Towards a Compiler Analysis for Parallel Algorithmic Skeletons

Towards a Compiler Analysis for Parallel Algorithmic Skeletons

Tobias J.K. Edler von Koch
tvkoch@acm.org
Qualcomm Innovation Center
USA

Stanislav Manilov
S.Z.Manilov@sms.ed.ac.uk
University of Edinburgh
United Kingdom

Christos Vasiladiotis
C.Vasiladiotis@sms.ed.ac.uk
University of Edinburgh
United Kingdom

Murray Cole
mic@inf.ed.ac.uk
University of Edinburgh
United Kingdom

Björn Franke
bfranke@inf.ed.ac.uk
University of Edinburgh
United Kingdom

Abstract

Parallelizing compilers aim to detect data-parallel loops in sequential programs, which – after suitable transformation – can be safely and profitably executed in parallel. However, in the traditional model safe parallelization requires provable absence of dependences. At the same time, several well-known parallel algorithmic skeletons cannot be easily expressed in a data dependence framework due to spurious depedences, which prevent parallel execution. In this paper we argue that *commutativity* is a more suitable concept supporting formal characterization of parallel algorithmic skeletons. We show that existing commutativity definitions cannot be easily adapted for practical use, and develop a new concept of commutativity based on liveness, which readily integrates with existing compiler analyses. This enables us to develop formal definitions of parallel algorithmic skeletons such as task farms, MapReduce and Divide&Conquer. We show that existing informal characterizations of various parallel algorithmic skeletons are captured by our abstract formalizations. In this way we provide the urgently needed formal characterization of widely used parallel constructs allowing their immediate use in novel parallelizing compilers.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords Commutativity, commutativity analysis, algorithmic skeletons, parallelism

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179513

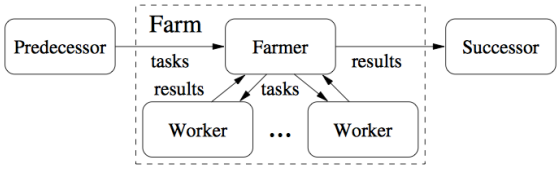
ACM Reference Format:
Tobias J.K. Edler von Koch, Stanislav Manilov, Christos Vasiladiotis, Murray Cole, and Björn Franke. 2018. Towards a Compiler Analysis for Parallel Algorithmic Skeletons. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179513

1 Introduction

Parallel algorithmic skeletons have been widely accepted as a fundamental abstraction supporting parallel programming ever since their original inception by Cole [12]. Such skeletons are often implemented either as libraries [19, 30], polymorphic higher-order functions [17], domain specific languages [48] or extensions to existing languages [18]. Their popular use and presence in e.g. Intel's Threading Building Blocks [44] or MapReduce [15] provides sufficient evidence that the patterns described by algorithmic skeletons exist and are generally considered useful.

In this paper we are interested in the role algorithmic skeletons can play in parallelism detection in sequential legacy code. Specifically, we are considering the requirements for automating the process of examining legacy code to determine – or at least strongly hint – that particular patterns are implicitly present. However, automating the detection of algorithmic skeletons critically depends on the availability of **formal skeleton definitions**, suitable for use in a compiler framework. In spite of all the interest in algorithmic skeletons, and while there appears to exist a set of patterns which are frequently cited, e.g. pipeline, divide & conquer, task farm, or stencil [22], there are **no agreed definitions** of what precisely constitutes each individual pattern. So far, algorithmic skeletons have been developed as an informal programming model and indeed, outside the pure functional programming domain [24], no efforts to formally capture skeletons have been made.

In this paper we develop the much needed abstract formalizations of parallel algorithmic skeletons. We observe that many algorithmic skeletons do not explicitly seek to avoid data dependences despite their parallel nature, which is at odds with traditional approaches where dependences inhibit



(a) Graphical characterization of a task farm.

"Conceptually, a farm consists of a farmer and several workers. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. The worker executes the task and delivers the result back to the farmer who propagates it to some successor process (which may be the same as the predecessor)."

(b) Verbal characterization of a task farm.

Figure 1. Both graphical and verbal characterizations of the *task farm* skeleton (both cited from [41]) are not suitable for compiler implementation as key concepts, e.g. *farmer*, *worker* and *task*, are not sufficiently defined.

parallelization. For example, task farms often comprise dynamically updated work lists to which new work items can be added by worker threads – these spurious dependences create insurmountable obstacles to traditional dependence analysis and parallelization and require manual intervention, e.g. [53].

The necessity to cope with dependences and yet to capture parallelism motivates the use of **commutativity** as a primitive building block. We show that existing notions of commutativity are powerful, but are not suitable for practical use in compilers. Hence we develop a new concept of liveness-based commutativity: the **key idea** is that if the order of execution of any two code regions can be exchanged without affecting live, i.e. later used, values, then these two regions are commutative and can be executed in parallel, subject to transactional accesses to shared variables.

Once we have introduced commutativity to reason about parallelism we use this concept to construct definitions of larger parallel algorithmic skeletons, capturing a selection of data-parallel, task-parallel and resolution skeletons.

Clearly, the lack of existing formalism to describe algorithmic skeletons makes it hard for us to validate our technique and skeleton definitions presented later in this paper. Here we take a collection of community-approved pattern instances from diverse pattern libraries, apply our definitions and demonstrate that most instances are classified in agreement with community-wisdom. We further examine those where we differ, and ascribe divergence to the cause. In the absence of gold-standard pattern definitions, this provides as strong evidence as is possible that we have captured the common essence of the diverse informal specifications in a single test. It also provides a foundation for the much needed abstract formalization of algorithmic skeletons.

