A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers

Horacio González-Vélez1* and Mario Leyton2†

1 School of Computing/ IDEAS Research Institute, Robert Gordon University, UK
2 NIC Labs, Universidad de Chile, Chile

SUMMARY

Structured parallel programs ought to be conceived as two separate and complementary entities: computation, which expresses the calculations in a procedural manner, and coordination, which abstracts the interaction and communication. By abstracting commonly-used patterns of parallel computation, communication, and interaction, algorithmic skeletons enable programmers to code algorithms without specifying platform-dependent primitives. This article presents a literature review on algorithmic skeleton frameworks (ASKF), parallel software development environments furnishing a collection of parameterisable algorithmic skeletons, where the control flow, nesting, resource monitoring, and portability of the resulting parallel program is dictated by the ASKSF as opposed to the programmer. Consequently, the ASKSF can be positioned as high-level structured parallel programming enablers, as their systematic utilisation permits the abstract description of programs and fosters portability by focusing on the description of the algorithmic structure rather than on its detailed implementation. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Algorithmic Skeletons; Structured Parallelism; Parallel Constructs; Concurrent Programming Structures; Software Patterns; Parallel Computing

1. INTRODUCTION

Parallel programming aims to capitalise on the simultaneous execution of different program sections, in order to improve the overall performance of such program, and, eventually, that of the whole system. Despite major breakthroughs, parallel programming is still a highly demanding activity widely acknowledged to be more difficult than its sequential counterpart, where the use of efficient parallel programming models has long been coveted. These programming models must necessarily be performance-oriented, are expected to be defined in a scalable structured fashion, and provide guidance on the execution of their jobs in order to assist in the deployment of heterogeneous resources and policies.

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [1, 2]. While computation constructs manage logic, arithmetic, and control flow operations, communication and interaction primitives coordinate inter- and intra-process data exchange, process creation, and synchronisation. Skeletons provide top-down design, composition, and control inheritance throughout the program structure. Structured parallel programs are expressed by interweaving parameterised skeletons analogously to the way in which structured sequential programs are constructed.

*Correspondence to: Horacio González-Vélez, School of Computing, Robert Gordon University, St. Andrew Street, Aberdeen AB25 1HG, United Kingdom. E-mail: h.gonzalez-velez@rgu.ac.uk
†E-mail: mario@leyton.cl
Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using speauth.cls [Version: 2010/05/13 v3.00]
Figure 1. The typical structure of an algorithmic skeleton framework.

Algorithmic skeleton frameworks (ASKF) furnish a set of algorithmic skeletons with generic parallel functionality, that are parameterised by the programmer to generate a specific parallel program. Analogous to common software libraries, skeletons are typically accessible through language syntactic extensions or well-defined application programming interfaces. However, the control flow, nesting, resource monitoring, and portability of the resulting parallel program is dictated by the ASKF as opposed to the programmer. Figure 1 shows the functioning of ASKF, where a programmer transforms a parallel algorithm into a program by selecting an algorithmic skeleton, or a nesting of them, through an API from a given skeleton repository in the form of a library. Then the system links the library and the multiprocessing support to produce a structured parallel program.

Known as structured parallelism, the systematic utilisation of ASKF provides a high-level parallel programming methodology which allows an abstract description of programs. ASKF foster portability by focusing on the description of the algorithmic structure rather than on its detailed implementation. Structured parallelism provides a clear and consistent behaviour across platforms, with the underlying structure depending on the particular implementation of a given ASKF. The skeleton behaviour refers to the outcome sought by the application programmer, and the skeleton structure concerns the resource to functionality correspondence established at the infrastructure level.

Therefore, by decoupling the behaviour from the structure of a parallel program, structured parallelism benefits entirely from any performance improvements in the system infrastructure, while preserving the program properties. This behaviour-structure decoupling has enabled ASKF to be seamlessly deployed on different dedicated and non-dedicated architectures including symmetric multiprocessing, massively parallel processing, and heterogeneous distributed systems such as clusters and grids.

This article is structured as follows. Section 2 presents an overview of the fundamental concepts of shared-memory and message-passing programming, which have historically motivated the use of ASKF. Section 3 introduces algorithmic skeletons as a viable parallel programming model. Section 4 provides a classification of ASKF. Section 5 examines in detail a series of representative ASKF, followed by a critical comparison of the representative ASKF in Section 6. Section 7 discusses approaches related to algorithmic skeletons. Finally, Section 8 examines existing directions in the development of ASKF and supplies some concluding remarks.
2. BACKGROUND

Hardware-independent parallel programming abstractions have long been assayed. Monitors, critical regions, and shared-memory models, in general, supply parallelism, concurrency and synchronisation through the deployment of data structures partaken by all processes. Thus their application space is mostly confined to specific intra-process regions. Orchestrated efforts have led to the introduction of different shared-memory programming models, such as OpenMP [3], and thread-only programming like POSIX threads [4], Java threads [5] and C# threads [6]. OpenMP—a set of compiler directives and library routines—allows the programmer to insert explicit parallelism directives into monolithic blocks of iterative code, while thread-based programming uses lightweight processes to achieve low-level independent control flow.

Conversely, message-passing programming provides synchronisation through send-receive pairing between specific processes and concurrency through the explicit initialisation of participating processes, and therefore allows the programmer to control inter-process interaction. Standard message passing libraries, such as the Parallel Virtual Machine (PVM) [7] and the Message Passing Interface (MPI) [8, 9], allow heterogeneous collections of interconnected systems, with potentially different architectures and operating systems, to act as a single computing unit.

Although there is not a definitive answer in the subject, experimental studies [10, 11, 12] have consistently demonstrated that shared-memory is easy to program but lacks scalability and coarse-grain scope, while message-passing is portable, tunable, and scalable but error prone. Furthermore, the low-level communication primitives in message passing have long been compared to the assembly language crudeness and even equated to the usage of the infamous ‘go-to’ statement [13].

Above all, a parallel program ought to be conceived as two separate and complementary concerns: computation, which expresses the calculations in a procedural manner, and coordination, which abstracts the interaction and communication. In principle, the coordination and the computation should be orthogonal and generic, so that a coordination style can be applied to any parallel program, coarse- or fine-grained. Nonetheless, conventional parallel applications impose a bottom-up model of parallel programming, where computation and coordination are not necessarily separated, and communications and synchronisation primitives are typically interwoven with calculations.

Although the MPI standard has been augmented with collective operations which, in essence, provide an upper layer of computation and communication [14, 15], their implementation and its associated performance is dependent on the physical topology of the system, number of processes involved, message sizes, location of the root node, and actual algorithm employed [16, 17]. Alternatively, coordination has been directly enforced through syntactic structures in ad-hoc languages such as Linda [18], Opus [19], and Orca [20]. Linda uses a generative model for process creation and communication, where processes interact with each other using primitive data structures, known as tuples. The tuples form a tuple space, which is designed to be independent of the host language. Opus deals with the task parallelism using shared data abstractions, which execute autonomously on their own resources. Similarly, Orca data structures have a certain number of pre-defined operations that can be executed by processes.

Shared-memory, message-passing, or coordination languages do not provide, on their own, a standard and portable way of decoupling the structure from the behaviour in a program. Hence, traditional parallel programs often interleave computation and coordination for a certain algorithmic solution, greatly reducing the possibility of using the program structure as a steering criterion for adaptivity, performance, and programmability. Structured parallelism can, arguably, shed light on this matter.

