Towards a Compiler Analysis for Parallel Algorithmic Skeletons

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3178372.3179513

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 27th International Conference on Compiler Construction (CC2018)
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 dependences, 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
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.
int Graph::Traverse(int s) {
    int result = 0;
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    // Create a queue for graph traversal
    list<int> queue;
    // Mark current node as visited; enqueue it
    visited[s] = true;
    queue.push_back(s);
    // 'i' = iterator over adjacent vertices
    list<int>::iterator i;
    while (!queue.empty())
    {
        // Dequeue a vertex from queue
        s = queue.front();
        queue.pop_front();
        // Apply function f to s, accumulate values
        result += f(s);
        // Get all adjacent vertices of s.
        // If an adjacent node hasn’t been visited,
        // then mark it as visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
    return result;
}

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
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. \( R_1 \neq R_2 \): \( R_1 \) and \( R_2 \) are canonical Sese regions both contained in the same maximal Sese region, and \( R \) 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 live-out[\( R \)] are the same for any execution order of \( R_1 \) and \( R_2 \); or
2. \( R_1 = R_2 \) ("\( R_1 \) 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 \( R \) is the smallest such containing region, and all variables and their values in live-out[\( R \)] are the same for all execution orders for (dynamically) repeated executions of \( R_1 \).

2. The values in live-through[\( R_1 \)] and 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.
Towards a Compiler Analysis for Parallel Algorithmic Skeletons

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 Crg, 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 $\text{live}_{\text{out}}[L] \cap \text{def}[L] = \Omega(1)$, 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 $\text{use}[L] \subseteq \text{live}_{\text{in}}[L]$, and
4. a. Data-parallel loop: it produces data $\text{live}_{\text{out}}[L] \cap \text{def}[L] = \Theta(1)$,
   b. Reduction: it produces data $\text{live}_{\text{out}}[L] \cap \text{def}[L] = \Theta(1)$, 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. \( \text{use}[L] \cap \text{def}[L] \subsetneq \text{live}_{in}[L] \).

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 skeletal \( MR \) iff \( \text{live}_{out}[M] = \text{live}_{in}[R] \) and \( \text{use}[R] \subsetneq \text{def}[M] \).

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 CSE representing the body of \( f \) contains a maximal SSE 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 SSE. 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 \( x_1 \) and \( x_2 \) in lines 2 and 3 and \( t_3 \) and \( t_4 \) (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. \( x_1 \) and \( x_2 \) can be computed in parallel, and so can \( t_3 \) and \( t_4 \)). The computation of \( s_x \) and \( s_y \) each represents a reduction, i.e. the iterations of the embracing for loops are commutative. According to our definitions the computation of \( q[1] \) 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
Serial Quicksort example, e.g. Intel TBB [25]

```c
void SerialQuicksort(T* begin, T* end) {
    if (end - begin > 1) {
        using namespace std;
        T* mid = partition(begin + 1, end,
        bind2nd(less<T>(), *begin));
        swap(*begin, mid[-1]);
        SerialQuicksort(begin, mid - 1);
        SerialQuicksort(mid, end);
    }
}
```

Serial Mandelbrot example [7], e.g. eSkel [13]

```c
for (cy = yMin, y = 0; cy < yMax; cy += dxy, y++) {
    for (cx = xMin, x = 0; cx < xMax; cx += dxy, x++) {
        zx = 0.0;
        zy = 0.0;
        n = 0;
        while ((zx*zx + zy*zy < 4.0) &&
                (n != UCHAR_MAX)) {
            new_zx = zx*zx - zy*zy + cx;
            zy = 2.0*zx*zy + cy;
            zx = new_zx;
            n++;
        }
        image[x][y] = n;
    }
}
```

Simple reduction example

```c
x = 0.0;
for (i = 0; i < N, i) {
    x += a[i];
}
```

Complex reduction example (NAS EP, Gaussian dev.)

```c
for (i = 0; i < NK; i++) {
    x1 = 2.0 * x[2*i] - 1.0;
    x2 = 2.0 * x[2*i + 1] - 1.0;
    t1 = x1 * x1 + x2 * x2;
    if (t1 <= 1.0) {
        t2 = sqrt(-2.0 * log(t1) / t1);
        t3 = (x1 * t2);
        t4 = (x2 * t2);
        q[i] = MAX(fabs(t3), fabs(t4));
        sx = sx + t3;
        sy = sy + t4;
    }
}
```

Graph traversal (from Figure 2)

```c
while (!queue.empty()) {
    s = queue.front();
    queue.pop_front();
    result += f(s);
    for (i=adj[s].begin(); i!=adj[s].end(); ++i) {
        if (!visited[*i]) {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}
```

**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 contrast, 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. 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
1. class Mapper
2.   method Map(docid id, doc d)
3.     for all term t in doc d do
4.       Emit(term t, count 1)
5.   class Reducer
6.   method Reduce(term t, counts [c1, c2, ...])
7.     sum = 0
8.     for all count c in [c1, c2, ...] do
9.       sum = sum + c
10.      Emit(term t, count sum)

Figure 6. A basic HADOOP MapReduce skeleton.

1. Condition <P> condition = ...;
2. Split <P,P> split = ...;
3. Skeleton <P,R> nested = ...;
4. Merge <R,R> merge = ...;
5. Skeleton <P,R> dac =
6.   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 skeletons 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 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.

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. ALTEx [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

\(^1\)We are only considering programs, which in their sequential form satisfy the usual notions of correctness.
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