1.1 Motivating Example

Consider the two characterizations of the *task farm* skeleton in Figure 1. Neither the graphical representation in Figure 1(a), nor the verbal description in Figure 1(b) are precise enough to characterize a task farm skeleton beyond a level of intuitive understanding. Central concepts of a task farm, e.g. *farmer*, *workers* and *tasks*, and their interaction remain *undefined*. Now compare this to the concrete code example in Figure 2. This code excerpt shows a *graph traversal* routine written in C++, which employs a worklist to iterate over all nodes of a graph, applies a function to each node and accumulates their return values. The final accumulated value is then returned to the caller. The body of the `while` loop in line 19 can be treated as a *task farm*, where the entire loop body represents a task. Obviously, there exist many dependences between iterations introduced by the (auxiliary) worklist (`queue`) and the marker array (`visited`), which would prohibit parallelization. However, if we look at the only **live-out** variable after this `while` loop, namely `result`, we will notice that this variable always holds the same value irrespective of the particular order in which the loop processes elements of the worklist. This means if we use a definition of commutativity, which only considers equality of **live-out** values, i.e values used further on in the program, we do not need to guarantee identical memory states for `queue` and `visited` for two implementation, one sequential and the other parallel. We can execute this loop in parallel, while actually violating spurious data dependences induced by `queue` as long as we provide transactional accesses to this shared variable.

In this paper we make the following **contributions**:

1. We introduce a new notion of *liveness-based commutativity*, which is the key building block for constructing our formal definition of algorithmic skeletons and is also more suitable for compiler integration than existing commutativity concepts.
2. For the first time, we provide *formal characterizations of parallel algorithmic skeletons* including task farms, divide&conquer, MapReduce and others.
3. We demonstrate that our formal characterizations *capture the essence* of widely agreed informal models of algorithmic parallel skeletons, indicating that our formal definitions agree with intuitive understanding of those parallel patterns held in the parallel programming community.

```
1  int Graph::Traverse(int s) {
2   int result = 0;
3
4   // Mark all the vertices as not visited
5   bool *visited = new bool[V];
6   for(int i = 0; i < V; i++)
7     visited[i] = false;
8
9   // Create a queue for graph traversal
10  list<int> queue;
11
12  // Mark current node as visited; enqueue it
13  visited[s] = true;
14  queue.push_back(s);
15
16  // 'i' - iterator over adjacent vertices
17  list<int>::iterator i;
18
19  while(!queue.empty())
20  {
21    // Dequeue a vertex from queue
22    s = queue.front();
23    queue.pop_front();
24
25    // Apply function f to s, accumulate values
26    result += f(s);
27
28    // Get all adjacent vertices of s.
29    // If an adjacent node hasn't been visited,
30    // then mark it as visited and enqueue it
31    for(i=adj[s].begin(); i!=adj[s].end(); ++i)
32    {
33      if(!visited[*i])
34      {
35        visited[*i] = true;
36        queue.push_back(*i);
37      }
38    }
39  }
40
41  return result;
42 }
```

Figure 2. A *graph traversal* algorithm written in C++. Conceptually, the `while` loop in line 19 represents a *task farm*. However, many other implementations of the same skeleton are possible. Tracking dependences introduced by the queue data structure is difficult, while the order of execution is not relevant for the calculation of `result`, the only live-out value leaving the code region.

2 Background

We discuss two notable contributions to commutativity, which have served as sources of inspiration for the work presented in this paper: *Separability-based commutativity* [45, 46] and *output-based commutativity* [4].

2.1 Separability-Based Commutativity

An early notion of commutativity views computation as composed of separable operations on objects [45, 46], where each operation has a receiver object and several parameters that are passed by value to the operation. When an operation executes it can access the parameters, invoke other operations or access the instance variables of the receiver. Restrictions apply to instance variable accesses, where the underlying model of computation require all accesses to happen indirectly by invoking operations that have the nested object as the receiver.

Commutativity analysis focuses on separable operations (i.e. it can be decomposed into an object section and an invocation section). The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. The motivation for separability is that the commutativity testing algorithm (which determines if operations commute) requires that each operation's accesses to the receiver execute atomically with respect to the operations that it invokes. Separability ensures that the actual computation obeys this constraint.

To test that method invocations commute, the compiler represents and reasons about the new values of the receiver's instance variables and the multiset of operations directly invoked when the two methods execute. The compiler represents the new values and multisets of invoked methods using symbolic expressions.

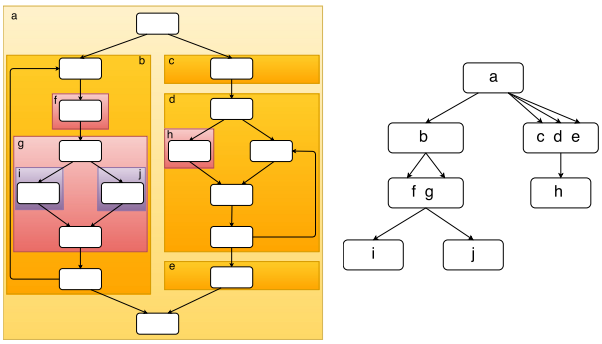
Unfortunately, this approach quickly reaches its limits for real-world applications. For example, applications are required to be coded as "clean" object-based programs. Yet, several major restrictions apply, which include: no virtual methods; no operator or method overloading; no multiple inheritance nor templates; no typedef, union, struct or enum types; global variables cannot be primitive data types – they must be class types; no use of pointers to members or static members; no casts between base types such as int, float and double that are used to represent numbers; no default arguments or methods with variable numbers of arguments; no operation accesses an instance variable of a nested object of the receiver or an instance variable declared in a class from which the receiver's class inherits.

Consequently, the evaluation of separability-based commutativity has been limited to highly sanitized, sequential versions of benchmarks derived from originally parallel code.