3. ALGORITHMIC SKELETONS: A BRIEF OVERVIEW

Cole pioneered the field with the definition of an algorithmic skeleton as a “specialised higher-order functions from which one must be selected as the outermost purpose in the program”, and the introduction of four initial skeletons: divide and conquer, iterative combination, cluster, and task queue [1, 2]. His work described a software engineering approach to high-level parallel
Table I. A taxonomy for the algorithmic skeleton constructs based on their functionality.

<table>
<thead>
<tr>
<th>Skeleton</th>
<th>Scope</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-parallel</td>
<td>Data structures</td>
<td>map, fork, scan, reduce, ...</td>
</tr>
<tr>
<td>Task-parallel</td>
<td>Tasks</td>
<td>sequential, farm, pipe, if, for, while, loop, ...</td>
</tr>
<tr>
<td>Resolution</td>
<td>Family of problems</td>
<td>divide-and-conquer, branch-and-bound, dynamic programming, ...</td>
</tr>
</tbody>
</table>

programming using a skeletal (virtual) machine rather than the deployment of a tool or language on a certain architecture. He later surveyed the field, describing formally the functionality of some data- and task-parallel skeletons, using functional programming notation [21].

Loosely coincident to Cole’s initial formulation, fundamental work on high-level parallel constructs was being developed elsewhere, including the identification of nine computational models for a one-dimensional processor array [22], including the pipeline and the task queue; the use of higher-level portable monitors [23], including an ask-for monitor, essentially, an early instantiation of a task farm; the initial description of a transputer processor farm [24], and the use of hardware-based scan primitives for data-parallel algorithms [25]; the foundations of the Bird-Meertens formalism [26], and the subsequent development of second-order functions based on this formalism, to functionally derive parallel algorithms [27]; a functional specification for pipeline and divide-and-conquer algorithms [28]; the generic parallel implementations of branch-and-bound and backtrack, precursors of resolution skeletons [29]; and, the automated derivation of programs from specifications [30].

Algorithmic skeletons essentially abstract commonly-used patterns of parallel computation, communication, and interaction. Skeletal parallel programs can be expressed by interweaving parameterised skeletons using descending composition and control inheritance throughout the program structure, analogously to the way in which sequential structured programs are constructed [31]. This high-level parallel programming technique, known as structured parallelism, enables the composition of skeletons for the development of programs where the control is inherited through the structure, and the programmer adheres to top-down design and construction. Facilitated through ASK F, structured parallelism provides a clear and consistent behaviour across platforms, while their structure depends on the particular implementation.

Since skeletons enable programmers to code algorithms without specifying the machine-dependent computation and coordination primitives, they have been positioned as coordination enablers in parallel programs [32, 33]. Dongarra, Foster, and Kennedy [34] have highlighted the importance of this behaviour-structure decoupling when suggesting that the encapsulation of algorithms and techniques fosters the production of reusable parallel programs, stating that “a pattern might specify the problem-independent structure and note where the problem-specific logic must be supplied”. Thus, they advocate raising the level of abstraction without sacrificing performance.

Based on their functionality, skeletons that can be categorised as data parallel, task-parallel, and resolution.

**Data-parallel skeletons** Work typically on bulk data structures. Their behaviour establishes functional correspondences between data, and their structure regulates resource layout at fine-grain parallelism.

**Task-parallel skeletons** Operate on tasks. Their behaviour is determined by the interaction between tasks. They have a variable granularity defined by the parametric code and data instance.
Resolution skeletons Delineate an algorithmic method to undertake a given family of problems. Their behaviour reflects the nature of the solution to a family of problems, and their structure may encompass different computation, communication, and control primitives.

3.1. A Classification for Algorithmic Skeletons

This section elaborates on the functionality associated with the algorithmic skeletons listed in Table I. While not fully comprehensive, this list is certainly representative of the main constructs included in mainstream ASK.F. Be aware that the actual syntax of a given skeleton may vary between ASK.F.

- Data-parallel
  - \textit{Map} is probably the quintessential data parallelism skeleton and its origins are closely related to functional languages. The semantics behind map specify that a function or a sub-skeleton can be applied simultaneously to all the elements of a list to achieve parallelism. The data parallelism occurs because a single data element can be split into multiple data, then the sub-skeleton is executed on each data element, and finally the results are united again into a single result. The map skeleton can be conceived as single instruction, multiple data parallelism.
  - \textit{Fork} behaves like map. The difference is that instead of applying the same function to all elements of a list, a different one is applied to each element. Thus fork can be construed as multiple instruction, multiple data parallelism.
  - \textit{Reduce}, also known as scan, is employed to compute prefix operations in a list by traversing the list from left to right and then applying a function to each pair of elements, typically summation. Note that as opposed to map, it maintains aggregated partial results.

- Task-parallel
  - \textit{Sequential}, simply known as seq, terminates a recursive nesting of skeletons. It wraps execution branches nested into the parallel program as terminal leaves.
  - \textit{Farm} is also known as master-slave/worker or bag of tasks. It embeds the ability to schedule independent tasks in a divisible workload across multiple computing nodes. Skeletons nested inside a farm are meant to be replicated for task parallelism.
  - \textit{Pipe} is one of the canonical task-parallel skeletons in the literature. It enables staged (pipelined) computations, where parallelism can be achieved by computing different stages simultaneously on different inputs. The number of stages provided by pipe can be variable or fixed, but it is worth noting that fixed staged pipes can be nested inside other fixed staged pipes to create a pipe with any number of stages.
  - \textit{If} provides conditional branching. Two sub-skeletons are provided as parameters, along with a conditional branch. When an input arrives at the if skeleton, either one or the other sub-skeleton is executed, depending on the result of the condition branch.
  - \textit{For} receives a sub-skeleton and an integer as parameters. The sub-skeleton will be executed the number of times specified by the integer parameter. The result of one invocation of a sub-skeleton is passed as a parameter to the following invocation of the sub-skeleton. Eventually, the result of the last sub-skeleton is provided as the for skeleton’s result.
  - The \textit{While} skeleton is analogous to the for skeleton but, instead of iterating a fixed number of times, a condition branch decides whether the iteration must continue or stop. On each iteration, the result of the previous execution of the sub-skeleton is passed as input to the same sub-skeleton.

It crucial to emphasise that a few of the aforementioned skeletons can be misconstrued as simple sequential structures supported by most programming languages. However, when included as skeletons in a given ASK.F, they enable parallel flow and control. In fact, it has
Listing 1: Example of the use of a skeleton algorithmic framework. It shows an implementation for the Quicksort with the divide & conquer algorithm skeleton in Skandium

been shown that sequential structures can be re-written into their equivalent normal-form to exploit task-parallelism and, typically, improve performance [35].

- Resolution
  - *Divide & Conquer* (d&c) is a classical resolution skeleton in the literature. A generalisation of the *map* skeleton, d&c maps are recursively applied until a condition is met within a optimisation search space. Its semantics are as follows. When an input arrives, a condition component is invoked on the input. Depending on the result two things can happen. Either the parameter is passed on to the sub-skeleton, or the input is split with the split component into a list of data. Then, for each list element the same process is applied recursively. When no further recursions are performed, the results obtained at each level are combined using a merge skeleton. Eventually, the merged results yield to one result which corresponds to the final result of the d&c skeleton. The depth of the recursion and width of the division may be fixed, or will depend on the invoked components and data provided by the user.
  - *Branch & Bound* (b&b) is a resolution skeleton which also features a generic use of a *map* skeleton. b&b divides recursively the search space (branch) and then determines the elements in the resulting sub-spaces by mapping an objective function (bound). Indeed the effectiveness of the method relies on the suitability of the objective function. Eventually, the merged results also produce one result which corresponds to the final result of the b&b skeleton.

