# Skel: A Streaming Process-based Skeleton Library for Erlang (Early Draft!)

Archibald Elliott[1], Christopher Brown[1], Marco Danelutto[2], and Kevin Hammond[1]

[1] School of Computer Science, University of St Andrews, Scotland, UK.
[2] Dept. Computer Science, University of Pisa, Pisa, Italy.

**Emails:** ashe@st-andrews.ac.uk, {chrisb, kh}@cs.st-andrews.ac.uk, marcod@di.unipi.it

**Abstract.** With the increasing shift towards parallel programming, parallel programs are still notoriously difficult to implement. Indeed, parallelism is usually made even more difficult by programmers typically using sequential or small-scale parallel programming techniques. Functional languages such as Erlang are increasingly starting to dominate the parallelism scene with their typical first-class light-weight parallelism models.
In this paper we introduce a new process-based skeleton framework for Erlang based on streaming models. In particular, we show a number of core "primitive" skeletons that can be used as a basis of more complex parallel systems. We give details of the skeletons, in Erlang, and in our full paper we promise to show promising and scalable speedups on a manycore machine on a number of examplars.

## 1 Introduction

The single-core processor, which has dominated for more than half a century is now obsolete. Machines with dual-, quad- and even hexa-core CPUs are already common place in desktop machines and CPUs with 50 cores as standard have already been announced [3]. There has been a *seismic* shift between sequential and parallel hardware, but programming models have been very slow to keep pace. Indeed, many programmers still use outdated sequential models for programming parallel systems, where parallel concepts have effectively been *bolted-on* to the language, rather than high-level parallel constructs being *first-class*. What is needed is an effective solution to help programmers *think parallel*. In the context of parallel programming, parallel design patterns represent a natural language description of a recurring problem and of the associated solution techniques that the parallel programmer may use to solve that problem.

[3] Intel's Many Integrated Core Family

An *algorithmic skeleton*, is a computational, abstract entity, typically described by a concurrent activity graph, modelling and embedding a frequently recurring parallelism exploitation pattern, provided to the application programmer as a new abstraction in the programming framework at hand; a parallel application is therefore developed as a composition of skeletons, which may be specialised by providing (suitably wrapped) sequential portions of code implementing the business logic of the application.

In this paper we introduce `skel`: a process-based streaming skeleton library for Erlang. `skel` aims to model the most common set of "primitive" skeletons that are typically used to make up more complex systems.

In particular, the contributions of our paper are:

1. we describe a new parallel skeleton library for Erlang. To our knowledge, this is the first time parallel skeletons have been exploited this way in Erlang; and,
2. we demonstrate the effectiveness of our skeletons on a set of synthetic benchmarks, therefore demonstrating the parallel capabilities of Erlang;

## 2 Erlang

Erlang is a strict, impure, functional programming language with support for *first-class* concurrency. This concurrency model allows the programmer to be explicit about processes and communication, but implicit about placement and synchronisation. Erlang supports a *lightweight* threading model, where processes model small units of computation (tasks) that are executed on a capability. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. Erlang typically has three primitives for handling concurrency:

- `spawn()`, allowing new functions to execute in a lightweight Erlang process;
- `!`, allow messages to be explicitly sent from one Erlang process to another; and,
- `receive`, to allow messages to be received in another process queue.

Furthermore, Erlang also supports fault tolerance, by allowing groups of processes to be *supervised*, and new instances of processes can be spawned in the case failure. Although Erlang supports concurrency, there has been little research into how Erlang can be used to effectively support *deterministic* parallelism.

## 3 Skeletons in Erlang

The design of the `skel` library has been based upon the design of FastFlow [1], a parallel programming framework for multi-core platforms written in C++, but with significant changes to take advantage of features provided by the Erlang language and VM.

### 3.1 Skeletons

So far we have only implemented seven core "primitive" skeletons so far, however they can be combined to make more complex skeletons. They are:

**Seq** a skeleton to encapsulate an indivisible portion of sequential code.
**Pipe** the functional composition of multiple skeletons. In `skel`, these are implicit.
**Farm** a skeleton that schedules inputs onto replicas of a pipeline, and then collects the results back into a single stream.
**Map** a skeleton that can decompose each item, put each decomposed part through its own replica of a pipeline, and then recompose the results back into a single item again.
**Reduce** a skeleton that applies a treefold, in parallel, over each decomposed item.
**Feedback** a skeleton that can send independent items back through a skeleton.
**Ord** a skeleton to restore order to a stream of items[4].

The skeletons we have implemented here are a set of foundation skeletons, encapsulating common functional patterns. By investigating foundational skeletons, we can explore the parallel capabilities of Erlang while providing a strong framework that allows for more complex skeletons to be implemented in the future.

Most of these skeletons need to have one or more skeletons nested inside them to work correctly, for instance a **Farm** skeleton requires at least a single **Seq** skeleton to be nested within it or there would be no point in having the **Farm** skeleton in your pipeline. When more than one skeleton is specified to be nested, those skeletons will be assembled into a pipeline that is nested inside the other skeleton. The only two skeletons that cannot have other skeletons nested inside them are **Seq**, which is designed to encapsulate sequential code, and **Reduce**, which applies a binary function to each item.

