NESTED PARALLELISM WITH ALGORITHMIC SKELETONS

A Thesis

by

ALIREZA MAJIDI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Timothy Davis
	Diego Donzis
Head of Department,	Scott Schaefer

August 2020

Major Subject: Computer Science and Engineering

Copyright 2020 Alireza Majidi

ABSTRACT

New trend in design of computer architectures, from memory hierarchy design to grouping computing units in different hierarchical levels in CPUs, pushes developers toward algorithms that can exploit these hierarchical designs. This trend makes support of nested-parallelism an important feature for parallel programming models. It enables implementation of parallel programs that can then be mapped onto the system hierarchy. However, supporting nested-parallelism is not a trivial task due to complexity in spawning nested sections, destructing them and more importantly communication between these nested parallel sections. Structured parallel programming models are proven to be a good choice since while they hide the parallel programming complexities from the programmers, they allow programmers to customize the algorithm execution without going through radical changes to the other parts of the program. In this thesis, nested algorithm composition in the *STAPL Skeleton Library* (*SSL*) is presented, which uses a nested dataflow model as its internal representation. We show how a high level program specification using *SSL* allows for asynchronous computation and improved locality. We study both the specification and performance of the STAPL implementation of *Kripke*, a mini-app developed by Lawrence Livermore National Laboratory. *Kripke* has multiple levels of parallelism and a number of data layouts, making it an excellent test bed to exercise the effectiveness of a nested parallel programming approach. Performance results are provided for six different nesting orders of the benchmark under different degrees of nested-parallelism, demonstrating the flexibility and performance of nested algorithmic skeleton composition in STAPL.

ACKNOWLEDGMENTS

First, I wish to express my sincere appreciation to my supervisor, Dr. Lawrence Rauchwerger, for his support and guidance throughout this work. I am also grateful to Dr. Nancy M. Amato, my initial co-advisor for her supports and feedbacks in my research.

During these years as a member of the Parasol Laboratory, I had the opportunity to work with many smart researchers and students. First, I am indebted to Mani Zandifar for his mentorship during my first years. Without Mani's work on design and implementation of Algorithmic Skeletons in STAPL library, this work couldn't be done. A big special thank to Dr. Timmie Smith and Dr. Nathan Thomas, the research staff of the Parasol lab. I cannot begin to express how much they helped me during the implementation and integration of this work in STAPL library. I would like to pay my special regards to Adam Fidel. I learned a lot from him, not only from his software engineering knowledge and practical suggestions, but from the refreshing talks we had during our coffee breaks. I would also like to extend my deepest gratitude to Dr. Ioannis Papadopoulos, for his constructive advises, and his work on support of nested parallelism in STAPL run-time system.

Finally, I would like to thank my committee members, Dr. Timothy A. Davis and Dr. Diego Donzis for dedicating considerable amount of time and effort into guiding me through this work.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES vi

LIST OF TABLES vi

1. INTRODUCTION AND LITERATURE REVIEW 1

1.1 Parallel Programming 1
1.2 Nested Parallelism 2
1.3 STAPL Overview 3
1.4 STAPL Skeleton Library 4
1.4.1 Parametric Dependencies 4
1.4.2 Skeleton Composition 5
1.4.2.1 Elem 5
1.4.2.2 Compose 6
1.4.2.3 Repeat 7
1.4.3 Skeleton Transformation 8

2. NESTED COMPOSITION 10

2.1 Algorithmic Skeletons 10
2.1.1 Definition 10
2.1.2 Specification 10
2.1.3 Execution 11
2.2 Containers 14
2.2.1 Nested Parallel Containers 15
2.2.2 Multi Level Presentation of Flat Containers 16

3. KRIPKE, MINI TRANSPORT BENCHMARK 18

3.1 Problem Description and Reference Implementation 18
3.2 Kripke Implementation in STAPL 20

4. EXPERIMENTAL RESULTS 23

4.1 Two Levels of Nested Parallelism 23
4.1.1 Single-node Performance 23
4.1.2 Multi-node Performance 24
4.2 Beyond Two Levels of Nested Parallelism 27

5. RELATED WORK 29

6. SUMMARY AND CONCLUSION 31

REFERENCES 32

LIST OF FIGURES

FIGURE Page

1.1 The STAPL library component diagram. 3
1.2 Parametric Dependencies for zip, $wavefront$, $reduce$ and $stencil$. 5
1.3 Building zip and $wavefront$ skeletons and their ports using *elem* operator. 6
1.4 Examples of compositional operators in STAPL skeleton library. 7
2.1 Example of skeletons nested composition in STAPL skeleton library. 11
2.2 Point-to-Point communication between skeletons in nested sections. 13
2.3 Examples of parallel containers with different nesting levels. 16
2.4 2 levels presentation of 2Dimensional flat container using *views*. 17
3.1 Container composition used for angular flux storage in STAPL. 21
3.2 Nested composition of skeletons to describe DZG and ZGD sweep kernels. 21
4.1 Single node strong scaling for all nesting orders. 25
4.2 Multi-node weak scaling for all nesting orders. 26
4.3 DGZ, DZG and ZDG kernels performance varying levels of nested parallelism. ... 28

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Parallel Programming

In the past few decades, despite advances in building high performance machines with high number of cores for parallel computing, how to express a program to run efficiently on these machines is still a challenge. Parallel programming models are always challenged by two main issues : 1. *Performance* 2. *Expressivity*.

Between these two challenges, *Performance* is usually the focus of the parallel programming libraries, which has led to develop of parallel programming models with low level APIs. Providing low level interfaces enables programmers to leverage optimizations like memory management techniques to address the performance issues in parallel programs. However, there has been a growing awareness that while low level programming models deliver a reasonable performance, expressing such a high performance parallel programs is a burdensome task.

The ability to compose parallel programs in order to form a new parallel program (*Composability*) is often missed in parallel programming models. For decades researchers have argued functional style programming is a promising approach to tackle this issue, since *Composability* is a first class feature in functional languages. However, programming in pure declarative style leads to inefficiency in memory management and hinders use of functional languages in high performance computing.

algorithmic skeletons introduced in [1], are high level parallel programming abstracts which try to address the gap between *Performance* and *Expressivity* in parallel programming models. algorithmic skeletons libraries handle the parallel programming complexity issues like memory management and synchronizations between execution units under the hood and provide high level abstract functions to programmers to express their parallel programs. However, *Composability* is still a feature that available libraries using algorithmic skeletons as their programming model have difficulty to support. This difficulty is coming from lack of generalized specification of these high

level abstracts which leads to two major issues. First, inability to define closed-form compositional operators to compose these high level abstracts. Second, it leads to use of global barriers for synchronizations between execution of these high level abstracts, which immediately hurts the performance of skeleton libraries.

1.2 Nested Parallelism

Nested parallelism is the the invocation of a parallel construct from within another parallel section, and it is a natural way of expressing algorithms with a hierarchical nature. Nested parallelism presents a promising approach to address the complexities of application development on modern, high performance computing systems. As from one side, the current trend in hierarchical design of computer architectures (memories and processors) provides opportunity to leverage these designs and provides more locality in computation in case of nested parallelism, and from the other side, many algorithms can benefit from it to have different levels of parallelism, while they can adjust the amount of parallelism in each level to get the best possible configuration.

Nested Parallelism tackles the problem of lacking enough parallelism in the first level, possible load imbalance in workload, temporary memory usage explosion, etc. However, the composition of nested parallel invocations is only beginning to find its way into parallel programming frameworks. Frameworks adopted nested parallelism face the same classic issues in its support: *Expressivity* VS *Performance*.

Composability becomes more vital feature in presence of nested parallelism. For example, languages desiring to mimic the syntax of sequential counterparts often adopt a recursive fork-join execution model, requiring barriers between successive nested parallel sections in the program. This model is sufficient for simple parallel applications with good locality and coarse grain parallel sections. However, more complex programs will suffer from poor scalability due to unnecessary global synchronizations between nested parallel sections.

Nested parallel execution is often additionally constrained by the underlying communication primitives. Some models are effectively limited to two levels of parallelism: one across shared memory nodes using MPI and one within a node using OpenMP or a similar library. These lower

level concerns become part of the higher level programming model, decreasing portability and reuse as restrictions are put both on communication between parallel sections and their placement in the system.

The *STAPL Skeleton Library* (SSL) [2, 3], includes a set of operators that enables the composition of a sequence of algorithmic skeletons into a common parallel section. By using a dataflow model as the internal representation of skeletons in SSL, the need for global barriers is removed, allowing asynchronous execution of algorithms with fine-grain, point-to-point synchronizations between dataflow nodes which greatly improves scalability. In the next section, an overview of the STAPLframework is provided, followed by an in depth introduction to SSLwhich is the parallel programming model of STAPLlibrary.

1.3 STAPL Overview

The *Standard Template Adaptive Parallel Library* (STAPL) [4] is a framework developed in C++ for parallel programming. It follows the generic design of the Standard Template Library (STL) [5], with extensions and modifications for parallelism. STAPL is a library, requiring only a C++ compiler (e.g., gcc) and established communication libraries such as MPI. An overview of its major components is presented in Figure 1.1.

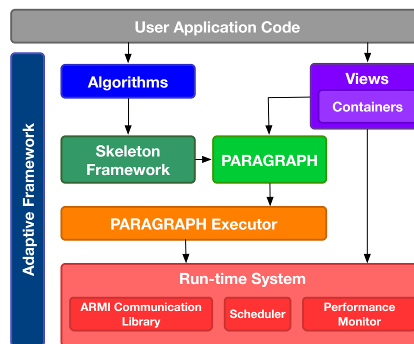


Figure 1.1: The STAPL library component diagram.