3.2. Programming Example: the quicksort

The code in Listing 1 provides an illustrative program *QuickSort*, a structured parallel implementation of the quicksort algorithm in the Skandium library [36]. It intends to exemplify the common procedure to program with algorithmic skeletons. The overall programming methodology is typical: program definition, object construction, data input, and results retrieval.

1. *Definition* The first thing to do is define the skeleton program. For *QuickSort* a d&c skeleton (indicated within the code as DaC) can be used. The d&c requires four functional components, known as *muscles* in the Skandium lingo, that are defined in lines 3 to 6 in Listing 1: \( f_c: \text{ShouldSplit}; f_s: \text{SplitList}; f_e: \text{Sort}; \) and \( f_m: \text{MergeList}. \) Note that \( f_e \) could be replaced by other skeletons, hence allowing nested skeletal composition. The code for each muscle is presented in Listing 2.

2. *Parameters* Once the skeleton program is defined, we can input parameters for computation. In this case, \( \text{Range} \) (line 9) is a simple class which represents an array, a start and a finish index. The input is performed via the input method which returns a *Future*, that is used to cancel the computation and get the results.
public ShouldSplit(int threshold, int maxTimes) {
    this.threshold = threshold;
    this.maxTimes = maxTimes;
    this.times = 0;
}

@Override
generated synchronized boolean condition(Range r) {
    return r.right - r.left > threshold &&
    times++ < this.maxTimes;
}

public class SplitList implements Split<Range, Range> {
    public Range[] split(Range r) {
        int i = partition(r.array, r.left, r.right);
        Range[] intervals = {new Range(r.array, r.left, i-1),
                              new Range(r.array, i+1, r.right)};
        return intervals;
    }
}

public class Sort implements Execute<Range, Range> {
    @Override
    public Range execute(Range r) {
        if (r.right <= r.left) return r;
        Arrays.sort(r.array, r.left, r.right+1);
        return r;
    }
}

public class MergeList implements Merge<Range, Range> {
    @Override
    public Range merge(Range[] r) {
        Range result = new Range(r[0].array, r[0].left, r[1].right);
        return result;
    }
}

Listing 2: The four components of a QuickSort implementation for Listing 1.

3. Idle time. The use of Future allows asynchronous computation. Therefore, other
   computational tasks can be performed by the main thread whilst waiting for the results. Bear
   in mind that the use of a Future, however, is not the norm in ASKf. Some ASKf provide
   blocking synchronous calls.

4. Results. We retrieve the results and block the process until they are available. Otherwise, an
   Exception is raised.
Table II. Grouping of mainstream algorithmic skeleton frameworks (AS\textsubscript{K}F) according to their programming paradigm.

<table>
<thead>
<tr>
<th>Programming Paradigm</th>
<th>AS\textsubscript{K}F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>llc, P3L, SAC, SCL, Skil</td>
</tr>
<tr>
<td>Functional</td>
<td>Eden, HDC, Skipper</td>
</tr>
<tr>
<td>Object-Oriented</td>
<td>Calcium, HODC, JaSkel, Lithium, Muesli, Mallba, Quaff, Skandium, SkeTo</td>
</tr>
<tr>
<td>Imperative</td>
<td>ASSIST, eSkel, SKElib, SkIE, Skipper</td>
</tr>
</tbody>
</table>

Listing 2 shows the implementation of the \texttt{QuickSort} muscles. Note that some methods have been omitted for clarity purposes.

- \texttt{f_c (Condition)}: The \texttt{ShouldSplit} condition holds if the range of the array is to be subdivided. This happens when the range is larger than a given threshold and the number of maximum subdivisions has not been reached. Note that the \texttt{condition} method has been synchronized to serialise access to the \texttt{times} variable. Since the \texttt{ShouldSplit} condition is composed into a DaC skeleton, it will be used to determine if further splits are required.

- \texttt{f_s (Split)}: \texttt{SplitList} takes an initial range and partitions it into two sub ranges using the \texttt{partition} method. The \texttt{partition} method mirrors the quicksort algorithm: all elements on the left of \texttt{i} are smaller than all elements on the right of \texttt{i}. Afterwards, two new \texttt{Range} objects are created to represent each partition and returned.

- \texttt{f_e (Execute)}: Once the DaC splitting is done, a \texttt{sort} method is applied to each \texttt{Range} of the array. The implementation is simple, we invoke the \texttt{Arrays.sort} Java method on the specific block. Once this method returns, the sub-array within the \texttt{Range} is sorted.

- \texttt{f_m (Merge)}: The final step is to merge back all the \texttt{Range} objects (sub-arrays) into a single one. No actual sorting is required here, as all sub-arrays are already sorted.

4. FUNCTIONAL CLASSIFICATION

The acceptance of the algorithmic skeleton concept to govern the common control structure of parallel applications in computational science [37, 38, 39, 40], along with the release of a programming manifesto [41] and a compilation of different research projects [42], has nurtured the field of structured parallel programming. It has also guided the AS\textsubscript{K}F development and their associated set of control and data constructs, regulating the flow, nesting, monitoring, and portability of structured parallel programs.

AS\textsubscript{K}F can be grouped according to the programming paradigm into:

- Coordination
- Functional
- Object-Oriented
- Imperative

\textbf{Coordination} This approach advocates the use of a high-level language to describe the algorithmic behaviour and a host language to handle interaction with the infrastructure. The Structured Coordination Language (SCL) [43], the Skeleton Imperative Language (Skil) [44], the Pisa Parallel Programming Language (P3L) [45], the llc language [46], and the Single Assignment
C language (SAC) [47] augment imperative languages, with a coordinating language to describe skeletons at high-level. By translating the skeletal description into the host language, they allow the programmer to generate a program by assembling the high-level skeletal portion with the host language structure on top of a low-level parallel software infrastructure, typically MPI. The main disadvantages of the coordination approach are the need to learn a new language, and the necessity to prepare an optimised system infrastructure for the host language, in the form of translators and compilers. Its forte is the clear separation, by design, of the coordination and communication parallel primitives.

**Functional** Structured parallelism has been incorporated into parallel functional languages as syntactic extensions, or as functors within existing languages. On the one hand, the Higher-order Divide-and-Conquer language (HDC) [48] and Eden [49] widen the Haskell scope with parallel extensions to describe skeletal behaviour. They generate executable parallel programs by either translating the program into a C/MPI source in the case of HDC, or by using the Glasgow Haskell Compiler as an infrastructure, in the case of Eden. Furthermore, these ASkF provide specific statements to manipulate processes and input/output data streams and present complete programming environments. On the other hand, skeletons have been introduced through Haskell functors into Concurrent Clean [50], into ML with the Parallel ML with Skeletons (PMLS) notation [51], OCamlP3l [52], and the Skipper system [53], and into Hope [54]. These functors allow the expression of skeletons without disrupting the syntax in the original language. For the purposes of this survey, we have chosen Skipper as the most representative functor-based ASkF to be discussed in detail in Section 5, as different functional implementations continue to appear on a regular basis [55]. Loidl et al. [56] report a comparative performance study with three skeletal programs in Eden, the parallel version of the Glasgow Haskell Compiler, and PMLS, using a C implementation as baseline. While functional ASkF are consistently more elegant in the abstraction of parallelism, C-language implementations provide the best performance.