2.2 Output-Based Commutativity

In [4] an alternative notion of commutativity (and analysis) is proposed for individual functions, which considers the *output* of a function at the point of its use. In order to automatically detect commutative functions, a candidate function is symbolically executed in two different orders to create an abstract representation of the result of the two execution orders. This symbolic result is then used as input to all functions that could potentially read the results, and those functions are symbolically executed. If the outputs of these reader functions are identical, then the initial function is commutative.

Here, commutativity is a property of a *single function* w.r.t. multiple invocations of the same function. It is not the memory state created by the execution of this function, but in



(a) Control flow graph. (b) Program structure tree.

Figure 3. A control flow graph (Cfg) with hierarchically nested, canonical Sese regions, Figure (a), and its program structure tree (Pst), Figure (b), with sequentially composed canonical Sese regions grouped together (according to [27]). Such regions (e.g. f and g, but also c, d, and e, respectively) grouped together in the Pst are a source of task-level parallelism and can be executed concurrently if they are *commutative*, i.e. their execution order can be exchanged without changing any values *live-out* at their exit.

fact the state generated by the all of the potential consumers of the values generated by the function that determines the commutativity property.

For example, an `insert` function might enter several items in a linked list. Reversing the order of multiple invocations to this function creates a list containing the same items, but in a different order. If a subsequent consumer of this list produces the same result, irrespective of the order in which items are stored in the list, then the `insert` function is considered commutative.

Whilst this notion of commutativity has strength in handling e.g. unordered container data structures and their use, it is limited to the repeated, possibly commutative invocation of a single function. It is not so well suited for the characterization of skeletons, where there is a need to look beyond a series of calls to the same function.

3 Liveness-Based Commutativity

Our new definition of commutativity aims to combine the best aspects from existing commutativity concepts. From separability-based commutativity we incorporate the expressive power to reason about any two (or more) regions of code, whilst avoiding its practical limitations. From output-based commutativity we adopt the idea to only consider values "live-out" at the end of candidate regions, but avoid the "single function" limitation.

Unlike [4] we define commutativity as a *binary relation* over regions of code, which may or may not comprise function calls, rather than a property of an individual function.

By design we only require equality of observable state leaving commutative code regions, i.e. live-out and live-through sets and values, but we do not require exact matches of memory contents for intermediate, dead variables. The following definitions can handle recursive commutative functions (by allowing function calls in regions) and recursive data structures (through liveness).

Our definition of liveness-based commutativity is designed to avoid the need for sophisticated symbolic computation in a possible implementation of commutativity analysis, but enables its profitable use of it, should it be available. We also build directly on existing compiler concepts such a Single-Entry Single-Exit (Sese) regions and Program Structure Trees (Pst) [27] illustrated in Figure 3.

Definition 1. *Two SESE regions R_1 and R_2 are **commutative** iff*

1. a. *$R_1 \neq R_2$: R_1 and R_2 are canonical SESE regions both contained in the same maximal SESE region, and $\widehat{R}$ is the smallest such maximal SESE region, and R_1 is not contained in R_2 (and vice versa), and all variables and their values in $\text{live}_{out}[\widehat{R}]$ are the same for any execution order of R_1 and R_2; **or***
   b. *$R_1 = R_2$ ("R_1 is commutative with respect to itself"): R_1 is a maximal SESE region contained in another SESE region which contains a direct control flow path from the exit of R_1 back to its entry, and $\widehat{R}$ is the smallest such containing region, and all variables and their values in $\text{live}_{out}[\widehat{R}]$ are the same for all execution orders for (dynamically) repeated executions of R_1.*
2. *The values in $\text{live}_{through}[R_1]$ and $\text{live}_{through}[R_2]$ are unaffected by the execution order of R_1 and R_2.*

For example, consider regions f and g in Figure 3(a), which are contained in the same maximal Sese (f, g) in the Pst in Figure 3(b). If the live-out sets and their values of (f, g) are indistinguishable for any execution order of f and g, then f and g are commutative. Similarly, if different executions orders for c, d and e result in the same live-out variables and their values of (c, d, e), then c, d and e are commutative. Note that any Sese may contain one or more function calls, and that we are not asking for commutative regions to be immediate control flow predecessor/successor pairs.

For an example of part 1(b) of Definition 1, consider the maximal Sese (f, g) in Figure 3(a). There exists a control flow edge from the exit of (f, g) back to its entry, so that (f, g) can be executed multiple times. If any execution order of multiple instances of (f, g) produces the same set of live-out variables and their values are identical once the loop terminates, (f, g) is commutative with respect to itself.

3.1 Limitations

Generally, we assume that regions under scrutiny do not contain I/O statements or produce any other side-effects

not captured by liveness (exceptions, `volatile` memory accesses, etc.).

4 Skeleton Characterization

In the following paragraphs we are going to formally characterize a selection of widely used skeletons using the commutativity concept introduced in Definition 1. In turn, we will consider task-parallel, data-parallel and resolution skeletons, before we briefly discuss the relationship between concurrency based on commutativity (i.e. any execution order results in the same functionality) and parallelism (simultaneous execution).

4.1 Task-Parallel Skeletons

4.1.1 Primitive Task Parallelism

We start off with the simplest algorithmic skeleton, namely primitive *task parallelism*, where tasks represented as nodes (= SESE regions) appear in a sequential order in the (static) program source, but can be executed concurrently.

Definition 2. *Two SESE regions a and b are **task parallel** iff a and b are commutative.*

For example, consider Figure 3. Regions f and g are potential sources of task parallelism as f and g are sequentially ordered within a single SESE in the CFG, in Figure 3(a), and, thus, are part of the same maximal SESE region in the PST, in Figure 3(b). If f and g are additionally commutative, i.e. their execution order can be exchanged, then f and g are task parallel. Analogously, c, d, and e can be shown to be task parallel.

