
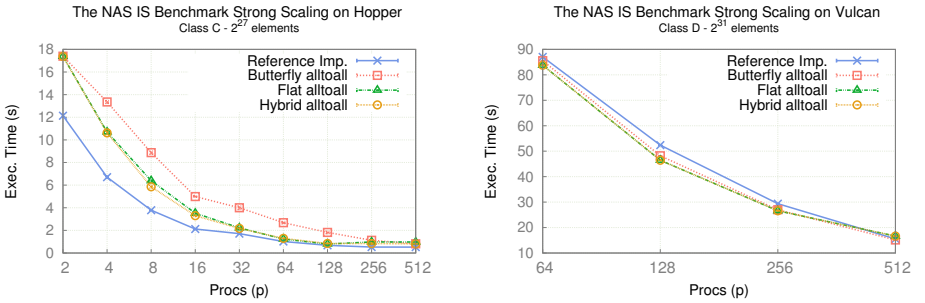



Fig. 7: The three versions of the *alltoall* representations used in the NAS IS benchmark.

## 7 Conclusions

In this paper, we introduced the STAPL Skeleton Framework, a framework which simplifies parallel programming by allowing programs to be written in terms of algorithmic skeletons and their composition. We showed the coarsening transformation on such skeletons, which enables applications to run efficiently on both shared and distributed memory systems. We showed that the direct mapping of skeletons to data flow graphs and formulating skeleton composition as data flow graph composition can remove the need for global synchronization. Our ex-




(a) Performance comparison of class C NAS IS benchmark with reference implementation.

(b) Performance comparison of class D NAS IS benchmark with reference implementation.

Fig. 8: The NAS IS benchmarks strong scaling results.

perimental results demonstrated the performance and scalability of our skeleton framework beyond 100,000 cores.

## 8 Acknowledgments

We would like to thank Adam Fidel for his help with the experimental evaluation.

## References

1. M. Aldinucci and S. Campa et al. Targeting distributed systems in fastflow. In *Euro-Par 2012: Par. Proc. Wkshps*, pages 47–56, 2013.
2. M. Aldinucci and M. Danelutto et al. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming Multi-core and Many-core Comp. Sys., Par. and Dist. Comp.*, 2011.
3. 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, Mail Stop T 27 A-1, Moffett Field, CA 94035-1000, USA, Dec. 1995.
4. G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, nov 1990.
5. Z. Budimlić and M. Burke et al. Concurrent collections. *Sci. Prog.*, 18(3):203–217, 2010.
6. A. Buss, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Houston, TX, USA, September 2010.
7. A. Buss, N. M. Amato, and L. Rauchwerger. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.
8. A. A. Buss, N. M. Amato, and L. Rauchwerger. Design for interoperability in STAPL: pMatrices and linear algebra algorithms. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Edmonton, Alberta, Canada, July 2008.

9. M. I. Cole. *Algorithmic skeletons: structured management of par. comp.* Pitman London, 1989.
10. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
11. J. Falcou and J. Sérot et al. Quaff: efficient c++ design for parallel skeletons. *Par. Comp.*, 32(7):604–615, 2006.
12. E. Gamma et al. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.
13. 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.
14. 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.
15. K. Hammond and M. Aldinucci et al. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In *Formal Methods for Components and Objects*, pages 218–236. Springer, 2013.
16. Harshvardhan, N. M. Amato, and L. Rauchwerger. The stapl parallel graph library. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Berlin Heidelberg, 2012.
17. C. A. Herrmann and C. Lengauer. Transforming rapid prototypes to efficient parallel programs. In *Patterns and skeletons for par. and dist. comp.*, pages 65–94. 2003.
18. W. M. Johnston and J. Hanna et al. Advances in dataflow programming languages. *ACM Comp. Surv. (CSUR)*, 36(1):1–34, 2004.
19. L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.
20. K. Matsuzaki and H. Iwasaki et al. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st international conference on Scalable information systems*, page 13. ACM, 2006.
21. M. McCool and J. Reinders et al. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.
22. J. P. Morrison. *Flow-Based Programming: A new approach to application development.* CreateSpace, 2010.
23. U. Müller-Funk and U. Thonemann et al. The münster skeleton library muesli–a comprehensive overview. 2009.
24. D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition.* Addison-Wesley, 2001.
25. P. Newton and J. C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, pages 167–177. ACM, 1992.
26. F. Rabhi and S. Gorlatch. *Patterns and skeletons for parallel and distributed computing.* Springer, 2003.
27. A. D. Robison. Composable parallel patterns with intel cilk plus. *Comp. in Sci. & Eng.*, 15(2):0066–71, 2013.
28. P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for mpi. In *Recent Advances in Par. Virt. Machine and Message Passing Interface*, pages 49–57. Springer, 2006.
29. G. Tanase, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.