**Object-Oriented** Skeletal constructs are introduced into object-oriented languages using classes. Based on C++ classes and MPI, the Skeletons in Tokyo (SkeTo) project [57], the Münster Skeleton Library (Muesli) [58], and the Málaga-La Laguna-Barcelona (Mallba) library [59], deploy data-parallel, task-parallel, and resolution skeletons respectively. SkeTo focuses on tree structures, Muesli on coarse-grained control structures, and Mallba on resolution constructs. Additionally, Calcium [60], the Java skeleton-based ASkF JaSkel [61], Lithium [62], muskel [63], Quaff [64], and Skandium [36] furnish distinct skeletons as Java or C++ classes in their integrated object-oriented ASkF. It is important to emphasise that the aforementioned class-centred skeletal libraries rely on the abstraction capabilities of the object-oriented host language and, since they do not impose a special syntax, they rarely introduce a significant overhead into the resulting program. This paradigm has remained buoyant as a result of the popularity of object-oriented languages and the implementation of skeletal libraries may help to address the performance-portability problem.

**Imperative** Skeletons are also deployed as APIs in procedural languages. By procedural calls within a low-level parallel environment, the Skeleton-based Integrated Environment (SkIE) [65], the Software development System based upon Integrated Skeleton Technology (ASSIST) [66], the Pisa’s Skeleton Library (SKElib) [67], and the Edinburgh Skeleton Library (eSkel) [41] deliver data- and task-parallel skeletal APIs. SkIE, the first commercial ASkF, places particular importance on inter-operability and rapid prototyping, as applications can be formed by encapsulating sequential modules in different languages, such as C, C++, Fortran 77 & 90, and Java. eSkel concentrates on portability, as it extends the scope of MPI collective operations by providing a constructive data model, and its emphasis on performance has led to further development [68]. Note that despite the fact that ASSIST and SkIE are implemented on top of C++, we have classified them as imperative because of their programming style. Overall, the portability and performance of this paradigm have greatly benefited from the C language performance capabilities, and the fact that it does not introduce syntactic extensions.
allows its insertion into existing application environments. A consistent criticism of this approach is the inability to enforce type checking.

5. INDIVIDUALISED DESCRIPTION OF FEATURES

Section 4 has furnished a general classification of the main ASkF based on their programming paradigm. This section examines their functionality in detail. Note that each subsection typically comprises one or more ASkF typically developed by a research group or project.

5.1. ASSIST

ASSIST provides a structured coordination language to express parallel programs as an arbitrary graph of software modules [66]. The module graph describes how a set of modules interact with each other using a set of typed data streams. Modules can be sequential or parallel. Sequential modules can be written in C, C++, or Fortran, while parallel modules are programmed with a special ASSIST parallel module (parmod) [69].

A significant infrastructure component of ASSIST, AdHoc is a hierarchical and fault-tolerant distributed shared memory infrastructure to interconnect streams of data between processing elements by providing a repository with get, put, remove, and execute operations [70, 71]. It is primarily focused on providing integration, scalability, and fault-tolerance to the data repository access.

Programming in ASSIST is peculiar, as skeletons are not pre-defined, and the programmer is expected to specialise its generic parmod construct into skeletons such as farm or map. It also supports autonomic control [72] and can be subject to a performance contract by dynamically adapting the number of resources used.

ASSIST also furnishes load balancing mechanisms through an application manager [72]. This manager uses configuration-safe points within the program to enable load balancing when a bottleneck is encountered. While the reported results on reconfiguration overheads are interesting, scant reference is made to the allocation policies employed, either at the start of the application or at reconfiguration.

5.2. Calcium and Skandium

Predominantly inspired by Lithium and muskel, Calcium [60] supplies skeletons in a Java library. Both task and data parallel skeletons are fully nestable and are instantiated via parametric objects.

Calcium supports the execution of skeleton applications on several parallel and distributed infrastructures including a multi-threaded environment for symmetric multiprocessing infrastructures, the ProActive environment for cluster computing, and a grid scheduler. Calcium provides a single way of writing skeletal programs which can be deployed and executed on different parallel and distributed infrastructures without changes.

Additionally, Calcium has three distinctive features for algorithmic skeleton programming. Firstly, a performance tuning model which helps programmers identify code responsible for performance bugs [73]. Secondly, a type system for nestable skeletons implemented using Java Generics [60]. Thirdly, a transparent algorithmic skeleton file access model for data intensive applications [74].

Recent research from this group has centred on Skandium, a complete re-implementation of Calcium for multi-core computing. Programs written on Skandium take advantage of shared memory to simplify parallel computing. Programs written on Skandium take advantage of shared memory to simplify parallel programming [36].

5.3. Eden

An extension to Haskell, Eden [49] provides support for parallel and distributed environments. Processes are defined explicitly to achieve parallel programming, while their communications remain implicit [75]. Processes communicate through unidirectional channels, which connect one
writer to exactly one reader. Programmers only need to specify which data a processes depends on. Eden’s process model provides direct control over process granularity, data distribution and communication topology.

Eden has introduced the concept of the implementation skeleton [76], which is an architecture independent scheme that describes the parallel implementation of an algorithmic skeleton. Within Eden, skeletons are defined on top of its lower-level process abstraction, supporting both task- and data-parallelism.

Recent research on Eden has focused on scalability and resource awareness. A hierarchical task pool has now been included to handle dynamically-created tasks in the computation nodes [77, 78] and to distribute the computational workload using work stealing algorithms [79].

5.4. eSkel

The Edinburgh Skeleton Library (eSkel) is deployed in C and runs on top of MPI. The first version of eSkel was described in [41], while a later version is presented in [68].

The eSkel has two distinct modes, nesting and interaction, where skeletons are defined [80]. The nesting mode can either be transient or persistent, while the interaction mode can either be implicit or explicit. Transient nesting means that the nested skeleton is instantiated for each invocation and destroyed afterwards, while persistent means that the skeleton is instantiated once and the same skeleton instance will be invoked throughout the application. Implicit interaction means that the flow of data between skeletons is completely defined by the skeleton composition, while explicit means that data can be generated or removed from the flow in a way not specified by the skeleton composition. For example, a skeleton that produces an output without ever receiving an input has explicit interaction.

eSkel has initially employed empirical methods [81] and, subsequently, process algebra through Amoget [82] in order to predict performance. Amoget, a pre-execution Perl scripting feature, generates a prognostic performance model for a given skeleton which is then fed to the eSkel library to select the resources accordingly. In light of those initial results, where sudden variations in resource usage were not immediately managed by Amoget, eSkel has been extended to incorporate reactive process-algebra scheduling as part of its main library. Under this context, works on scheduling and resource mapping, mainly for pipeline skeletons, have been published. They provide performance models for each mapping [83], or a determination of the complexity of the models [84].

More recent works from the Edinburgh group have addressed the problem of adaptation on structured parallel programming [85, 86], in particular for the pipe skeleton [87, 88] and the farm [89].

Furthermore, this group has extended their research to develop a methodology known as Adaptive Structured Parallelism which puts particular emphasis on the automatic scheduling of algorithmic skeletons [90]. It instruments a skeletal program with a series of rules, which depend on particular performance thresholds based on the nature of the skeleton, the computation/communication ratio of the program, and the availability of resources in the system. Employing the pipe and farm skeletons as a basis for experimentation, this methodology has been successfully applied to allocate divisible workloads to real multi-node configurations [91] and to solve complex parameter sweeps in the biomedical sciences [92].