4.1.2 Function Task Parallelism

This form of parallelism is a special case of task parallelism, where two or more task-parallel regions (according to Definition 2) each contain a function call.

4.1.3 Task Farm Parallelism

A *task farm* is a dynamic task-parallel algorithmic skeleton, already introduced in the motivating example.

Definition 3. *A loop L is a **task farm** iff*

1. *L is a canonical SESE region,*
2. *the maximal SESE region R representing its loop body is commutative according to Definition 1.1(b),*
3. *it consumes a data $|\,\text{use}[L]| = \Omega(n)$ and produces live-out data $|\,\text{live}_{out}[L] \cap \text{def}[L]| = \Omega(1)$, where n is the number of loop iterations.*

We use Knuth-style asymptotic bounds to reason about the volume of data produced and consumed. Clause 3 can be read informally as "the loop has to consume at least n data items and has to produce at least one data item".

We allow arbitrary control flow in the loop body and any number of (non-statically determined) loop iterations. In fact, the loop body may generate more work items. Our commutativity based characterization avoids complex shape analysis [40] to identify internal work list data structures. Note that we do not require tasks (=operations performed in R) to be completely independent of each other.

4.2 Data-Parallel Skeletons

Data parallelism is the most widely used form of parallelism and refers to scenarios in which the same operation is performed concurrently on elements of a collection (usually arrays). Data parallel operations are partitioned so that multiple threads can operate on different segments concurrently. Traditional data parallelization requires that there are no data dependences between loop iterations.

4.2.1 "Conventional" Data-Parallel Loops/Reductions

We initially provide a commutativity characterization for ordinary data-parallel loops, also called *DO* loops [32], and parallel reductions.

Definition 4. *A simple, non-nested loop L is **data parallel** or a **reduction** iff*

1. *L is a canonical SESE region,*
2. *the maximal SESE region R representing its loop body is commutative according to Definition 1.1(b) and does not contain any cyclic control flow,*
3. *it consumes data $|\,\text{use}[L]| = \Theta(n)$ and $\text{use}[L] \subseteq \text{live}_{in}[L]$, and*
4. a. ***Data-parallel loop:** it produces data $|\,\text{live}_{out}[L] \cap \text{def}[L]| = \Theta(n)$, **or***
   b. ***Reduction:** it produces data $|\,\text{live}_{out}[L] \cap \text{def}[L]| = \Theta(1)$,*

where n is the number of loop iterations.

Note that we do not explicitly model cross-iteration dependences, but require commutativity of the loop body. We ask for the loop to consume a volume of data proportional to its iteration count. A data-parallel loop also produces a proportional volume of data, while a parallel reduction produces a constant volume of live-out data irrespective of the iteration count. The definition can be suitably adapted for nested loops. Note that any data-parallel loop is a task farm, but the inverse is not true.

4.2.2 Map Parallelism and Reduce Parallelism

Map is the first actual data-parallel algorithmic skeleton we define. We use a functional view of this skeleton and demand non-destructive mapping of inputs to outputs [15]. The *reduce* skeleton can be defined similarly.

Definition 5. *A simple, non-nested loop L is a **map skeleton** (or **reduce skeleton**, respectively) iff*

1. *L is a data parallel loop (or parallel reduction, respectively) according to Definition 4, and*

2. *it does not modify its input, i.e.* $(\text{use}[L] \cap \text{def}[L]) \nsubseteq \text{live}_{in}[L]$.

4.2.3 Map/Reduce Parallelism

Map and *Reduce* skeletons can be merged together to a combined *Map/Reduce* skeleton in following way:

Definition 6. *A sequence of a map skeleton M and a reduce skeleton R is a* ***map/reduce skeleton*** *MR iff* $\text{live}_{out}[M] = \text{live}_{in}[R]$ *and* $\text{use}[R] \subseteq \text{def}[M]$.

4.2.4 Fused Data-Parallel Skeletons

In practice, legacy C code will contain frequent use of fused data-parallel skeletons. This means that sequences of data-parallel loops over arrays may have been transformed into a single data-parallel loop using loop fusion. Although common in practice the individual characterizations of such fused skeletons are outwith the scope of this paper, but can be derived from the definitions of the basic skeletons.

4.3 Resolution Skeletons

4.3.1 Divide & Conquer

Definition 7. *A function f implements the* ***divide & conquer skeleton*** *iff the CFG representing the body of f contains a maximal SESE region R and there exist two distinct nodes $x_f, y_f \in R$, which*

1. *each comprise a recursive call to f,*
2. *x_f dominates y_f and y_f postdominates x_f,*
3. *$\text{use}[x_f] \neq \text{use}[y_f]$ (but possibly $\text{use}[x_f] \cap \text{use}[y_f] \neq \emptyset$), and*
4. *x_f and y_f are commutative.*

Informally, divide & conquer is represented by a function that contains two commutative recursive calls. Clause 3 further stipulates that these calls may share some of the same arguments (e.g. in the case of MergeSort, a pointer to the array being sorted) but must differ in at least one argument (e.g. the bounds of the array segments to be sorted).

4.4 Limitations

There exist algorithmic skeletons that cannot be characterized using commutativity. This is the case for skeletons which do not change the execution order of code regions, e.g. pipelines. Specialized techniques, such as [51, 52], may still be required to address these skeletons separately.

5 Validation

In this section we show that our formal skeleton definitions are in line with existing informal characterizations. Due to the absence of a formal reference for skeletons in general, we have to resort to a case-by-case discussion.