STAPL provides *parallel algorithms* and *distributed data structures* [6, 7] with interfaces similar to the STL. Instead of iterators, algorithms use *views* [8] which decouple the container interfaces from the underlying storage. The *skeletons framework* [2] allows the user to express an application as a composition of simpler parallel patterns (e.g., map, reduce, scan and others). These skeletons are instantiated at runtime as task dependence graphs by the `PARAGRAPH Executor`, STAPL's data flow engine. It enforces task dependencies and is responsible for the transmission of intermediate values between tasks.

1.4 STAPL Skeleton Library

The STAPL skeleton library is the component of STAPL used to specify parallel algorithms implemented in C++. In the following sections, we discuss how common composition operations found in functional languages are provided in our library. However, we first discuss *parametric dependencies* which are a basic building block of dependence patterns in STAPL skeletons.

1.4.1 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* (PDs). A parametric dependency defines the relation between the input elements of a skeleton and its output as a parametric coordinate mapping and an operation. Parametric dependency specifications for *map*, *zip* and a 2D *wavefront* skeleton are provided below in Equations 1.1, 1.2 and 1.3, respectively:

$$zip_pd_{<1>}(\oslash) = map_pd(\oslash) \equiv \{i \mapsto i, \oslash\} \tag{1.1}$$

$$zip\text{-}pd_{<k>}(\oplus) \equiv \{< \underbrace{i, \ldots, i}_{\text{k}} > \mapsto < i >, \oplus)\} \tag{1.2}$$

$$wavefront_pd_{2D}(\oplus) \equiv \{(i, j-1), (i-1, j) \mapsto (i, j), \oplus\} \tag{1.3}$$

For the *zip-pd* shown in Equation 1.2, which is general form of the *map-pd* (equation 1.1),

the value of element i of the output is computed by applying the $\oplus$ work-function (unary work-function $\oslash$ in case of *map-pd*), on the ith element of the inputs. For the *wavefront-pd*, the value of element (i, j) is result of applying the $\oplus$ work-function on the values of elements indexed by $(i-1, j)$ and $(i, j-1)$. In Figure 1.2, examples of *PD*s for zip, $wavefront$, $reduce$ and $stencil$ skeletons are shown. At runtime, *PD*s are expanded to generate data flow graph nodes by the *elem* composition operator, which we describe next.

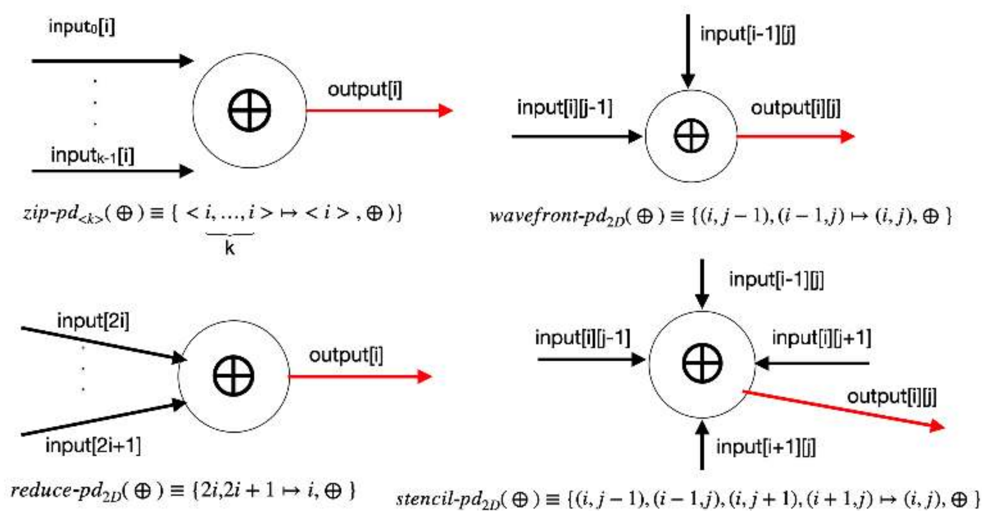
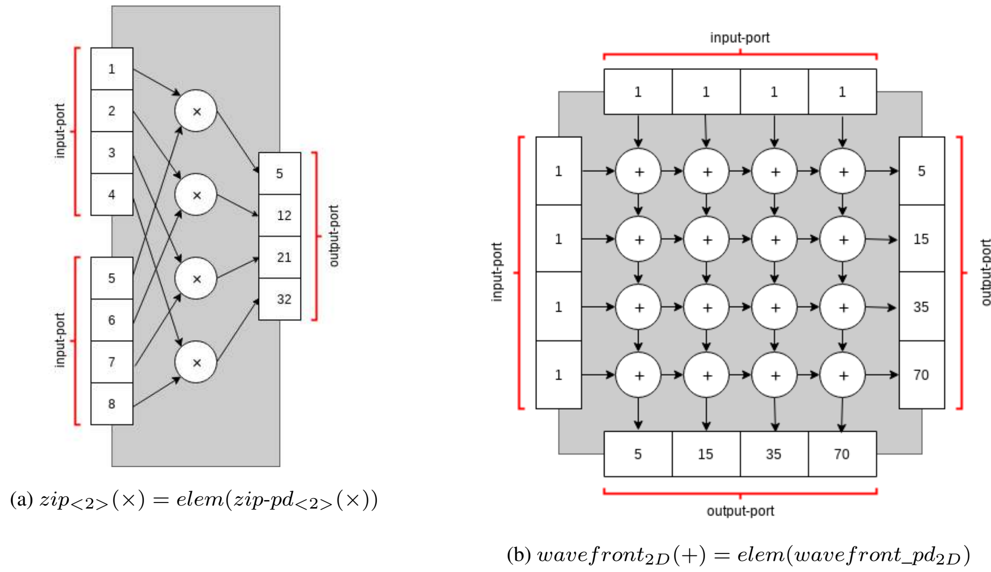


Figure 1.2: Parametric Dependencies for zip, $wavefront$, $reduce$ and $stencil$.

1.4.2 Skeleton Composition

1.4.2.1 Elem

Elem is the most basic operator employed to build skeletons. It expands the given parametric dependence over the domain of each input passed to the skeleton at run-time. The graph generated by the expansion of *PD*s is encapsulated by skeleton *port*s. A *port* provides an interface to access the inputs/outputs passed to/from skeletons. *Port*s play an important role as they allow the sending and receiving of data between skeletons without exposing the internal representation of skeletons (data-flow graph). For instance, the *zip* and *wavefront* skeletons are built by applying the *elem*



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

(b) $wavefront_{2D}(+) = elem(wavefront_pd_{2D})$

Figure 1.3: Building zip and $wavefront$ skeletons and their ports using *elem* operator.

operator to *zip-pd* and *wavefront-pd*, respectively:

$$zip_{<k>}(\oplus) = elem(zip_pd_{<k>}(\oplus)) \tag{1.4}$$

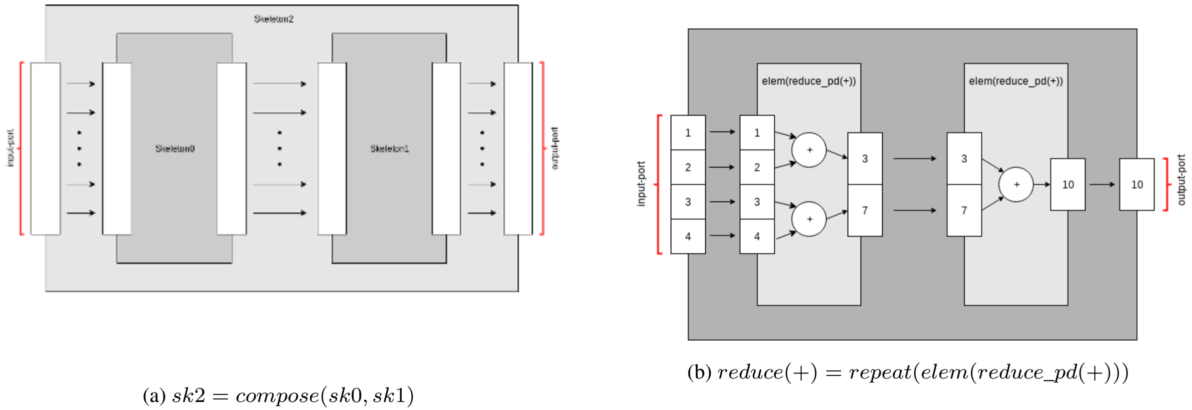
$$wavefront_{2D}(\oplus) = elem(wavefront_pd_{2D}(\oplus)) \tag{1.5}$$

Figure 1.3 shows the *port*s and data-flow graph generated for zip (1.3a) and wavefront (1.3b) skeletons using the *elem* operator.

1.4.2.2 Compose

Function composition, denoted by $h = f \circ g$, is the ability to create a new function of h by applying f to the input of h and then passing the output to g and then returning the result of that function invocation. The equivalent of the "$\circ$" is the *compose* operator in the STAPL skeleton library:

$$Skeleton2 = compose(Skeleton0, Skeleton1) \tag{1.6}$$



(a) $sk2 = compose(sk0, sk1)$

(b) $reduce(+) = repeat(elem(reduce_pd(+)))$

Figure 1.4: Examples of compositional operators in STAPL skeleton library.