The `skel` API that is presented to programmers is a single function:

```
skel:run(Pipeline, ItemSource).
```

- `Pipeline` an ordered list of skeletons that each item on the stream is to be processed through.
- `ItemSource` either a list of stream items, or is a module with functions that can supply stream items.

The process will then receive a message in response with the contents `{sink_results, ResultItems}` where `ResultItems` is a list of the results of sending the items from `ItemSource` through `Pipeline`. In the code examples, this is denoted by `% -> {sink_results, ItemSource}` (`sink_results` just tells the receiver that this message contains the results from the sink).

[4] The implementation of this skeleton is not described, though it is present for applications where the order of items is important.

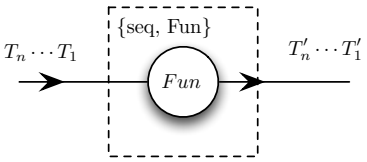


**Fig. 1.** The **Seq** Skeleton

In the following schematics, circles should be read as explicit Erlang processes – if they are white then they have some user-defined behaviour in them, as specified by the identifier inside them, and if they are black they only have some system-level logic happening in them. Rectangles with solid edges denote an internal pipeline, and the outer dotted rectangle denotes which processes are contained in that particular skeleton. Data messages travel along the lines with arrows, though as mentioned, they are not explicit channel objects.

**Pipe** Pipe is the only skeleton that does not have any of its own processes, and does not explicitly exist. The basic building block of our library is a pipeline, so skeletons are defined in terms of pipelines with 1 or more stages. We then use algorithms (termed "assembly algorithms") to turn this skeleton declaration into a system that can run computations on a stream of items. Our implementation currently contains two assembly algorithms: a parallel one (the default) and a sequential one. Functionally, both assemblers will give you exactly the same results, however the assembled process structures (and hence the parallelism degrees) are completely different.

The parallel assembly algorithm is one that maps over the list of skeleton declarations, using the details in each declaration to create a list of what we term a "maker functions". We then do a right fold over this list, so that we start each one in reverse pipeline order, starting with the process id of a sink (the end of the pipeline). The "maker functions" take the process id of the receiving part of the next skeleton, start up that skeleton's requisite processes, and then returns the process id of the receiving part of that skeleton, hence the fold right.

The sequential assembly algorithm makes each skeleton declaration into a single function, then composes them together into an entirely sequential version of the pipeline. On the face of it, this may not seem sensible, however it is useful for benchmarks and for working out, at a functional level, what each skeleton does. All future focus is on the parallel versions of each skeleton.

**Seq** Seq is the most basic of all the skeletons. It consists of a single process that applies a function, `Fun`, to any data messages it receives, before sending the results on to the next skeleton. Should that process receive an end-of-stream message it will exit immediately. A schematic of this skeleton is shown in Figure 1, and an example of its operation can be seen in Figure 2.

```
skel:run([{seq, fun (X) -> X+1 end}],
         [1,2,3,4,5,6]).
% -> {sink_results,[2,3,4,5,6]}
```

**Fig. 2.** An example of the **Seq** Skeleton

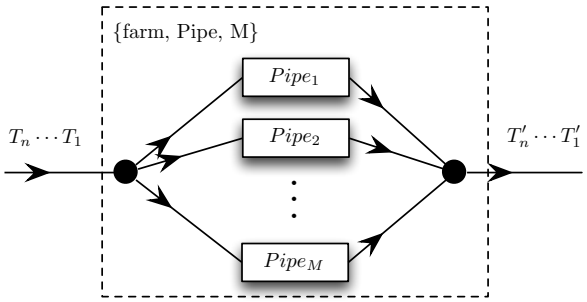


**Fig. 3.** The **Farm** Skeleton

**Farm** In a Farm skeleton, an emitter forwards inputs into one of `M` replicas of the pipeline, `Pipe`, which then forwards results onto a collector and then the next skeleton. A schematic of this skeleton is shown in Figure 3, and an example of its operation is shown in Figure 4.

The emitter process is very simple. When it receives a data message, it forwards that message on to any single one of the pipeline replicas. When it receives an end-of-stream message, it forwards this message to every single one of the pipeline replicas. At the moment the scheduling algorithm is round-robin, but we expect to add other algorithms in the future.

The collector process is also simple. When it receives a data message (from any of the pipeline replicas), it forwards the message onto the next skeleton. The collector waits for `M` end-of-stream messages before exiting, to make sure that it has received all messages from each of the pipeline replicas.

**Map** In a Map skeleton, each item is decomposed by the `Decomp` function into a number, $m_i$, of parts, then each of these is forwarded into one of $m_i$ replicas of the pipeline, `Pipe`. After the pipes, each part is then forwarded to a recomposer, which combines all the parts back into a single item using the `Recomp` function. A schematic is shown in Figure 5, and an example of its operation is shown in Figure 6.