5.5. HDC

HDC is a subset of the functional language of Haskell [48]. Functional programs are presented as polymorphic higher-order functions, which can be compiled into C with MPI, and linked with skeleton implementations. The language centres on the divide and conquer paradigm, starting from a general kind of divide and conquer skeleton to derive specific cases with efficient implementations. The specific cases, handled by the compiler, correspond to fixed recursion depth, constant recursion degree, multiple block recursion, element-wise operations, and correspondent communications. Subsequent extensions to HDC allow skeletal requirements to be handled independently from the compiler [93].
HDC pays special attention to the subproblem’s granularity and its relation to the number of available processors. The total number of processors is a key parameter for the performance of the skeleton program as HDC strives to estimate an adequate assignment of processors for each part of the program. Thus, the performance of the application is strongly related to the estimated number of processors leading to either exceeding the number of subproblems, or to there not being enough parallelism to exploit the available processors.

5.6. HOC & Alt

Alt et al. have proposed a series of ASKF for distributed systems and grids [94, 95, 96], collectively referred to in this work as the Alt ASKF. In the Alt ASKF, skeletons are offered as services accessible through Java Remote Method Invocation [97]. Once a skeleton service has been found, the skeleton is remotely called from the client program using an invoke API. Special container objects are used to pass parameters and obtain results between different skeleton invocations. Contrary to other ASKF, skeletons are not nestable. The control flow between skeletons is explicitly manipulated by the programmer from inside the client application. Thus, type verification is explicitly performed by Java in the client program.

The concept has been later conceptualised as the Higher Order Component (HOC). HOCs combine skeletons, components, and services to enable the remote client access to distant services which implement a parallelism construct [98]. A remote client provides the specific application code to the HOC and the input data. The code and data are shipped from the client host to the remote service host. Then, the HOC is deployed and executed on the remote infrastructure in accordance with the specific HOC pattern. Once the computation is finished, the result is delivered back to the programmer. A description language regulates the execution flow between different skeletal services.

As a HOC is mainly focused on how a skeleton can be remotely accessed as a service, its main contributions are the storage and look up of remote services and the code shipping from the client. On the back-end, HOCs have been coupled with different grid middleware packages including Globus [98, 99].

5.7. JaSkel

JaSkel [61] provides skeletons such as farm, pipe and heartbeat, where skeletons are specialised using inheritance. Programmers are expected to implement the abstract methods for each skeleton to provide their application specific code.

Skeletons in JaSkel are provided in sequential, concurrent, and distributed versions such as OcamlP3l [52]. For example, the concurrent farm can be used in shared memory environments with threads, while in clusters a distributed farm is used. To change from one version to another, programmers must change their classes’ signature to inherit from a different class. The distributed aspects of the computation are handled in JaSkel with aspect-oriented programming, more specifically with AspectJ [100]. Thus, JaSkel can be deployed on both cluster and grid infrastructures [101, 102].

The nesting of skeletons uses the basic Java class system and therefore no type system is enforced during composition. A drawback of the JaSkel approach is that the nesting of the skeleton strictly relates to the deployment infrastructure. Thus, a double nesting of a farm yields a better performance than a single farm on hierarchical infrastructures. This appears to be counterintuitive to the separation of distribution and functional concerns in skeletal programs.

5.8. Lithium and Muskel

Lithium [62] and its successor muskel [103] are ASKF developed at Università di Pisa. Both provide nestable skeletons to the programmer as Java libraries and a formal semantics to handle both task and data parallelism [104, 105]. Such semantics describe both functional and parallel behaviour of the skeletal language using a labelled transition system. Performance optimisation is deployed through skeleton rewriting techniques, task look-ahead, and server-to-server lazy binding [106].
At the implementation level, Lithium exploits macro-data flow to achieve parallelism [107]. When the input stream receives a new parameter, the skeleton program is processed to obtain a macro-data flow graph. The nodes of the graph are macro-data flow instructions, which represent the sequential pieces of code provided by the programmer. Tasks are used to group together several instructions and are consumed by idle processing elements from a task pool. When the computation of the graph is concluded, the result is placed into the output stream and thus delivered back to the user.

Lithium has been extended to provide future-based Java RMI optimisation mechanisms to enhance load-balancing through a statically-defined thread interval [108]. This interval, typically of two to six threads, preemptively controls the node load. Although reported results provide favourable guidance, it is unclear how the interval is defined.

Moreover, Muskel provides performance-oriented features such as quality of service [109]; security between task pool and interpreters [110, 111]; and resource discovery, load balancing, and fault tolerance when interfaced with the Java and Jini technologies [112]. More recent research from the Pisa group has addressed skeletal extensibility [63], autonomic components [113], and behavioural skeletons [114]. Behavioural skeletons are of particular interest for multi-site, distributed computing as they deploy, in addition to the standard pattern, the management of concerns such as performance tuning or fault tolerance to enable autonomic execution [115].

5.9. llc & Mallba

The llc and Mallba ASKf have been primarily championed by the parallel algorithms and languages group at Universidad de La Laguna.

The llc language implements an OpenMP-style syntax to describe skeletal algorithms which use C/MPI as their host [116]. It automatically generates MPI code for a given set of skeletons. The main innovations in llc are the use of pragma directives in its notation and the support of hybrid MPI-OpenMP environments [117]. llc has been successfully applied to the solution of linear algebra problems with irregular parallelism [118].

Mallba [59] is a library for combinatorial optimisations supporting exact and heuristic search strategies [119]. Each strategy is implemented in Mallba as a generic skeleton which can be used by providing the required code. It provides branch-and-bound and dynamic-optimisation skeletons for exact search; and hill climbing, metropolis, simulated annealing, and tabu search for heuristic search. It also allows population-based heuristics derived from evolutionary computing. Hybrid skeletons can combine strategies e.g. genetic algorithms and simulated annealing. Comprised in a C++ library, skeletons are type safe but not nestable. A custom MPI abstraction layer, NetStream, takes care of primitive data type marshalling and synchronisation. It instruments the skeleton structure in the library, providing high-level network communication services. Resource selection is based on the readings of network links and node load, but according to the authors [119], they are “still at the stage of developing intelligent algorithms to use this [network] information to perform a more efficient search [of resources].” Consequently, a skeleton may have multiple lower-level parallel implementations depending on the target architectures: sequential, LAN, and WAN, e.g. centralised master-slave or distributed master-slave.

Mallba also provides state variables which hold the state of the search skeleton. The state links the search with the environment, and can be accessed to inspect the evolution of the search and decide on future actions. For example, the state can be used to store the best solution found so far, or the values for branch and bound pruning [120]. It is interesting to emphasise that, in contrast to other ASKf, Mallba defines skeletons as parametric search strategies rather than parameterisable parallel constructs.

5.10. P3L, SkIE, SKElib

P3L is a skeleton based coordination language [121]. P3L provides skeleton constructs which are used to coordinate the parallel or sequential execution of C code. A compiler is provided for the language. It uses implementation templates to compile P3L code into a target architecture. Thus, a skeleton can have several templates, each optimised for a different architecture. A template implements a skeleton on a specific architecture and provides a parametric process graph with a
A $P^3$L module corresponds to a properly defined skeleton construct with input and output streams, and other sub-modules or sequential C code. Modules can be nested using the two-tier model, where the outer level is composed of task parallel skeletons, while data parallel skeletons may be used in the inner level [123]. Type verification is performed at the data flow level, when the programmer explicitly specifies the type of the input and output streams, and by specifying the flow of data between sub-modules.

SkIE [124] is quite similar to $P^3$L, as it is also based on a coordination language, but provides advanced features such as debugging tools, performance analysis, visualisation and graphical user interface. Instead of directly using the coordination language, programmers interact with a graphical tool, where parallel modules based on skeletons can be composed.