As shown in Figure 1.4(a), *compose* connects the ports of skeletons passed to it as an argument in a functional composition manner. In this example, the input port of *sk0* is connected to the input port of *sk2* and the *sk1* output port will be the output port of the *sk2*. Finally, the output port of the *sk0* will be connected to the input port of *sk1*. Encapsulation of the data-flow graph representation of skeletons behind their *ports* makes it possible to access the output of skeletons without having any knowledge about the shape of the graph and still allows communication between graph nodes belonging to different skeletons without bulk synchronization between skeleton executions (the point-to-point synchronization shown in Figure 1.4(a) with black arrows).

1.4.2.3 Repeat

The *repeat* operator takes a skeleton as an argument and composes it in a functional manner with itself some number of times. This allows the expression of many skeletons which require tree-based or multilevel data-flow graph, such as *reduce* or *scan*. In Figure 1.4(b) the *reduce* skeleton is defined by applying the *repeat* operator on one level of the *reduce* skeleton, which is implemented by utilizing the *elem* operator and the reduce parametric dependency.

$$reduce(\otimes) = repeat(elem(reduce_pd(\otimes))) \quad (1.7)$$

These operators are closed under composition and enable the programmer to succinctly express

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-nd, stencil-nd

Table 1.1: List of all predefined skeletons and compositional operators in SSL.

complex algorithms in a manner that remains readily translatable to a data-flow graph at runtime for execution. List of all the predefined skeletons provided in SSL are provided in Table 1.1.

1.4.3 Skeleton Transformation

As we saw in previous section, skeletons are defined in terms of parametric data flow graphs, exposing parallelism in very fine-grained form. Execution of such a fine-grained data-flow graphs results in significant overhead on program execution due to lack of spatial and temporal locality and the overhead of task creation and its execution. The optimum granularity of data flow graphs depends on many factors. One of the important being the available execution units to execute parallel tasks. To overcome the overheads causing by using fine-grained data-flow graphs we use the coarsening transformation on algorithmic skeletonsin order to coarsen the data-graphs created from the corresponding Algorithmic Skeleton.

The coarsening transformation uses skeleton described earlier and transform them to skeletons which are suitable for parallel execution similar to the method used in [9] . In equation 1.8, the coarsening transformation ($\mathcal{C}$) for zip, $reduce$, $wavefront$ and $map\text{-}reduce$ skeletons are provided

$$\begin{aligned}
\mathcal{C}(zip(\oslash)) &= zip(zip(\oslash)) \\
\mathcal{C}(zip(\oslash)) &= zip(zip(zip...(\oslash))) \\
\mathcal{C}(wavefront(\oslash)) &= wavefront(wavefront(\oslash)) \\
\mathcal{C}(wavefront(\oslash)) &= wavefront(wavefront(wavefront...(\oslash))) \\
\mathcal{C}(reduce(\otimes)) &= reduce(\otimes) \circ map(reduce(\otimes)) \\
\mathcal{C}(map\text{-}reduce(\oplus, \otimes)) &= \mathcal{C}(reduce(\otimes) \circ map(\oplus)) = \mathcal{C}(reduce(\otimes)) \circ \mathcal{C}(map(\oplus)) \\
\mathcal{C}(map\text{-}reduce(\oplus, \otimes)) &= reduce(\otimes) \circ map(map\text{-}reduce(\oplus, \otimes))
\end{aligned} \tag{1.8}$$

As it is shown in equation 1.8, a common transformation of skeletons is nested composition of the same skeleton with itself arbitrary number of times (e.g. zip, $wavefront$). We will discuss how coarsening transformations is leveraged for experiments in chapter 4.

In the following chapters, the *nested* composition of algorithmic skeletons in SSL is presented (chapter 2), which uses nested dataflow model representation for description of nested parallelism in STAPL. We define nested skeleton composition, give examples of its use and describe how programs using it can be efficiently mapped onto the system for execution. Though the composition is static (i.e., strongly typed, recognized at compile time), the evaluation and mapping is dynamic, evaluated at runtime and executed using the asynchronous, nested parallelism support of the STAPL runtime described in [10]. Our implementation enables STAPL to support an arbitrary number of levels of nested parallelism and exploit point-to-point communication across nested sections.

In chapter 3, Kripke [11], a parallel transport mini-app developed at Lawrence Livermore National Laboratories is demonstrated which exposes good potentials to leverage the nested parallelism benefits. In same chapter, the STAPLimplementation of the kripke benchmark is described which is used to evaluate the proposed functionality later in chapter 4 by comparing against the reference implementation (MPI+OpenMP) from Lawrence Livermore National Laboratories.

2. NESTED COMPOSITION

2.1 Algorithmic Skeletons

2.1.1 Definition

Algorithmic skeletons are rooted in the functional programming paradigm, where functions are treated as first-class citizens and are the primary means of building programs. In functional languages, functions are higher-order, meaning they should be able to accept another function as an argument. Similarly, in our skeleton framework skeletons can receive other skeletons as arguments. This forms the basis for nested skeleton composition which in turn forms the basis of nested parallel algorithm specification in STAPL.

2.1.2 Specification

An ideal specification of skeletons should be oblivious of any nested composition. Consider specification of the map function in *Haskell* in equation 2.1. Except for specifying the type of the work-function ($a \rightarrow b$) for *map*, no additional detail is provided about whether the argument is a simple user-defined function or another skeleton made by composition operators in the language.

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \tag{2.1}$$

Skeleton specifications in SSL follow the same approach. Equation 2.2 shows that the general parametric dependency definition doesn't consider any difference between a regular user-defined operator and an algorithmic skeleton as a work-function specified by $\oplus$.

$$pd(\oplus) \equiv \{(\underbrace{i_0, \ldots, i_k}_{\text{inputs}}) \mapsto (i), \oplus)\} \tag{2.2}$$

This general specification of parametric dependencies makes algorithmic skeletons in SSL

```
1   // simple reduce skeleton
2   auto reduce_sk    = reduce(plus());
3
4   // 2 level skeletons nesting
5   auto map_sk       = map(reduce_sk);
6
7   // 3 level skeletons nestings
8   auto wavefront_sk = wavefront(map_sk);
9
10  // function composition of 3 level skeleton nesting
11  // with 2 level skeleton nesting
12  auto new_skeleton = compose(wavefront_sk, zip(zip(plus())));
```

Figure 2.1: Example of skeletons nested composition in STAPL skeleton library.

closed under nesting compositions, which allows arbitrary specification of nested composition of skeletons. For instance, figure 2.1 shows a small kernel written in SSL utilizing a skeleton with three levels of nested composition which is functionally composed with another skeleton with two levels of nesting.

Note that inputs to a nested skeleton need to reflect the algorithmic hierarchy. Specifically, a skeleton with n levels of nesting compositions need input views with n levels of addressing.

2.1.3 Execution

As mentioned earlier, there is uniform treatment in the framework of skeletons specified with or without nested composition. However, during evaluation and execution of a skeleton, we internally specialize the implementation based on whether the work-function passed to a skeleton is sequential or another algorithmic skeleton. In this section, we use a $wavefront_{2D}(zip(+))$ skeleton with two levels of parallelism, where a $zip(+)$ skeleton is passed as work-function of $wavefront_{2D}$ skeleton, to show how nested parallel execution is realized in the STAPL skeleton framework.

As shown in Equation 1.3, the $wavefront$ skeleton is specified by applying the *elem* operator to *wavefront-pd*. At run-time the *elem* operator traverses the domain of its input ports and uses the parametric dependency passed to it to spawn nodes, adding them to the underlying data-flow engine (PARAGRAPH).

The *parametric dependency* determines the corresponding inputs of each node and the work-

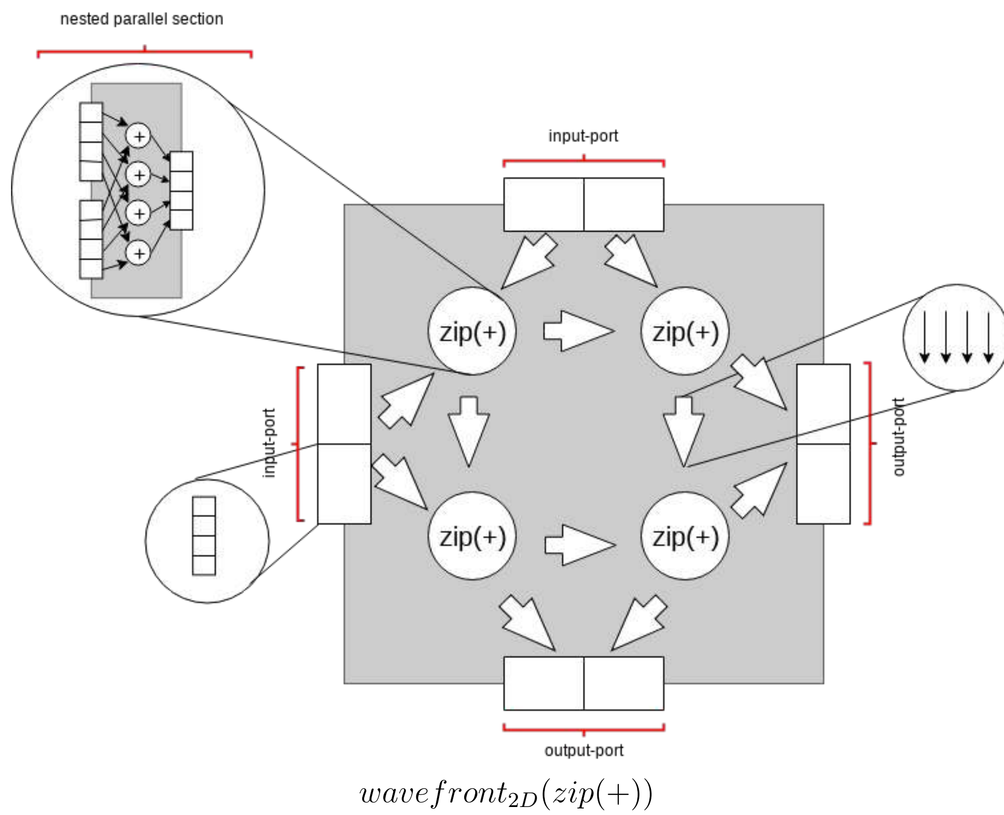
function which will be applied to its inputs during execution. As shown in Figure 2.2, applying the *elem* operator on $wavefront$-pd spawned 4 nodes (2×2) with the $wavefront$ pattern.