We approach validation in two ways: First, we show for examples of sequential code, which have been used elsewhere to illustrate skeleton based parallelization, that our characterization correctly classifies these examples. Second, we show for a selection of skeleton interfaces taken from existing libraries and frameworks that their implicit assumptions are equivalent to our formal definitions.

5.1 Validation against Sequential Code

Consider the code examples in Figure 4, where we show a number of (sequential) code examples, their CFGs and PSTs, and comment on the conditions according to the skeleton definitions provided earlier in this paper along with the resulting skeleton classifications.

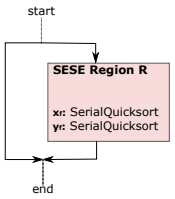
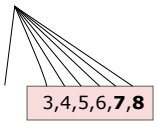
Quicksort This is a prime example of a "divide&conquer" algorithm, which is used elsewhere [25] to illustrate skeleton based parallelization. Two recursive calls are inside the innermost SESE. The recursive calls are commutative and operate on non-identical regions of data. As expected this meets our Divide & Conquer skeleton definition.

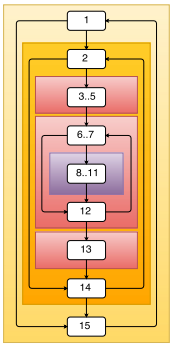
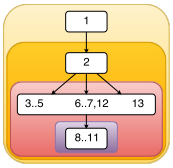
Mandelbrot This fractal generation algorithm is frequently cited in the literature as an example of a task farm skeleton, e.g. [13]. Whilst the innermost `while` loop is genuinely sequential (it contains loop-carried data dependences, but it is also not commutative), the two `for` loops each represent a task farm, i.e. their loop bodies are commutative and can be executed in any order without affecting the final outcome. In fact, both loops are data parallel (subject to correct handling of induction variables) and there is no data produced in one loop, which has been produced in an "earlier" iteration. Note that task farms in general can exhibit loop carried dependences, e.g. induced by worklists.

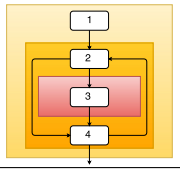
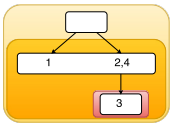
Simple Reduction Typically, reductions require special handling in traditional data dependence frameworks as they naturally exhibit a loop-carried dependence. In a commutativity framework a reduction skeleton is characterized as a commutative loop, which produces a live-out value. Following this intuition our definitions correctly classify this reduction example.

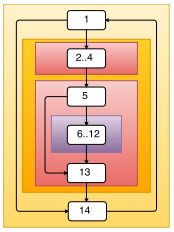
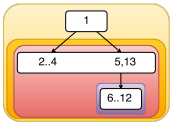
Complex Reduction This is an example of a fused skeleton, where several skeletons can be identified in the same piece of code. Trivially, the computations of `x1` and `x2` (in lines 2 and 3) and `t3` and `t4` (in lines 7 and 8), respectively, are commutative. Following our definitions each of these two pairs of statements constitute examples of primitive task parallelism (i.e. `x1` and `x2` can be computed in parallel, and so can `t3` and `t4`). The computation of `sx` and `sy` each represents a reduction, i.e. the iterations of the embracing `for` loops are commutative. According to our definitions the computation of `q[l]` is not a reduction as more than a single data item is produced. We would classify this as a task farm due to commutativity, though. It is conceivable

Code example	CFG	PST	Conditions	Skeleton(s)

Serial Quicksort example, e.g. Intel TBB [25]
`1 void SerialQuicksort(T* begin,T* end) {`
`2  if( end-begin>1 ) {`
`3  using namespace std;`
`4  T* mid = partition( begin+1, end,`
`5  bind2nd(less<T>(),*begin));`
`6  swap( *begin, mid[-1] );`
`7  SerialQuicksort( begin, mid-1 );`
`8  SerialQuicksort( mid, end );`
`9  }`
`10 }`
			• 7, 8 comprise recursive calls,
			• 7 dominates 8 and 8 postdominates 7,
			• use[7] ≠ use[8], and
			• 7 and 8 are commutative.
				Divide & Conquer (whole function & lines 7/8)
				Function Task Parallelism (lines 7/8 only)

Serial Mandelbrot example [7], e.g. eSkel [13]
`1 for(cy=yMin,y=0;cy<yMax;cy+=dxy,y++) {`
`2  for(cx=xMin,x=0;cx<xMax;cx+=dxy,x++) {`
`3  zx = 0.0;`
`4  zy = 0.0;`
`5  n = 0;`
`6  while ((zx*zx + zy*zy < 4.0) &&`
`7  (n != UCHAR_MAX)) {`
`8  new_zx = zx*zx - zy*zy + cx;`
`9  zy = 2.0*zx*zy + cy;`
`10  zx = new_zx;`
`11  n++;`
`12  }`
`13  image[x][y] = n;`
`14  }`
`15 }`
			• loops 1/15, 2/14 consume N/M items,
			• loops 1/15, 2/15 produce N/M items,
			• all data is live-in, and
			• loop bodies 1/15, 2/14 are commutative.
				Task Farm/Data Parallelism (loop spanning 1 . . . 15)
				Task Farm/Data Parallelism (loop spanning 2 . . . 14)
				Note: Innermost `while` loop is not commutative

Simple reduction example
`1 x = 0.0;`
`2 for(i = 0; i < N, i) {`
`3  x += a[i];`
`4 }`
			• consumes N live-in items,
			• 3 commutative, and
			• x is produced.
				Reduction (loop spanning 2 . . . 4)