SKELib [67] builds upon the contributions of $P^3$L and SkIE by inheriting, among others, the template system. It differs from them because a coordination language is no longer used, but instead skeletons are provided as a library in C, with performance similar to the one achieved in $P^3$L. Contrary to Skil, type safety is not addressed in SKELib.

5.11. SAC

SAC, a dedicated array imperative language with HPF-like syntax, supplies multi-threaded vector operators and loop coordination statements on top of a host language and Pthreads [125]. It supplies a limited set of array primitives such as rank (dimension), shape, and elements. Through the use of the $\textit{with}$ operator, SAC allows the creation of data-parallel skeletons such as $\textit{fold}$ that specifies a reduction operation.

Its contributions are the purity of its coordination approach, which provides a clear-cut separation of the behaviour and structure of a parallel program, and its performance-oriented approach to the solution of computational problems [126]. SAC has been recently augmented to support multi-core architectures [127].

5.12. SCL

One of the first languages introduced for skeletal programming, the Structured Coordination Language (SCL) [33, 54] is considered to be a base language, and is designed to be integrated with a host language, typically Fortran.

In SCL, skeletons are classified into three types: configuration, elementary and computation. Configuration skeletons abstract patterns for commonly used data structures such as distributed arrays (ParArray). Elementary skeletons correspond to data parallel skeletons such as map, scan, and fold. Computation skeletons which abstract the control flow and correspond mainly to task parallel skeletons including farm, SPMD, and iterateUntil.

SCL skeletons are instantiated in Fortran, but a standard Fortran program cannot directly invoke an SCL primitive [128].

5.13. SkeTo

A C++ library coupled with MPI, SkeTo is dissimilar to other ASkF because, instead of providing nestable parallelism patterns, it offers operations for parallel data structures such as lists [57], trees [129, 130], and matrices [131].

Parallelism therefore resides in data structures which are typed using templates, allowing a defined set of operations. For example, data structures permit not only traditional data-parallel operations (skeletons) such as map, reduce, scan, zip, and shift, but also resolution ones such as divide & conquer.

SkeTo is nestable as SkeTo provides a fusion transformation [132] to merge two successive function invocations into a single one. It therefore decreases the function call overheads and avoids the creation of intermediate data structures passed between functions.
Additional research around SkeTo has also focused on improving their optimisation strategies for the construction of data structures [133] as well as the formalisation of their operations using functional programming theory [134].

5.14. Skil and Muesli

Within Skil [44], skeletons are not directly part of the language but are implemented with it [135, 136]. Skil uses a subset of C language which provides functional language like features such as higher order functions, currying and polymorphic types [137, 138]. When Skil is compiled, such features are eliminated and a regular C code is produced. Thus, Skil transforms polymorphic higher-order functions into monomorphic first-order C functions. Skil does not support nestable composition of skeletons. Data parallelism is achieved using specific data parallel structures, for example to spread arrays among available processors.

The Muesli skeleton library [58] builds upon many of the concepts introduced in Skil, but instead of a subset of the C language, skeletons are offered through C++ methods [139, 140, 141]. Contrary to Skil, Muesli supports the nesting of task and data parallel skeletons [142] but is limited to two levels, as in P^3L. C++ templates are used to render skeletons polymorphic. The supported skeletons are distributed array and matrix for data parallelism; and pipeline, farm, and parallel composition (a farm variant) for task parallelism [143]. Recent versions of Muesli also support multi-core programming with OpenMP additionally to cluster support which is done via MPI.

Additional research has focused mostly on optimisation and the scalability for specific skeletons such as the farm, branch and bound, and divide and conquer [144, 145].

5.15. SKiPPER & QUAFF

SKiPPER [53] is a domain specific skeleton library for vision applications which provides skeletons in Caml, a popular version of ML. It benefits from the Caml type safety enforcement. Skeletons are presented in two ways: declarative and operational. Declarative skeletons are directly used by programmers, while their operational versions provide an architecture specific target implementation [146]. The Caml skeleton specifications and application-specific functions have been traditionally transformed into C code and compiled to run the parallel application on the target architecture. An interesting characteristic about SKiPPER is that the skeletal program can be executed sequentially for debugging.

Different approaches have been explored in SKiPPER for writing operational skeletons: static data-flow graphs, parametric process networks, hierarchical task graphs, and tagged-token data-flow graphs [53].

QUAFF [64] is a more recent skeleton library written in C++ and MPI. QUAFF relies on template-based meta-programming techniques to reduce runtime overheads and perform skeleton expansions and optimisations at compilation time. Skeletons can be nested and sequential functions are stateful. Besides type checking, QUAFF takes advantage of C++ templates to generate, at compilation time, new C/MPI code. QUAFF is based on the CSP-model, where the skeletons are described as a process network with single, serial, par, join production rules [147].

6. CRITICAL COMPARISON

This section renders a critical comparative perspective of different ASkF. Table III lists the ASkF from Section 5 and appraises them in terms of the following six main characteristics:

1. Programming language.
2. Execution language.
3. Distribution environment.
4. Type safety.
5. Skeleton nesting.
Table III. Comparative table of the algorithmic skeleton frameworks considered in this survey.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Programming Language</th>
<th>Execution Language</th>
<th>Distribution Library</th>
<th>Type Safe</th>
<th>Skeleton Nesting</th>
<th>Skeleton Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt</td>
<td>Java</td>
<td>Java</td>
<td>Java RMI</td>
<td>yes</td>
<td>no</td>
<td>map, zip, apply, scan, sort, reduce, replicate</td>
</tr>
<tr>
<td>ASSIST</td>
<td>Custom Control Lang.</td>
<td>C++</td>
<td>TCP/IP + ssh/scp</td>
<td>yes</td>
<td>no</td>
<td>seq, parmod</td>
</tr>
<tr>
<td>Calcium</td>
<td>Java</td>
<td>Java</td>
<td>ProActive</td>
<td>yes</td>
<td>yes</td>
<td>seq, pipe, farm, for, while, map, d&amp;c, fork</td>
</tr>
<tr>
<td>Eden</td>
<td>Haskell (extension)</td>
<td>C</td>
<td>PVM / MPI</td>
<td>yes</td>
<td>yes</td>
<td>map, d&amp;c, pipe, pipe, iterUntil, torus, ring</td>
</tr>
<tr>
<td>eSkel</td>
<td>C</td>
<td>C</td>
<td>MPI</td>
<td>no</td>
<td>yes</td>
<td>pipe, farm, deal, butterfly, hallowSwap</td>
</tr>
<tr>
<td>HDC</td>
<td>Haskell (subset)</td>
<td>C</td>
<td>MPI</td>
<td>yes</td>
<td>yes</td>
<td>map, red, scan, filter, dcA, dcB, dcD, dcE, dcF</td>
</tr>
<tr>
<td>HOC</td>
<td>Java</td>
<td>Java</td>
<td>Globus</td>
<td>no</td>
<td>no</td>
<td>farm, pipe, wavefront</td>
</tr>
<tr>
<td>JaSkel</td>
<td>Java</td>
<td>Java</td>
<td>RMI</td>
<td>no</td>
<td>yes</td>
<td>farm, pipe, heartbeat</td>
</tr>
<tr>
<td>Lithium</td>
<td>Java</td>
<td>Java</td>
<td>RMI</td>
<td>no</td>
<td>yes</td>
<td>pipe, map, farm, reduce</td>
</tr>
<tr>
<td>Malba</td>
<td>C++</td>
<td>C++</td>
<td>NetStream / MPI</td>
<td>yes</td>
<td>no</td>
<td>exact, heuristic, hybrid</td>
</tr>
<tr>
<td>Muesli</td>
<td>C++</td>
<td>C++</td>
<td>MPI</td>
<td>yes</td>
<td>limited</td>
<td>array, matrix, farm, pipe, parallel comp.</td>
</tr>
<tr>
<td>Muskel</td>
<td>Java</td>
<td>Java</td>
<td>RMI</td>
<td>no</td>
<td>yes</td>
<td>farm, pipe, seq, custom</td>
</tr>
<tr>
<td>P3L</td>
<td>Custom Cont. Lang.</td>
<td>C</td>
<td>MPI</td>
<td>yes</td>
<td>limited</td>
<td>map, reduce, seq, comp, pipe, farm, scan, loop</td>
</tr>
<tr>
<td>QUAFF</td>
<td>C++</td>
<td>C</td>
<td>MPI</td>
<td>yes</td>
<td>yes</td>
<td>seq, pipe, farm scm, pardo</td>
</tr>
<tr>
<td>SAC</td>
<td>Custom</td>
<td>C-like</td>
<td>Threads</td>
<td>no</td>
<td>no</td>
<td>genarray, modarray, fold</td>
</tr>
<tr>
<td>SCL</td>
<td>Custom Cont. Lang.</td>
<td>Fortran</td>
<td>Ad-hoc Tools</td>
<td>yes</td>
<td>limited</td>
<td>map, scan, farm, fold, SPMD, iterateUntil</td>
</tr>
<tr>
<td>Skandium</td>
<td>Java</td>
<td>Java</td>
<td>Threads</td>
<td>yes</td>
<td>yes</td>
<td>seq, pipe, farm, for, while, map, d&amp;c, fork</td>
</tr>
<tr>
<td>SKELib</td>
<td>C</td>
<td>C</td>
<td>MPI</td>
<td>no</td>
<td>no</td>
<td>farm, pipe</td>
</tr>
<tr>
<td>SkeTo</td>
<td>C++</td>
<td>C++</td>
<td>MPI</td>
<td>yes</td>
<td>no</td>
<td>list, matrix, tree</td>
</tr>
<tr>
<td>SkIE</td>
<td>GUI / Custom Cont. Lang.</td>
<td>C++</td>
<td>MPI</td>
<td>yes</td>
<td>limited</td>
<td>farm, pipe, map reduce, loop</td>
</tr>
<tr>
<td>Skil</td>
<td>C (subset)</td>
<td>C</td>
<td>–</td>
<td>yes</td>
<td>no</td>
<td>pardata, map, fold</td>
</tr>
<tr>
<td>SkiPPER</td>
<td>CAML</td>
<td>C</td>
<td>SynDex</td>
<td>yes</td>
<td>limited</td>
<td>scm, df, tf, intermem</td>
</tr>
</tbody>
</table>
Programming Language is the interface with which programmers interact to code their skeleton applications. The paradigms have been diverse, encompassing functional, coordination, markup, imperative, object oriented, and graphical. Skeletons are therefore accessible as language constructs or APIs in libraries. Providing skeletons as language constructs implies the development of a custom domain-specific language and its compiler, the trend in the early days. More recently, skeletons are likely to be presented as libraries in an established language such as C, C++, or Java.

Execution Language is the target language which the skeleton applications are compiled into. Note that the execution language may be different from the programming language, especially when dealing with functional languages where performance has been considered an issue. Therefore, transformation processes have been introduced to convert the skeletal applications, defined in the programming language, into an equivalent application in the target execution language. Different transformation processes—code generation or instantiation of lower level skeletons (sometimes called operational skeletons)—interact with a library in the execution language. The transformed application can sometimes introduce target architecture code, customised for performance, into the transformed parallel application. So far, the most common execution language has been C.

Distribution Library provides the functionality to achieve parallel/distributed computations. The norm has, unsurprisingly, been MPI since it integrates well with the C language and is probably the most widely-used parallel infrastructure. The disadvantages of directly programming in the distribution library are, of course, safely hidden away from programmers by the ASkF.

Type Safety refers to the capability of detecting type incompatibility errors in skeletal programming. On the one hand, ASkF built on top of functional languages, have the type safety enforced by the host language. On the other hand, custom languages and non-functional libraries rely on compilers and ad-hoc features for type checking.

Skeleton Nesting is the capability of hierarchical composition of skeletons, an important feature in structured parallel programming [41]. It allows the composition of more complex patterns starting from a basic set of simpler patterns. Nevertheless, it has taken the community a long time to fully support the arbitrary nesting of skeletons, mainly because of the scheduling and type checking difficulties. The norm is the full support of skeleton nesting.

Skeleton Set is the list of supported skeleton patterns. It is important to emphasise that members, naming conventions, and description of the set vary greatly among different ASkF. In fact, skeletons with the same name can have different syntax across ASkF. Syntactical differences aside, the most common skeleton patterns in the literature are the farm, the pipe, and map.

7. RELATED APPROACHES

In addition to the described approach to high-level parallel programming, there have been different models to foster the separation of computation and coordination. Arguably, the most representative are patterns, templates, and components. Early comparative studies can be found in [148, 149].

7.1. Patterns

Having been designed as abstractions of common themes in object-oriented programming [150, 151], patterns have been incorporated into parallel programming [152]. Pattern-based parallel programming allows an application programmer the freedom to generate parallel codes by parameterising the abstractions and adding the sequential parts [153, 154].

The parallel programming pattern concept has been extended into a design method under the umbrella of parallel pattern languages. Unlike other parallel programming languages, parallel
pattern languages present rules to design parallel codes based on: archetypes—problem-class abstractions which describe parallel structure, dataflow, and communication [155]; critical region locks, such as test-and-set and queued for simple mutual exclusion, or reader/writer for concurrent execution [156]; or socket-based operators for web applications [157]. Furthermore, Triana deploys a pattern-based programming framework for scientific workflows [158].

Developed by Google for the efficient deployment of computationally-intensive parallel algorithms, MapReduce is a distributed programming model and framework used to compute problems that can be parallelised by mapping a function over a given dataset and then combining the results [159, 160]. As a framework, it is used to implement MapReduce jobs which encapsulate the features of the model while hiding the complexities inherent in parallelism from users. This framework is, arguably, the largest pattern framework in operation, and has spun off different open-source development projects such as Hadoop [161] and Phoenix [162].

A pattern-based C++ library for parallel programming, the Threading Building Blocks (TBB) has been developed by Intel to take advantage of multi-core architectures [163]. TBB offers parallel patterns including for, reduce, scan, do, sort, and pipeline; and concurrent data structures such as hashmap, vector, and queue [164]. TBB provides abstractions with more control on low-level parallelism aspects such as granularity, the possibility to combine with other thread libraries, and direct access to the task scheduler. It is important to emphasise that TBB is only aimed at shared-memory infrastructures, especially multi-core processors.

A major component of the parallel software offering from Microsoft, the Task Parallel Library (TPL) provides concurrency support for the Microsoft .NET framework and does not require any language extensions [165]. It furnishes parallel constructs such as aggregate, do and for with work-stealing features in order to keep different thread queues balanced.

An interesting aspect of MapReduce, TBB, and TPL is that, contrary to most of the ASkF described here, they have been created by major industry powerhouses as opposed to academic institutions. Parallel programming patterns and their derived languages and frameworks have maintained, arguably, the best adoption rate; however, they have become conglomerates of generic attributes for specific purposes, oriented towards code generation rather than the abstraction of structural attributes.