Up until this point everything about data-flow graph initialization is the same, regardless of whether the work-function of the node is an algorithmic skeleton or a simple sequential operator. At this point, however, we take a different path based on the type of work-function presented to the skeleton. If the work-function is a sequential operator, we add the node with its corresponding dependency information to the PARAGRAPH. In the presence of a nested skeleton, before adding the node to the PARAGRAPH, we tag the node to tell the PARAGRAPH to initialize a nested parallel section for execution of this node with its own instance of a nested PARAGRAPH. Creation of nested sections in STAPL is discussed in detail in [10].

In our example where the work-function is a $zip(+)$ skeleton we create a nested parallel section for each of the nodes in the data-flow graph of the *wavefront* skeleton at execution time, which is shown by dashed line in section of Figure 2.2. At each nested section, as soon as the inputs to each node of the $zip(+)$ skeleton are ready, the PARAGRAPH starts the execution of that node and sends its corresponding output to its consumer without waiting at a barrier at the end of each nested section for completion of other tasks. The manner in which the input and output port of $zip(+)$ skeletons are connected to each other and also how the input port and output port of the $wavefront$ skeleton are connected to each node is determined by the $wavefront$ dependence pattern, shown by bigger arrows in Figure 2.2.

While our example only shows two levels of nested-parallelism, nested composition of algorithmic skeletons in SSL allows expressing arbitrary levels of nested parallelism without using any global (or subgroup) synchronization. We discuss initial findings with additional levels of parallelism in chapter 4.

There are times when it might be preferable to serialize the execution of some levels of the nested parallel algorithm specification. Whether it is insufficient levels of the system hierarchy to map onto or insufficient parallelism in the section to practically exploit, there are times when one may want to suppress some of the available parallelism in the algorithm. For this purpose,



$wavefront_{2D}(zip(+))$

Figure 2.2: Point-to-Point communication between skeletons in nested sections.

we provide execution policy directives that are used to request serial execution *without changing the algorithm specification*. Many of the experimental configurations in chapter 4 use these directives to mimic the behavior of the reference implementation of *Kripke*. For instance, in $wavefront_{2D}(zip(+))$, user can specify sequential execution policy for execution of $zip(+)$ as shown in 2.3.

$$wavefront_{2D}(zip < seq\text{-}exec > (+)) \tag{2.3}$$

By using the data-flow model as the internal representation for skeletons, SSL can specify dependencies between nodes in data-flow graph at the most fine-grain level, exposing all the available parallelism in the program and enabling an asynchronous nested parallel execution.

2.2 Containers

In previous sections, we described how we compose algorithmic skeletons to describe algorithms with nested parallel patterns. In this section we focus on the representation of the containers provided as input to our program.

The first step to have hierarchical execution of an algorithm is providing the corresponding hierarchical input to the algorithm. More specifically, the input to the an algorithm which certain number of nesting levels should provide same levels of nesting which matches with the nesting levels of algorithm specification.

The natural way of providing a hierarchical input is using the nested composition of containers, for example considering std::vector<T> container from *C++* standard library where T denotes the container's elements type can provide arbitrary levels of nesting by replacing T with another container type e.g. std::vector<std::vector<T>>.

The same approach can be applied to the distributed containers, containers which support distribution of their elements across the system. `pContainers` in STAPL [12] distribute data across the system and provide data access operations that encapsulate the details of accessing distributed data. While providing the ability to compose parallel containers is not an easy task, a generic im-

plementation of parallel containers should support nested composition. We will describe how in in *STAPL* support nested composition of parallel containers in next section.

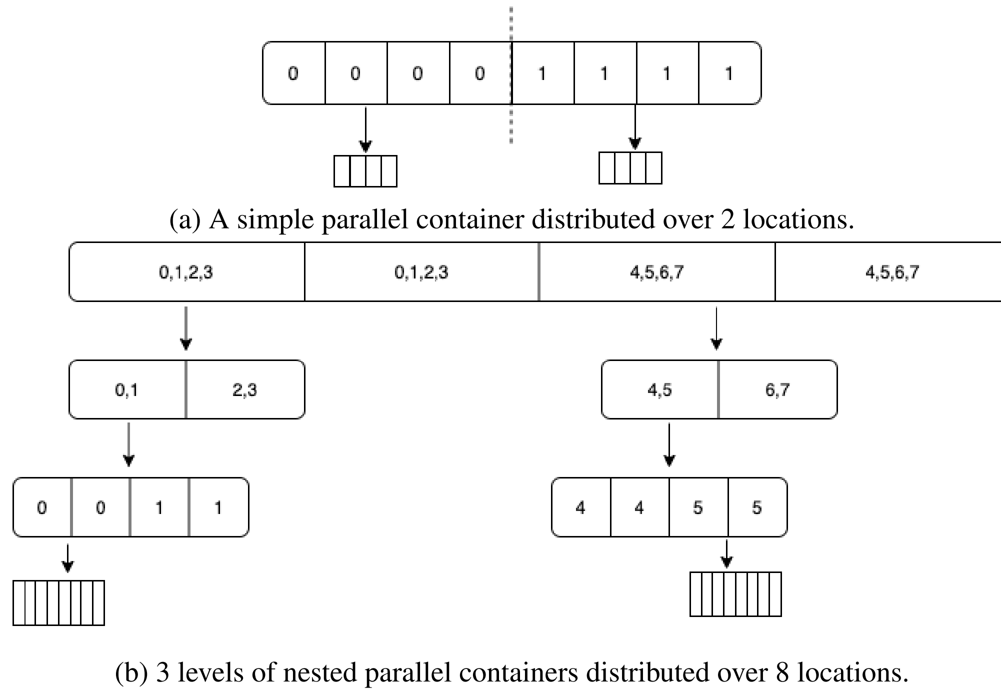
While nested composition of containers seems the natural way of providing hierarchical inputs to the program, it's not the only way to support the hierarchical behavior of the input. An alternative approach is wrapping the underlying flat container with an interface which provides the illusion of having a multilevel nested container. In Following we discuss each of these approaches in more detail.

2.2.1 Nested Parallel Containers

Nested composition of parallel containers is the natural way of describing parallel containers, hence it is much easier to reason about and program. The main challenge in describing a parallel container with multiple level of nested composition is how to partition the elements of a nested container in each level and how to group the locations who participates in creation of a container in an specific level. This information should be passed to the constructor of the container, which be used to determine the location-layout on each level and number of elements in each level.

The main advantage of using nested composition of parallel containers is the ability to ask the locality of the elements (where they are located, or in case of a nested container, which locations hold each of its elements) on each container without need of doing any extra computation since this information is already stored in the container. Examples of parallel containers in STAPL are shown in Figure 2.3. In Figure 2.3a a simple (non-nested) parallel container with 8 elements, distributed over 2 locations is provided, while in Figure 2.3b, a nested container with 3 levels of nesting partitioned over 8 locations has been shown. Labels on each element indicate locations that element is distributed over. At the lowest level, each element of the container resides a sequential containers in one location.

As we discussed in previous section, algorithmic skeletons use the locality information of the parallel containers to spawn parallel tasks in each level. Use of nested containers inhibits the ability to determine the level of nested parallelism dynamically at the execution time of the algorithm, which means this approach is not a good choice for dynamic applications e.g. graph application,



(a) A simple parallel container distributed over 2 locations.

(b) 3 levels of nested parallel containers distributed over 8 locations.

Figure 2.3: Examples of parallel containers with different nesting levels.

where the decision of parallel execution depends on run-time information. However, for static applications which nesting level is determined at the compile time, use of nested containers sounds a good solution, since it removes lot of inefficiency caused by calculation of locality of elements, and it provides a good abstraction at each level.

2.2.2 Multi Level Presentation of Flat Containers

Another way of supporting multilevel presentation of inputs to nested Algorithmic Skeletonis providing the behavior of nested containers using abstraction layers for each level over a flat container. *Views* [12] in STAPL provides the ability to reference a range of elements in the flat container at each level. *View*s allow the behavior of a nested container over a flat container without having to specify the locality and distribution of each container in different levels statically. This means views can be used to change the degree of nested parallelism at run-time without needing to modify the underlying container, which makes this approach a good candidate for dynamic nested parallelism execution.