Complex reduction example (NAS EP, Gaussian dev.)
`1 for (i = 0; i < NK; i++) {`
`2  x1 = 2.0 * x[2*i] - 1.0;`
`3  x2 = 2.0 * x[2*i+1] - 1.0;`
`4  t1 = x1 * x1 + x2 * x2;`
`5  if (t1 <= 1.0) {`
`6  t2  = sqrt(-2.0 * log(t1) / t1);`
`7  t3  = (x1 * t2);`
`8  t4  = (x2 * t2);`
`9  l  = MAX(fabs(t3), fabs(t4));`
`10  q[l] = q[l] + 1.0;`
`11  sx  = sx + t3;`
`12  sy  = sy + t4;`
`13  }`
`14 }`
			• data consumed in 2/3 is live-in,
			• 2/3 commutative,
			• 10/11/12 commutative,
			• produces q, sx, sy,
			• loop body 2 . . . 13 is commutative.
				Primitive Task Parallelism (lines 2/3, 7/8, 10/11/12, respectively)
				Reduction (loop and lines 11,12)
				Task Farm (loop and line 10)

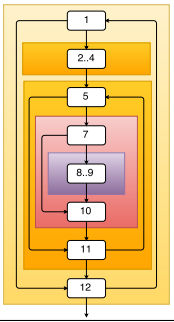
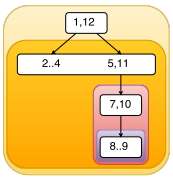
Graph traversal (from Figure 2)
`1  while(!queue.empty()) {`
`2  s = queue.front();`
`3  queue.pop_front();`
`4  result += f(s);`
`5  for(i=adj[s].begin();i!=adj[s].end();`
`6  ++i) {`
`7  if(!visited[*i]) {`
`8  visited[*i] = true;`
`9  queue.push_back(*i);`
`10  }`
`11  }`
`12 }`
			• loop 1 . . . 12 consumes N items,
			• produces `result`,
			• loop body 2 . . . 11 is commutative.
				Task Farm (loop spanning 1 . . . 12)
				Note: This is not a "traditional" reduction, as some data consumed is only produced throughout loop execution.

Figure 4. Skeleton characterizations for given sequential code examples. Note that some examples may match more than a single skeleton, and skeletons can be fused and/or nested.

to introduce another skeleton, e.g. *histogram*, to deal with the computation of `q[l]` more appropriately.

Graph Traversal This graph traversal algorithm is not traditionally parallelizable due to the cross-iteration dependence induced by the worklist queue. However, the outer `while` loop is commutative and, hence, we can classify this loop as a task farm producing `result`. This means that any execution order will be functionally equivalent and a parallel implementation, which suitably synchronizes accesses to `queue` and `visited`, is possible. The classification of the inner `for` loop is trickier: `queue` is live-out and whether we can classify this loop as a task farm depends on our notion of equality for container data structures. If we regard two queues containing the same elements, possible in a different order, as equal, then the loop in line 5 could be treated as a task farm. However, without further assumptions we are safe to adopt a more conservative approach as require "bit-wise" equality for `queue` and treat this loop as non-commutative. The output-based notion of commutativity analysis from [4] would be able to reason about the commutativity of these repeated invocations of `push_back` with respect to the later use of the queue, though.

5.2 Validation against Library Patterns

For a number of skeleton supported by various skeleton frameworks we now demonstrate that our formal characterizations describe the same concept.

Task Farm For example, consider the task farm example in Figure 5, which implements a matrix multiplication algorithm using the FastFlow [3] skeleton framework. In this example, the farm skeleton orchestrates the parallel execution of the same code (within a worker object) on independent items of the input stream. Within the *Worker* class the `svc` function (a SESE region) performs the actual computation. During each invocation of this function data is consumed from the *task* descriptor and the arrays `A` and `B`, whilst a single element of the resulting matrix is produced in array `C`. By contract, invocations to task workers (the `svc` function) are commutative. In fact, these inherent properties match our task farm characterization from Definition 3.

MapReduce Now consider Figure 6. The Hadoop code snippet shows a simple MapReduce skeleton. A *Mapper* simply emits "1" for each term it processes, while *Reducer* goes through the lists of ones and sums them up. *Mapper* implements a method *map*, which consumes terms *t*, and emits "1". *map* does not modify any of its input, each iteration of the contained loop consumes a live-in data element and produces a live-out result.

```
1  // FastFlow accelerated code
2  #define N 1024
3  long A[N][N],B[N][N],C[N][N];
4  int main() {
5  // < init A,B,C>
6
7  ff ::ff_farm<> farm(true /* accel */);
8  std :: vector<ff:: ff node âĹŮ> w;
9  for(int i=0;i<PAR_DEGREE;++i)
10    w.push_back(new Worker);
11    farm.add_workers(w);
12    farm.run_then_freeze();
13
14    for (int i=0;i<N;i++) {
15       for(int j=0;j<N;++j) {
16          task t * task = new task_t(i,j);
17          farm.offload(task);
18       }
19    }
20    farm.offload((void *)ff ::FF_EOS);
21    farm.wait(); // Here join
22 }
23
24 // Includes
25 struct task_t {
26    task_t(int i ,int j) : i(i), j(j) {}
27    int i ; int j ;
28 };
29
30 class Worker: public ff::ff_node {
31 public: // Offload target service
32    void * svc(void *task) {
33       task_t * t = (task_t *)task;
34       int C=0;
35
36       for(int k=0;k<N;++k)
37          C += A[t->i][k]*B[k][t->j];
38       C[t->i][t->j] = C;
39       delete t;
40       return GO_ON;
41    }
42 };
```

Figure 5. Matrix multiplication using a task farm implementation based on FastFlow, taken from [2].

This exactly matches our *Map* skeleton characterization in Definition 5. Following a similar argument, *Reducer* meets our provided *Reduce* definition. Together, the sequence of *Mapper* and *Reducer* meets Definition 6 for a MapReduce skeleton.