### 7.2. Templates

Commonly denoted in object-oriented programming by templates or generics, type parameterisation resembles higher-order functions, where instantiation is performed through the types received. Templates provide support for data-parallel programming through object and collective operations such as join, map, and reduce [166], domain-specific operators, e.g., multidimensional arrays and objects to model particle physics experiments [167], or C++ matrix and vector data structures [168].

Incorporated into the C++ standard through the C++ Standard Template Library (STL) [169], templates can contain pointers to different language implementations, computation, and tuning considerations, and information on their usage. STAPL, the Standard Template Adaptive Parallel Library, is widely regarded as the most representative parallel implementation of STL. It has successfully applied generic programming techniques through abstract data structures in order to deploy parallel algorithms [170, 171]. Subsequently, template extensions to the MPI [172] and OpenMP [173] standards have been proposed in order to use built-in lexical abstractions for generic parallel programming, and diverse template libraries now support multi-core environments [174] and self-scheduling task parallelism [175].

In summary, generic parallel programming can be conceived as an expansion to the templates concept which provides a method for the automated development or the platform optimisation of parallel libraries. Templates have demonstrated an interesting potential in high-level parallel programming [176], mostly confined to the development of scientific applications [177, 178, 179].

### 7.3. Components

Components are objects to associate operations with events. A component model is a set of objects with published interfaces which comply with a set of rules defined by a specific concurrent model.
A component model augmented with a set of system components defines a component architecture, where parallel programs are assembled using independent components.

Components have been particularly effective in the deployment of infrastructure services—messaging, access and security, information and directory, job submission, scheduling, and user support—in distributed environments [180, 181], leading to the creation of the Common Component Architecture [182], a standard for the development of component-based applications.

In terms of component aggregation, generative programming proposes the automatic selection and assembly of components on demand, where the programmer specifies the application in a domain-specific language [183]. Aspect-oriented methods propose the self-assembling integration of existing application and platform code components—with minimal modification to original sequential structures with different crosscutting concerns—by using inter-component monitoring and cross-component manipulation of internal states [184]. Nonetheless, despite its popularity in distributed systems programming, component-based programming has long posed a major challenge to system complexity and compatibility, due to the multiplicity of sources and formats of components [185].

There has been cross-pollination of the aforesaid parallel programming models and the skeletal paradigm, resulting in evolved hybrid methods which include: skeletal libraries implemented using C++ templates [64, 141]; components deploying skeletal functionalities for mobile environments [186], autonomic computing [115], and grids [187]; and the encapsulation of high-level communications and processing by implementing common parallel blocks and low-level communication primitives, i.e., parallel patterns with skeletal conduct [188, 189].

Nevertheless, we consider it important to emphasise the distinction between the ‘puristic’ ASkF and the other approaches presented in this section. In the former, skeletons are the raison d’être for the formulation and implementation of all structured parallelism solutions; in the latter, the demonstrated capabilities resemble skeletal constructs but do not specifically enforce the use of skeletons as such.

8. CURRENT TRENDS AND CONCLUSIONS

The most visible research directions in ASkF appear to be infrastructure support with particular emphasis on multi-core processors and resource adaptivity.

Infrastructure Support The advent of multi-core processors, chip multiprocessors, and multi-node clusters and constellations has steeply increased the number of concurrent processors available to a single application. From a single node perspective, dozens of cores have begun to be commonplace and, as a result, the machine with the largest number of cores according to the June 2010 Top500 list [190], JUGENE, the Blue Gene/P at Julich, features 294,912 processors. Several ASkF provide support for multi-core architectures including Muesli, Skandum, and SAC. Additionally, distinct ASkF can interact with more than one parallelisation library. For example, Mallba can use Netstream and MPI; llc and Muesli can provide multi-tier MPI-OpenMP code, and JaSkel uses AspectJ to execute the skeleton applications on different ASkF.

Resource Adaptivity Vadhiyar and Dongarra [191] suggest that a “self-adaptive software system examines the characteristics of the computing environments and chooses the software parameters needed to achieve high efficiency on that environment”. In other words, it is suggested that efficient ASkF ought to furnish resource awareness to allow the correct selection of resources (processors, links) from amongst those available; the correct adjustment of algorithmic parameters (for example, blocking of communications, granularity); and, most importantly, the ability to adjust all of these factors dynamically in the light of...
evolving external pressure on the chosen resources. Recent surveys on adaptive high-level parallel systems have only reinforced the importance of resource awareness for the automatic optimisation of parallel codes in heterogeneous distributed systems. Cunha, Rana, and Medeiros [192] cite a series of component-based problem-solving environments which “allow a clear separation between computation and interaction.” While the list is far from comprehensive, it provides clear guidance on the need for enhanced high-level parallel programming tools. It is important to emphasise that ASKF enact resource awareness differently by i) applying ab-initio resource-node matching strategies, based on theoretical performance cost modelling (eSkel and Lithium); ii) current resource usage only (Mallba); iii) reactive modification of the resource allocation regardless of the application structure (ASSIST); and iv) dispersion-based heuristics to allow responsive application steering in the Adaptive Structured Parallelism methodology from Edinburgh. It is also important to stress the fact that effective ASKF should use online resource monitoring, which favourably compares to the use of offline or theoretical models. It permits the automatic feed of the resource status at execution, which in turn allows a more accurate feedback process.

8.1. Concluding Remarks

This paper has reviewed the current state of algorithmic skeleton frameworks as enablers for high-level structured parallel programming. Conceived as higher order functions corresponding to good parallel algorithmic techniques, algorithmic skeletons have proved to be effective enablers for high-level, structured parallel programming solutions. The term structured parallelism is often assigned to parallel programming with algorithmic skeletons as an implicit association with principles of the structured sequential programming model, where the programmer must adhere to top-down design and construction, limited control structures, and limited scope of data structures.

Having initially set the context by presenting a brief overview of the fundamentals of shared-memory and message-passing programming, this paper has analysed the algorithmic skeleton concept and the different implementations and paradigms for ASKF. We have concluded with a section on how favourably ASKF compare to other reusable distributed and parallel programming tools based on patterns, templates, and components.

Despite its elegance and potential, it is important to state that structured parallelism still lacks the necessary critical mass to become a mainstream parallel programming technique. Its principal shortcomings are its application space, since it can only address well-defined algorithmic solutions, and the lack of a specification to define and exchange skeletons between different implementations. Some consideration has already been devoted to the matter and future research may lead to a complete standard.

ACKNOWLEDGEMENTS

This work is primarily based on the authors’ theses in the field of algorithmic skeletons written at the Universities of Edinburgh and Nice-Sophia Antipolis respectively. Several individuals have contributed to the authors’ progress in this research, particularly their respective supervisors Murray Cole and Denis Caromel; and their thesis committees composed of M. O’Boyle, M.A. Baker, and O.F. Rana at Edinburgh and of M. Cole, M. Danelutto, and F. Loulergue at Nice-Sophia Antipolis.

REFERENCES


Copyright © 0000 John Wiley & Sons, Ltd. Softw. Pract. Exper. (0000)
Prepared using speauth.cls DOI: 10.1002/spe
ALGORITHMIC SKELETON FRAMEWORKS


34. Dongarra J, Foster I, Kennedy K. Reusable software and algorithms. Dongarra et al. [193], chap. 17, 483–490.


Copyright © 0000 John Wiley & Sons, Ltd.
Softw. Pract. Exper. (0000) DOI: 10.1002/spe
Prepared using speauth.cls


ALGORITHMIC SKELETON FRAMEWORKS