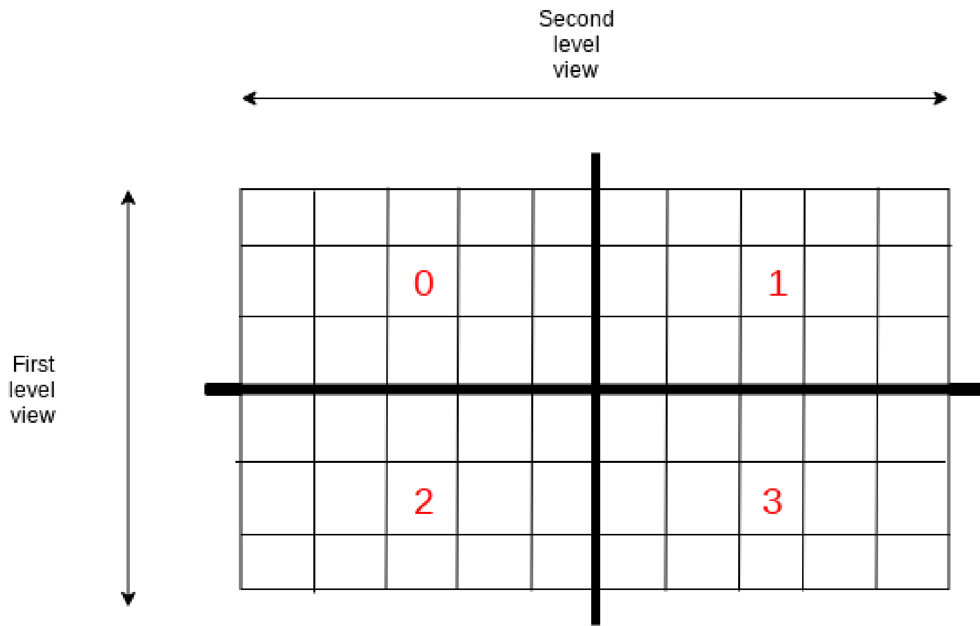


Figure 2.4: 2 levels presentation of 2Dimensional flat container using *views*.

However, the burden of specifying the locality information at construction time of the container, will be offloaded to later, in middle of the algorithm execution. If we consider views only functionality in specifying the sub-domains of underlying container, then we need to do some computation to figure out the locality of the elements in sub-domain and spawns task according to it. As shown in figure 2.4, a 2Dimensional array can be accessed through 2 levels of slicing, while the views in each level should be aware of the locality of elements in each sub-domain.

While use of *Views*, provides flexibility to decide on the level of nested parallelism at run-time, for applications which the input containers has structured layout and can be mapped to the machine hierarchy at compile time, nested parallel containers are better fit. We used the nested parallel containers approach to implement the *Kripke* in STAPL since it was enough to express the structured grids in *Kripke*, However, use of *Views* to represent these structured container could be a promising approach to test the decision of using nested parallelism at run-time.

3. KRIPKE, MINI TRANSPORT BENCHMARK

Kripke is a simple mini-app for 3D Sn deterministic particle transport[11]. Mini-apps are important tools that capture the essential performance behaviors of complex applications and allow rapid implementation of new approaches and exploration of new techniques and architectures. *Kripke*, like its parent application *ADRA*, is implemented with C++, OpenMP, and MPI. We have implemented *Kripke* v1.0 [13] using our skeleton framework and studied the performance of the sweep kernel with varying levels of parallelism and loop nesting orders.

3.1 Problem Description and Reference Implementation

Kripke solves the steady state form of the Boltzmann transport equation and stores the angular flux for every point in the phase space representing every point in the discretized angle, energy, and 3D spatial dimensions. The mini-app has three distinct kernels that perform a discrete-to-moments transformation (LTimes), a moments-to-discrete transformation (LPlusTimes), and a Sweep of the spatial domain for all discretized points in the energy and angle domains. The LTimes and LPlus-Times kernels are completely parallel map operations. We focus on the Sweep kernel and its unique composition challenges in this work.

Algorithm 1 Sweep Algorithm

```
 1: G all groups in the domain problem
 2: D all directions in the domain problem
 3: Z_p zone-set assigned to the current task
 4: procedure SWEEP-SOLVER
 5:     for each G_p in G do
 6:         for each D_p in D do
 7:             sweep({G_p, D_p, Z_p})
 8:         end for
 9:         send/receive local sweep results between MPI tasks
10:     end for
11: end procedure
```

The discretized problem phase space ($G \cdot D \cdot Z$) is partitioned into P subsets, identified by $\{G_p, D_p, Z_p\}_{p \in P}$. Each of these subsets are assigned to an MPI task in the reference implementation. MPI tasks operate on sub-domain of the problem. The choice of the data-layout for storing these subsets and corresponding choice of OpenMP loop-level threading for on-node parallelism greatly affects the performance of the application and is the primary research interest for the mini-app. The reference implementation supports all six layout orders and provides computational kernels written for each layout. The layout/loop nesting order is referred to by the strings (*DGZ*, *DZG*, *GZD*, *GDZ*, *ZGD* and *ZDG*) that indicate the order of the loops in the Sweep kernel. The Sweep kernel for the DGZ and ZGD nestings are provided as examples in Algorithms 2 and 3, respectively. The *diamond-difference* computation and more details about Sweep algorithm is provided in [11].

Algorithm 2 Sweep Kernel for DGZ nesting order

```
1: procedure Sweep_DGZ({G_p, D_p, Z_p})
2:     for each d in D_p do
3:         for each g in G_p do
4:             for z_k in range Z_pk do
5:                 for z_j in range Z_pj do
6:                     for z_i in range Z_pi do
7:                         diamond-difference computation
8:                     end for
9:                 end for
10:            end for
11:        end for
12:    end for
13: end procedure
```

The partitioning of the problem phase space across MPI tasks in the reference implementation partitions the spatial dimensions only. The result is that the sweep kernel is a sweep of the spatial domain that is distributed across the nodes of the system. Within each node, OpenMP is used to parallelize the loop iterating over the energy or angle domains, depending on the layout chosen.

Algorithm 3 Sweep Kernel for ZGD nesting order

```
1: procedure Sweep_ZGD({G_p, D_p, Z_p})
2:     for z_k in range Z_pk do
3:         for z_j in range Z_pj do
4:             for z_i in range Z_pi do
5:                 for each g in G_p do
6:                     for each d in D_p do
7:                         diamond-difference computation
8:                     end for
9:                 end for
10:            end for
11:        end for
12:    end for
13: end procedure
```

The outermost of the direction or energy loops is the loop that is parallelized using OpenMP in the reference implementation. In the STAPL implementation, we follow the same partitioning of the spatial domain across nodes and explore parallelization at multiple levels in processing a partition of the phase space on a node.

The sweep algorithm across MPI tasks is provided in Algorithm 1. As mentioned above, the sweep is a nested computation since *Zones* are partitioned into P sets by partitioning the spatial domain only. The outer sweep is performed for each direction set (a grouping of directions within the same octant) and energy group set (a grouping of consecutive energy groups). The on-node operation for the Sweep is the call to the sweep function in line 7. The communication of angular flux values from the on-node sweep computation is line 9 in Algorithm 1.

3.2 Kripke Implementation in STAPL

The Sweep implementation of *Kripke* in STAPL [14] follows the same approach for decomposing the domain problem into set of sub-domains and doing two sweeps, one over all subdomains and a nested sweep on each sub-domain. In the STAPL implementation, STL containers are replaced by `pContainers`. The STAPL container framework supports composition of `pContainers` which makes them a natural fit for expressing applications with nested-parallelism.

```
1  // storing angular flux storage in
2  // a 5D space (z_i, z_j, z_k, d, g)
3  using zoneset_container = multiarray<5, double>;
4
5  // decomposition of zones in a 3D geometry space
6  using zonesets_container = multiarray<3, zoneset_container>;
7
```

Figure 3.1: Container composition used for angular flux storage in STAPL.

```
1   // sweep kernel for DGZ nesting order
2   auto DGZ_sweep_kernel =
3       wavefront(                                  // sweep over zone-sets
4           zip(                                    // for loop over groups
5               zip(                                // for loop over directions
6                   wavefront(diamond_difference_wf) // sweep over each zone-set
7                   )));
8
9   // sweep kernel for ZGD nesting order
10  auto ZGD_sweep_kernel =
11      wavefront(                          // sweep over zone-sets
12          wavefront(                      // sweep over each zone-set
13              zip(                        // for loop over groups
14                  zip(diamond_difference_wf) // for loop over directions
15                  )));
```

Figure 3.2: Nested composition of skeletons to describe DZG and ZGD sweep kernels.

Figure 3.1 describes the data structure to store the angular fluxes in the STAPL implementation of *Kripke* using a $multiarray$ container, a generic multidimensional container in the STAPL container framework.

The Sweep kernel is implemented by nested composition of algorithmic skeletons provided in SSL. A *wavefront* skeleton captures the sweep pattern, and the *zip* skeleton is employed to process the energy and direction domains. In Figure 3.2, sweep skeletons for two nesting orders, *DGZ* and *ZGD*, are shown. In the first level of the composition, *wavefront* is used to sweep across all the *ZoneSets*, regardless of the nesting order. Based on the nesting order chosen, the work-function passed to wavefront skeleton will differ.

The use of SSL algorithmic skeletons to describe the sweep algorithm has several advantages over use of low-level libraries like MPI and OpenMP, besides programming abstraction and concise specification: First, parallelizing the second sweep over each *ZoneSet* using OpenMP is not a

trivial task. However, use of the data-flow graph representation as an internal model for skeletons enables parallelizing skeletons regardless of the parallel library chosen for execution. This means the second sweep in the algorithm could be considered as a candidate to be executed in parallel. Furthermore, SSL supports parallel and sequential implementation for all provided skeletons, which allows the programmer to test different execution policies to find the best configuration. Finally, using SSL Algorithmic Skeletons enables nested parallel execution beyond the common two levels, which enables taking advantage of the hierarchical design of new computer architectures.

4. EXPERIMENTAL RESULTS

4.1 Two Levels of Nested Parallelism

In this section, we compare the performance of the *Kripke* reference code with the STAPL implementation. All experiments are performed on a Cray XK7m-200 with twenty-four compute nodes of $2.1GHz$ AMD Opteron Interlagos 16-core processors. Twelve nodes are single socket with $32GB$ RAM, and twelve are dual socket with $64GB$ RAM. We use *gcc 4.9* with the *O3* optimization flag and *craype-hugepages2M* module. We perform two sets of experiments, one exercising single node scaling and one showing scaling across multiple nodes. In both experiments, the performance of the STAPL implementation of *Kripke*'s sweep computation is compared with that of the reference code.