Divide & Conquer Next, consider the Skandium [33] Divide & Conquer skeleton in Figure 7, which uses recursive data decomposition to recursively apply a function until a condition is reached. The specific operation of this skeleton can be broken down as follows: Given an initial input, the *condition* is evaluated. This can either result into a split of the input using the *split* or in the passing of data to the next stage of computation (e.g. a sub-skeleton). This process is recursively applied until the full traversal of the recursion tree. The next stage involves the *merging* of the partial results from each level. Ultimately after the final result

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

Figure 6. A basic HADOOP MapReduce skeleton.

```
Condition <P> condition = ...;
Split <P,P> split = ...;
Skeleton <P,R> nested = ...;
Merge <R,R> merge = ...;
Skeleton <P,R> dac =
    new DaC <P,R>(condition,split,nested,merge);
```

Figure 7. The SKANDIUM [33] Divide & Conquer skeleton.

is constructed, it is returned to the user. According to Definition 7 Divide& Conquer is a special case of the more generic Function Task Parallelism skeleton, where a function is invoked (two or more times) recursively, each with (partially) disjoint parameters. This means, that the recursive calls operate on split input data. Our definition does not explicitly refer to the termination condition, but it is safe to assume that a correct[1], sequential, recursive function has a valid termination condition. Similarly, our Divide & Conquer definition does not specify the behaviour of the *merge* step – this can even be absent as in the *quicksort* algorithm. The SKANDIUM DaC skeleton does not provide any guarantees on the execution order of the skeleton muscle (`nested`); this is an internal implementation detail. In turn, this means that by contract the operations performed by `nested` are commutative, which aligns with our Definition 7.

5.3 Validation against Real-World Applications

Next we validate our skeleton characterizations against real-world applications from the BioPerf benchmark suite [1, 8]. We have chosen applications from this benchmark suite, which have been independently parallelized and comprise task farm parallelism. In table 1 we compare our task farm skeleton characterizations applied to the *sequential* program versions to the task farms exploited in manual parallelization.

In all cases where manual parallelization has exploited task farm sketons our definition also successfully captures the same skeletons in the sequential code. This confirms that our definition of a task farm is in line with what the parallel programming community considers to be a task farm. One noteworthy exception is `ssearch34_34` from the FASTA benchmark, where a dynamic programming parallelization approach is taken in [16], but we identify this as a task farm. This is because dynamic programming can be implemented using a task farm like mechanism and this is exactly what our definition captures when applied to this benchmark. Future work would will refine the class of task farm skeletons and more precisely distinguish dynamic programming from other task farm skeletons.

We have also applied conventional parallelizing compilers including Intel ICC and LLVM/Polly to the task farms in these sequential benchmarks, but as expected they fail to discover any parallelism. In contrast, we expect a compiler enabled by commutativity analysis according to the definition 1 to be able to extract and exploit task farms and other skeletons characterized in Section 4. This, however, is beyond the scope of this paper.

[1]We are only considering programs, which in their sequential form satisfy the usual notions of correctness.

6 Related Work

Commutativity as a distinct and more general concept than dependence was explored as far back as [9]. In [29] a technique is presented to verify commutativity conditions, which are logical formulae that characterize when operations on a linked data structure commute. ALTER [53] is a framework for loop parallelization using manually placed commutativity annotations. Code annotations for commutative functions are also proposed in parallelization frameworks by [10], as well as in GALOIS [31] and PARALAX [54]. A generalized semantic commutativity based programming extension, called *Commutative Set* (COMMSET), and its associated compiler technology are presented in [42]. COMMSET supports pipeline and data parallelism, but not task parallelism. In [11] commutativity is identified as a key enabler for scalability in OS services. The informal concept of *algorithmic skeletons* was introduced by [12] in the 1980s to abstract commonly-used patterns of parallel computation, communication, and interaction. The idea has been widely adopted in the parallel programming community, for example, in the shape of Intel's *Threading Building Blocks* (TBB) [44], Google's MAPREDUCE [15], and in a large number of dedicated skeleton frameworks, e.g. STAPL, eSKEL, or FASTFLOW to just name a few. A detailed survey is given in [22]. In [37] and [36] catalogues of architecture and communication patterns for parallel applications are presented. Researchers from different communities independently use skeleton-like abstractions [31] and often refer to algorithmic skeletons using different terminology, e.g. *tao of parallelism* [39] for a class of irregular graph processing skeletons or *dwarfs* [6] for representatives for classes of algorithms from different domains. A number of specialized techniques have been developed to detect task parallelism or individual skeleton types, e.g. Divide&Conquer [20, 23, 47] or pipelines [14, 51, 52]. Detection of algorithmic skeletons differs significantly from pattern-driven automatic

Table 1. Comparison of task farms captured by our formal characterization in the original sequential versions of the BioPerf benchmarks to those identified and published by other researchers as part of manual efforts.

Package	Executable	Original Sequential Code			Manually Parallelized Reference	
		Captured Skeletons	Location	Cov. (in %)	Exploited Skeletons	Source
CLUSTALW	clustalw	Task Farm	`pairalign.c` `pairalign:164-242`	≈ 70%	Task Farm	[34]
FASTA	fasta34_t	Task Farm	`com_lib.c` `main:1097-1233`	> 95%	Task Farm	[38]
	ssearch34_34	Task Farm	`com_lib.c` `main:1097-1233`	> 95%	Dynamic Programming	[16]
HMMER	hmmsearch	Task Farm	`core_algorithms.c` `P7Viterbi:589-620`	> 90%	Task Farm	[26, 43]
	hmmpfam	Task Farm	`core_algorithms.c` `P7Viterbi:589-620`	> 90%	Task Farm	[26, 43]

parallelization, e.g. [5, 21, 28, 35, 49, 50], which aim to recognize a particular algorithmic idiom.