4.1.1 Single-node Performance

For the single node performance study all zones are decomposed into just one *ZoneSet*, meaning that there is no sweep over the *ZoneSets* (one element). The reference code uses OpenMP, so we configure the STAPL runtime to use OpenMP as well. As a result, strong scaling results show the performance of local sweep of different kernels with a varying number of threads used on the node. The problem test used in [11] has 12 $\times$ 12 $\times$ 12 spatial zones, 8 *DirectionSets*, 12 *Directions* per set (96 total *Directions*) and 1 *GroupSet* with 64 energy groups. The authors refer to it as the KP0 configuration. We use KP0 and also define a KP0$'$ configuration, increasing the spatial zones to 20 $\times$ 20 $\times$ 20 , number of directions to 348 (48 directions per DirectionSet) and number of groups to 128. Figures 4.1a and 4.1b show strong scaling results for all 6 different kernels with the two configurations of KP0 and KP0$'$, respectively. For the single node study we use a larger, 32 core node.

For the *DGZ* and *DZG* kernels with the KP0 configuration, STAPL sweep stops scaling after 12 threads since we are parallelizing the second level skeleton (*zip*) corresponding to D, while the reference code parallelizes the third level loop, G, using OpenMP. However, for the KP0$'$

configuration, due to larger number of directions, scaling continues for the STAPL version and matches the reference code's behavior.

The *GZD* and *GDZ* kernels show the same behavior for the small configuration of *KP0* and the larger configuration of *KP0'*. However, performance of the STAPL implementation of sweep is more sensitive when the program stops scaling, due to a higher overhead of parallelization, which needs to be investigated and optimized.

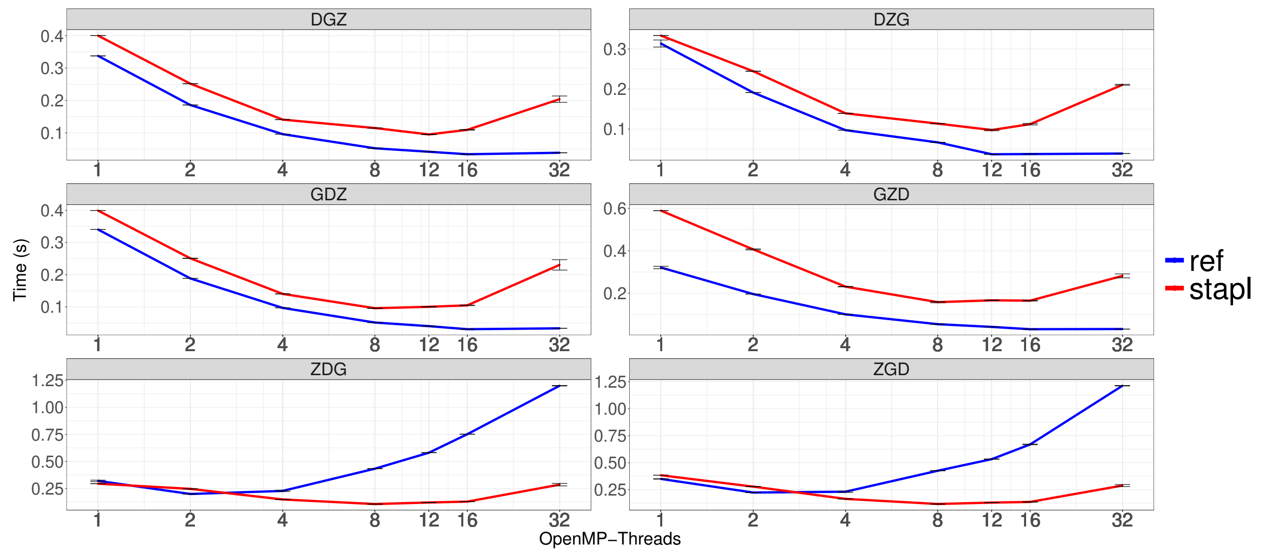
For kernels starting with Z loops (*ZGD* and *ZDG*), the ability to parallelize the inner *wavefront* skeleton (local sweep) in STAPL version allows scaling to a higher number of threads, while the reference code doesn't scale after 4 and 8 threads, respectively, due to a lack of this functionality.

4.1.2 Multi-node Performance

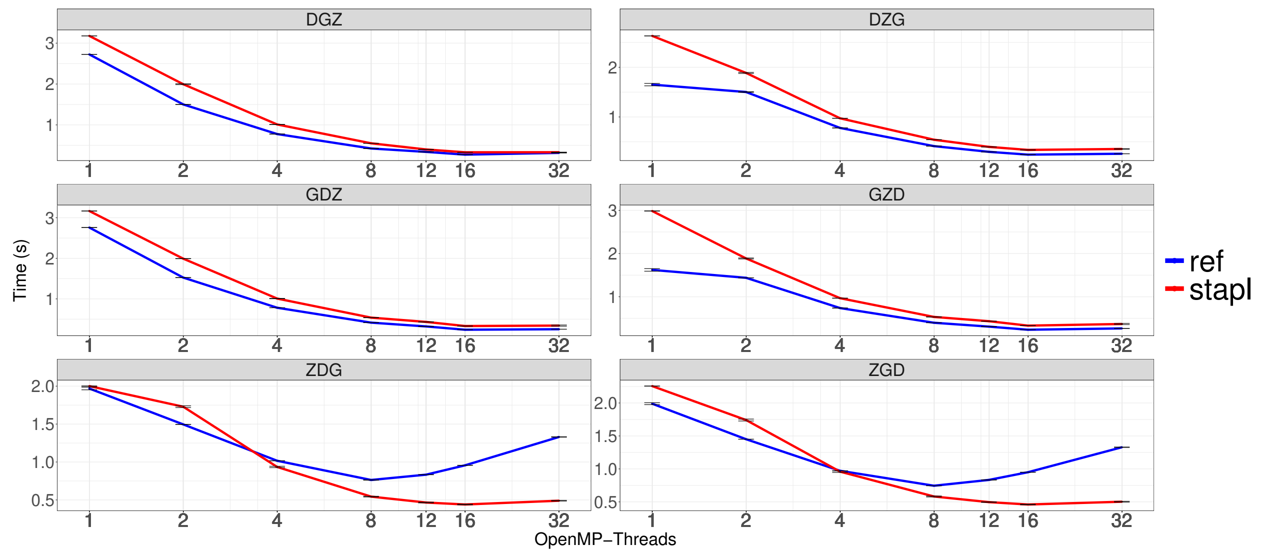
A weak scaling study similar to that in [11] is performed for investigating the scalability of the outer sweeps over the *ZoneSets*. All *ZoneSets* have the *KP0* configuration for Figure 4.2a and the *KP0*′ configuration for Figure 4.2b. Each *ZoneSet* is assigned to one MPI task. We have run this experiment for all kernels up to 64 MPI tasks, each with 8 threads (512 cores in total), to compare the sweep algorithm implemented in STAPL with the reference implementation.

Based on the kernel chosen for MPI scaling, the STAPL sweep is either faster or slower than the reference implementation. The choice of 8 threads per *ZoneSet* is not the best optimal case for on-node computation as shown in on-node scaling study. While this number was a reasonable basis of comparison and supports a large number of possible configurations, we intend to investigate different configurations of the node's cores further.

As can be seen in Figure 4.2, the STAPL implementation of sweep shows much less variability in scaling results compared to the reference code. This appears to be due the fact that STAPL is able to parallelize the inner wavefront in the nested section, while the reference's implementation restricts it to parallelizing the next lower loop level, resulting in more parallel sections with smaller granularity and more global synchronizations.

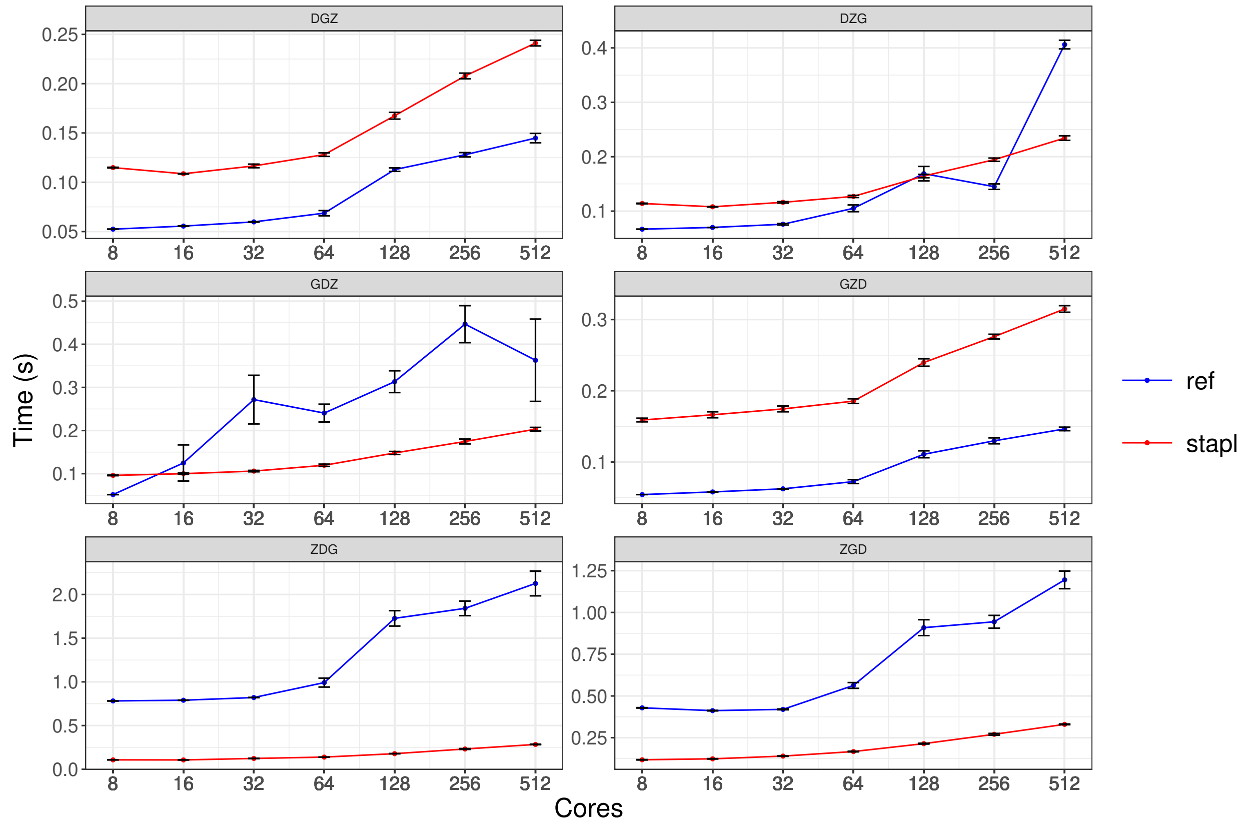


(a) KP0 configuration

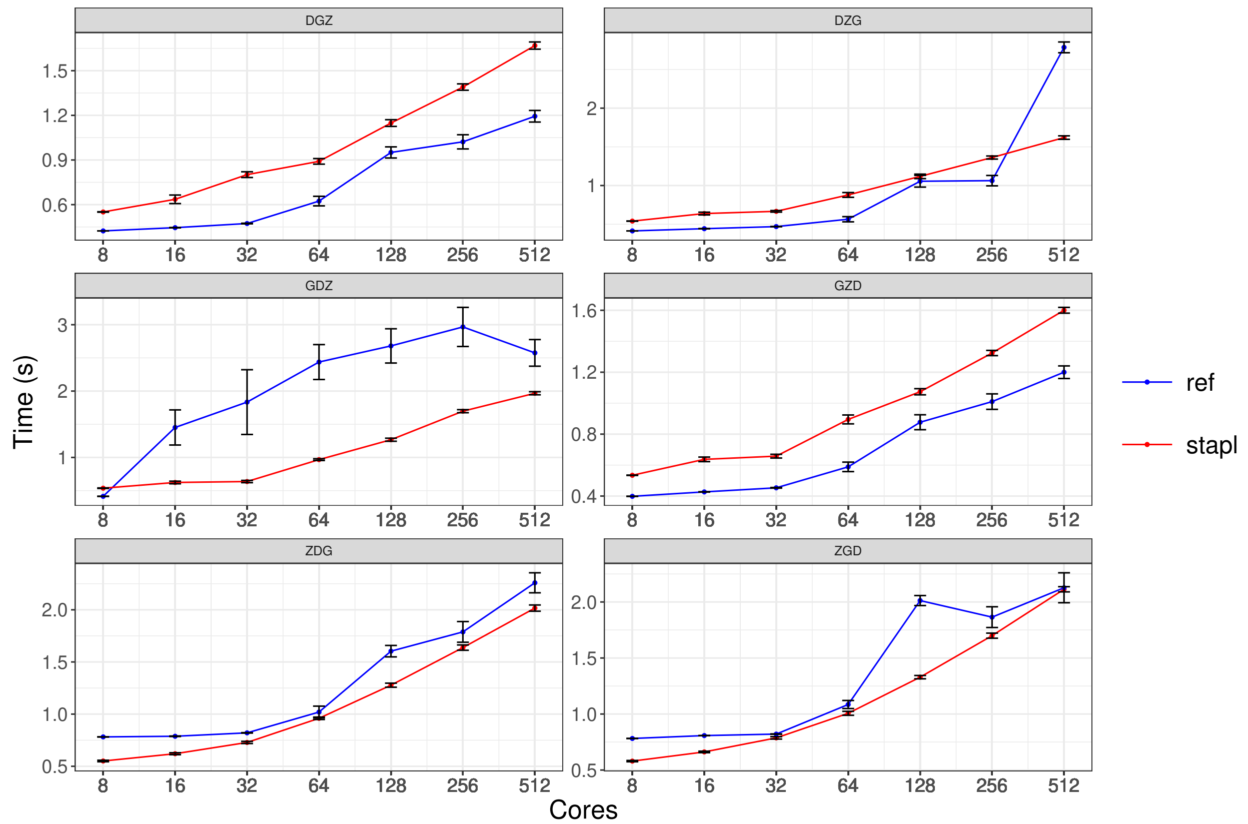


(b) KP0’ configuration

Figure 4.1: Single node strong scaling for all nesting orders.



(a) KP0 configuration



(b) KP0’ configuration

Figure 4.2: Multi-node weak scaling for all nesting orders.

4.2 Beyond Two Levels of Nested Parallelism

In previous section, we compared the kripke implementation in STAPL with the reference implementation. Given the abstraction of specifying nested parallelism from nested execution in SSL the STAPL implementation provides the ability to exploit more than two levels of parallelism. This ability is prohibited in reference implementation, due to complexity of the code for parallelizing the $wavefront$ pattern using $OpenMP$ and loop carried dependency in sweep pattern.

In this section we investigate the performance of sweep kernels under different degree of nested parallelism. As it shown in the previous section, creating more parallel tasks than available parallelism in one level hurts the performance. In order to investigate the performance of the nested parallelism implementation with more than two nested levels, we have chosen the input configuration so that there is enough parallelism available in every level. Removing the limitation on available parallelism imposed by the input size, allows us to investigate the benefits of nested parallelism when the available parallelism in each level is limited by the pattern of execution on that level.

We have picked three kernels (DGZ, DZG and ZDG) out of six possible kernels since they present all the different nesting order of $wavefront$,zip,zip skeletons and there is enough parallelism in each level. Figure 4.3 shows the performance results of the three chosen kernels using 8 cores under different degrees (1 level to 3 levels) of nested parallelism and locations layout. Points' labels indicate the speedup of the corresponding configuration over the sequential execution. As expected when there is enough available parallelism, DGZ and DZG kernels best configurations are when we use all the available execution units (specified by locations) in the first level. However, for ZDG kernel, using all the locations in first level is the worst configuration since the $wavefront$ pattern doesn't expose as much parallelism as the zip pattern does. For ZGD kernel the best configurations are when the locations are distributed among different levels.

The fact that for each nesting orders the best configuration depends on how much parallelism is exposed by the algorithm pattern in each level reveals the benefit of being able to specify nested parallelism execution in different nested levels of program execution.

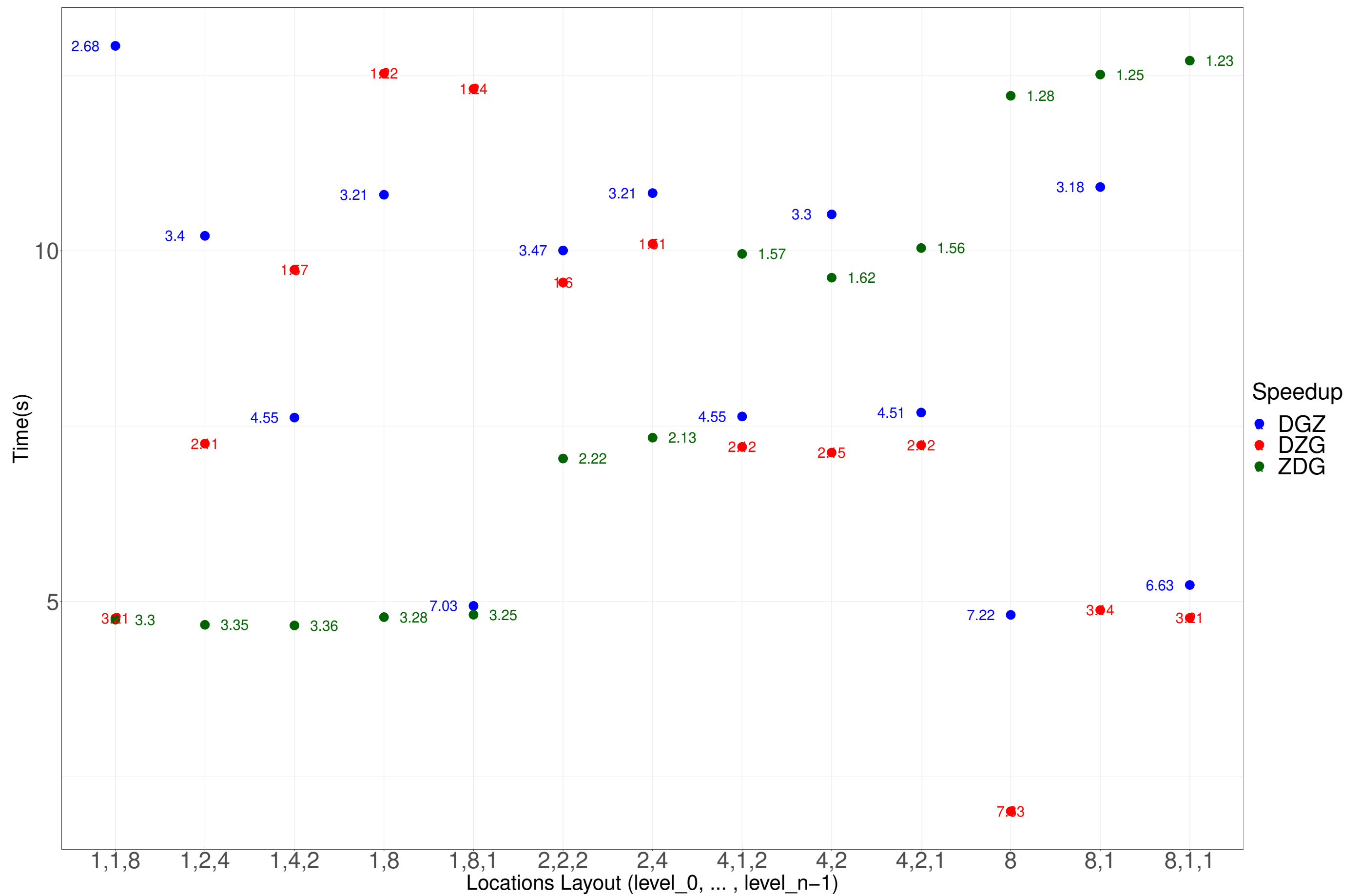


Figure 4.3: DGZ, DZG and ZDG kernels performance varying levels of nested parallelism.