7 Summary, Conclusion and Future Work

In this paper we have developed a formal characterization of a selection of popular algorithmic skeletons based on a novel notion of commutativity. This in turn is based on well-understood concepts in compiler theory: liveness and Sese regions. We have shown that many algorithmic skeletons have a simple and elegant commutativity characterization. Lacking prior formal definitions we validate our skeleton characterizations in two different ways: Initially, we apply our definitions to sequential code examples used elsewhere with the purpose of demonstrating the power of skeleton based parallelization. We show that our definitions result in the correct, i.e. widely agreed, skeleton classification for the given examples. In a second stage we review skeleton interfaces provided in popular skeleton frameworks and demonstrate again that our formal definitions meet the implicit assumptions and properties of those frameworks. We believe that our novel characterization of skeletons based on commutativity and liveness overcomes limitations of earlier dependence-based approaches and represents a promising new direction for the detection of structured parallelism in legacy applications. Future work will focus on (a) characterization of further skeletons, e.g. dynamic programming, stencils, etc., (b) development of advanced commutativity analyses integrating static, dynamic and probabilistic techniques, and (c) automatic mapping onto optimized skeleton frameworks, e.g. Tbb, Hadoop or FastFlow.

References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 2–9, March 2005.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating sequential programs using FastFlow and self-offloading. *CoRR*, abs/1002.4668, 2010.

[3] M. Aldinucci, M. Torquati, and M. Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, Sept. 2009.

[4] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 241–252, New York, NY, USA, 2009. ACM.

[5] M. Arenaz, J. Touriño, and R. Doallo. XARK: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst.*, 30(6):32:1–32:56, Oct. 2008.

[6] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[7] M. Ashley. PHYS2020 – Computational Physics, based on the C programming language. http://www.phys.unsw.edu.au/ mcba/phys2020, 2004.

[8] D. A. Bader and V. Sachdeva. An open benchmark suite for evaluating computer architecture on bioinformatics and life science applications. Technical Report GT-CSE-06-08, Georgia Institute of Technology, 2007.

[9] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966.

[10] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[11] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, Jan. 2015.

[12] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[13] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004.

[14] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel. Automatic extraction of pipeline parallelism for embedded heterogeneous multicore platforms. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 4:1–4:10, Piscataway, NJ, USA, 2013. IEEE Press.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[16] B. Deepa and V. Nagaveni. Parallel smith-waterman algorithm for gene sequencing. *International Journal on Recent and Innovation Trends in Computing and Communication (IJRITCC)*, 3(5):3237 – 3240, 2015.

[17] M. Dieterle, T. Horstmeyer, J. Berthold, and R. Loogen. Iterating skeletons - structured parallelism by composition. In *IFL*, 2012.

[18] A. Dorta, P. López, and F. de Sande. Basic skeletons in 11C. *Parallel Comput.*, 32(7):491–506, Sept. 2006.

[19] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. QUAFF: Efficient C++ design for parallel skeletons. *Parallel Comput.*, 32(7):604–615, Sept. 2006.

[20] B. Freisleben and T. Kielmann. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computers and Artificial Intelligence*, 14:579–596, 1995.

[21] P. Ginsbach and M. F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[22] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[23] M. Gupta, S. Mikhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 139–148, Oct 1999.

[24] K. Hammond and G. Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, UK, 2000.

[25] Intel. Intel Threading Building Blocks documentation – divide and conquer. https://software.intel.com/en-us/node/506118, 2016.

[26] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa. An efficient parallel implementation of the hidden markov methods for genomic sequence-search on a massively parallel system. *IEEE Transactions on Parallel and Distributed Systems*, 19(1):15–23, Jan 2008.

[27] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[28] C. W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, Aug 1996.

[29] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 528–541, New York, NY, USA, 2011. ACM.

[30] H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a C++ skeleton library. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 122–130, New York, NY, USA, 2002. ACM.

[31] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[32] L. Lamport. The parallel execution of DO loops. *Commun. ACM*, 17(2):83–93, Feb. 1974.

[33] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 289–296, Feb 2010.

[34] K.-B. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585, 2003.

[35] B. D. Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the 4th International Workshop on Program Comprehension*, WPC '96, pages 164–173, Washington, DC, USA, 1996. IEEE Computer Society.

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

[37] J. Ortega-Arjona. *Patterns for parallel software design*. Wiley, 2010.

[38] W. R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635 – 650, 1991.

[39] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[40] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer Berlin Heidelberg, 1994.

[41] M. Poldner and H. Kuchen. On implementing the farm skeleton. In *Proceedings of the 3rd International Workshop HLPP 2005*, 2005.

[42] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. ACM.

[43] S. Quirem, F. Ahmed, and B. K. Lee. CUDA acceleration of P7Viterbi algorithm in HMMER 3.0. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–2, Nov 2011.

[44] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[45] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 54–67, New York, NY, USA, 1996. ACM.

[46] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[47] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. ACM.

[48] T. Saidani, J. Falcou, C. Tadonki, L. Lacassagne, and D. Etiemble. Algorithmic skeletons within an embedded domain specific language for the CELL processor. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT'09, pages 67–76, Sept 2009.

[49] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.

[50] A. Shafiee Sarvestani, E. Hansson, and C. Kessler. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. Parallel Program.*, 41(6):806–824, Dec. 2013.

[51] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[52] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[53] A. Udupa, K. Rajan, and W. Thies. ALTER: Exploiting breakable dependences for parallelization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 480–491, New York, NY, USA, 2011. ACM.

[54] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.