5. RELATED WORK

OpenMP [15] supports nested parallelism using the fork-join model. When a thread inside a nested sections finishes, it waits at a global barrier in the nested section for other threads to finish. Inefficiencies can arise due to the global barrier at the end of each nested section. This has lead to the introduction of the *Collapse* keyword in OpenMP-3.0, which flattens the nested parallelism.

Several frameworks employ MPI's ability to partition the MPI communication groups into subgroups for nested execution. NestStep [16] uses this ability to partition communications groups into subgroups and run nested sub-supersteps on these subgroups asynchronously. However, the parent group needs to wait for all subgroups to finish their supersteps before going to next superstep, which can degrade performance. [17] and [18] follow the same approach in assigning nested parallel sections to sub-groups of processing elements.

MPI + OpenMP is a common approach to express two level of parallelism in benchmarks [11, 19] and [20]. However, this approach usually results in complex codes which are hard to maintain and are restricted to two levels of parallelism.

Among the frameworks which support task-level parallelism, Cilk [21] and TBB [22] allow spawning nested tasks where the system is responsible for mapping nested sections to the machine, which degrades the performance due to loss of locality. X10 [23], Habanero-Java [24], HPX [25], and Fortress [26] allow users to control task-placement. However, as tasks are independent, there is no communications between nested sections.

Legion [27] partitions memory into regions for spawning independent tasks using its dynamic machine mapping model to improve the locality of tasks. However, there is no support for dependencies between nested tasks, and it is limited to task level parallelism.

Nested parallelism is adopted in GPU computing and programming models [28, 29], which allows spawning nested kernels when there is enough parallelism available to increase the device utilization.

Among the previous frameworks with *skeleton* specifications with nested composition support,

[30] and [31] only support nesting of task-parallel skeletons and do not explicitly address data communication between nested sections. Skeleton frameworks presented in [32] and [33] support two levels of nesting for data-parallel skeletons. However, due to the use of a $master/slave$ scheme, their approach is not scalable, especially on distributed systems.

[34] proposes a construct for composition that describes the point-to-point dependency between partial outputs of algorithms in a nested data-flow model to analyze the complexity of algorithms with nested parallelism. [35] presented a memory management technique based on a defined property for memory to improve the performance of functional languages in presence of nested parallelism.

6. SUMMARY AND CONCLUSION

In this work, an implementation of nested parallelism using STAPL is presented, which allows the expression of arbitrary levels of parallelism using the nested composition of algorithmic skeletons. Choosing algorithmic skeletonsas a programming model allows the separation of specifying nested composition in the program from its execution and the restrictions imposed by low level APIs.

The ability to parallelize different nesting levels in program, allows the programmer to choose the best configuration based on the available parallelism in each level dictated by the algorithm pattern itself and also the input configuration. Choosing a data-flow graph as internal representation of skeletons removes the need for barriers between nested parallel sections, leading to a fully asynchronous implementation of nested parallelism.

Given this variety of platforms and computing nodes, using structured programming models such as algorithmic skeletons, allows programmers to provide better insight into how nested parallelism can help applications cope with the deepening hierarchies and heterogeneity present in modern HPC architectures.

REFERENCES

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

[2] M. Zandifar, M. A. Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM International Conference on Supercomputing (ICS)*, ICS '15, (New York, NY, USA), pp. 415–424, ACM, 2015. Conference Best Paper Award.

[3] 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.

[4] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: standard template adaptive parallel library," in *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, (New York, NY, USA), pp. 1–10, ACM, 2010.

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

[6] G. Tanase, A. A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL parallel container framework," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pp. 235–246, 2011.

[7] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "The STAPL Parallel Graph Library," in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 46–60, Springer Berlin Heidelberg, 2012.

[8] A. A. Buss, A. Fidel, Harshvardhan, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, “The STAPL pview,” in *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, pp. 261–275, 2010.

[9] C. A. Herrmann and C. Lengauer, “Transforming rapid prototypes to efficient parallel programs,” in *Patterns and skeletons for par. and dist. comp.*, pp. 65–94, 2003.

[10] I. Papadopoulos, N. Thomas, A. Fidel, D. Hoxha, N. M. Amato, and L. Rauchwerger, “Asynchronous nested parallelism for dynamic applications in distributed memory,” in *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, (Berlin, Heidelberg), pp. 106–121, Springer-Verlag, 2016.

[11] L. B. N. Laboratory, U. S. D. of Energy, U. S. D. of Energy. Office of Scientific, and T. Information, *Kripke - a Massively Parallel Transport Mini-App*. United States. Department of Energy., 2015.

[12] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, “STAPL: A Standard Template Adaptive Parallel C++ Library,” in *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, (Bucharest, Romania), Jul 2001.

[13] “Co-design: Kripke, https://computation.llnl.gov/projects/co-design/kripke.”

[14] A. Majidi, N. Thomas, T. Smith, N. Amato, and L. Rauchwerger, “Nested parallelism with algorithmic skeletons,” in *Languages and Compilers for Parallel Computing - 31st International Workshop, LCPC 2018, Revised Selected Papers* (M. Hall and H. Sundar, eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 159–175, Springer, Jan. 2019. 31st International Workshop on Languages and Compilers for Parallel Computing, LCPC 2018 ; Conference date: 09-10-2018 Through 11-10-2018.

[15] OpenMP, ARB, "OpenMP Application Program Interface," specification, 2011.

[16] C. W. Keßler, "NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model.," *The Journal of Supercomputing*, vol. 17, no. 3, pp. 245–262, 2000.

[17] U. Consortium, *UPC Language Specifications V1.2*, 2005. `http://www.gwu.edu/~upc/publications/LBNL-59208.pdf`.

[18] J. Mellor-Crummey, L. Adhianto, I. W. N. Scherer, and G. Jin, "A new vision for coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, (New York, NY, USA), pp. 5:1–5:9, ACM, 2009.

[19] F. Cappello and D. Etiemble, "MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, (Washington, DC, USA), IEEE Computer Society, 2000.

[20] J. Sillero, G. Borrell, J. Jiménez, and R. D. Moser, "Hybrid openMP-MPI Turbulent Boundary Layer Code over 32K Cores," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, (Berlin, Heidelberg), pp. 218–227, Springer-Verlag, 2011.

[21] A. D. Robison, "Composable parallel patterns with Intel Cilk Plus," *Comp. in Sci. and Eng.*, vol. 15, no. 2, pp. 0066–71, 2013.

[22] J. Reinders, *Intel threading building blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[23] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing," in *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY, USA), pp. 519–538, ACM Press, 2005.

[24] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The New Adventures of Old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, (New York, NY, USA), pp. 51–61, ACM, 2011.

[25] T. Heller, H. Kaiser, A. Schäfer, and D. Fey, "Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, (New York, NY, USA), pp. 1:1–1:8, ACM, 2013.

[26] G. L. S. Jr., E. E. Allen, D. Chase, C. H. Flood, V. Luchangco, J.-W. Maessen, and S. Ryu, "Fortress (Sun HPCS Language).," in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), pp. 718–735, Springer, 2011.

[27] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–11, IEEE Computer Society Press, Nov 2012.

[28] "The nvidia cuda toolkit, https://developer.nvidia.com/cuda-toolkit."

[29] "The opencl specification, https://www.khronos.org/registry/opencl/specs/2.2/html/opencl-api.html."

[30] D. G. Jocelyn SÃl'rot, "Skeletons for parallel image processing: an overview of the skipper project," 12 2002.

[31] A. Benoit and M. Cole, "Two fundamental concepts in skeletal parallel programming," in *Computational Science – ICCS 2005* (V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, eds.), pp. 764–771, Springer Berlin Heidelberg, 2005.

[32] G. Michaelson, N. Scaife, P. Bristow, and P. King, "Nested algorithmic skeletons from higher order functions," 2000.

[33] M. Hamdan, G. Michaelson, and P. King, "A scheme for nesting algorithmic skeletons," 10 1998.

[34] D. Dinh, H. V. Simhadri, and Y. Tang, "Extending the nested parallel model to the nested dataflow model with provably efficient schedulers," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pp. 49–60, ACM, 2016.

[35] S. Westrick, R. Yadav, M. Fluet, and U. A. Acar, “Disentanglement in nested-parallel programs,” *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.