The `Decomp` function is for disassembling a collection-like item into a list of its constituent parts, and the `Recomp` function is for turning the list of constituent parts back into a collection-like item. If you're dealing with lists, you

```
skel:run([{farm, [{seq, fun(X)-> X+1 end}],
                  3}],
         [1,2,3,4,5,6]).
% -> {sink_results,[2,5,3,6,4,7]}
```

**Fig. 4.** An example of the **Farm** Skeleton

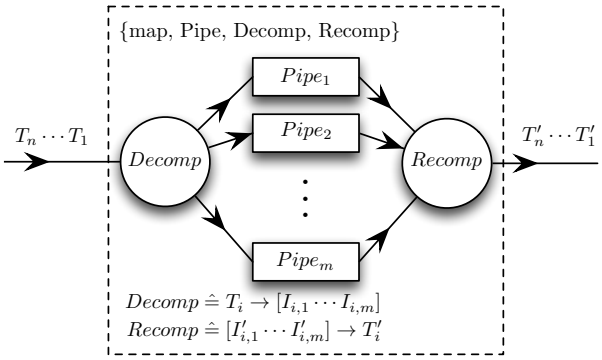


**Fig. 5.** The **Map** Skeleton

can just use an identity function, and helpfully Erlang has `tuple_to_list/1` and `list_to_tuple/1` if you're moving around tuples, like we are in Figure 6.

The decompose process is less trivial than a normal emitter. It waits for items and then splits them into many parts. After each data message is split, each part is labelled with a unique tag that relates to the input item, the index of that part in the collection of parts, and the total count of parts for that item (this is the reason we have the stack in each data message). Each part is then sent through a replica of the pipeline.

As we cannot make any assertions about the number of parts that the `Decomp` function will produce, the decompose process also has the ability to start more replicas of the pipeline if it does not have enough for all the parts of an input.

The recompose process is also quite complicated. It waits for each part, then puts it into a store (keyed by item unique tag and part index), along with the data about how many parts it has so far received, and how many it is expecting. When it has received as many as it is expecting, it calls the `Recomp` function with a list of the inputs that it has received (therefore preserving the order the parts were in when they came out the `Decomp` function), which produces a single item again, which it then forwards to the next skeleton. The process also stores the highest number of parts that it has received for any input, in order to know how many end-of-stream messages to wait for before exiting.

```
skel:run([{map, [{seq, fun(X)-> X+1 end}],
                fun erlang:tuple_to_list/1,
                fun erlang:list_to_tuple/1}],
         [{1,2},{3,4}]).
% -> {sink_results,[{2,3},{4,5}]}
```

**Fig. 6.** An example of the **Farm** Skeleton

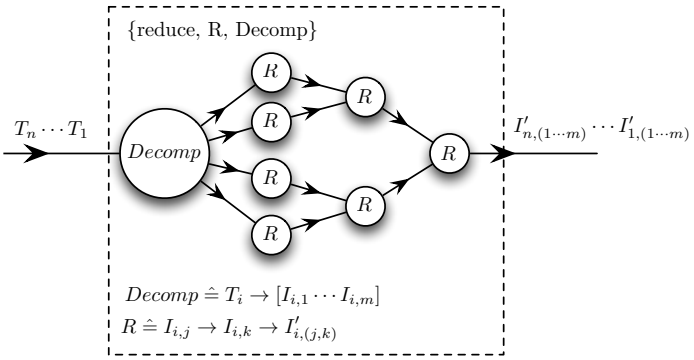


**Fig. 7.** The **Reduce** Skeleton

**Reduce** The Reduce skeleton executes a parallel treefold over the decomposed parts of each input. Inputs are first decomposed into several parts, and then they're submitted into a binary tree of reduce processes. The reduce processes compute their result with both parts of the item that they receive, then forward the result on to the next reduce process. A schematic is shown in Figure 7, and an example of its operation is shown in Figure 8.

The decompose process waits for inputs, then splits them into $m_i$ parts. After each data message is split, each part is labelled with a unique tag that relates to the input item, and a number that indicates how many reduce steps it will go through (this is calculated as $\lceil log_2(m_i) \rceil$). The parts are distributed to the correct level of the reduce process tree corresponding to $m_i$ parts. Two parts are given to each reducer, and then any reducers still waiting for a part are given a unit input. The unit input is a system-level message that would act as the unit value for the computation (this is explained further, later). When the decomposer receives an end-of-stream message, it forwards 2 end-of-stream messages to each reducer at the widest level of the tree.

Again, we cannot make any assertions about the number of parts the `Decomp` function produces, so process instantiation is dynamic.

The reduce processes wait for parts or unit values. For the first part (or unit value) of an item that they get, they store it, and then wait for another. When they receive the second part (or unit value), they use a fairly simple algorithm to

```
skel:run([{reduce, fun(X,Y) -> X + Y end,
                   fun erlang:tuple_to_list/1}],
         [{1,2,3,4,5,6},{7,8,9,10,11,12}]).
% -> {sink_results,[21,57]}
```

**Fig. 8.** An example of the **Reduce** Skeleton

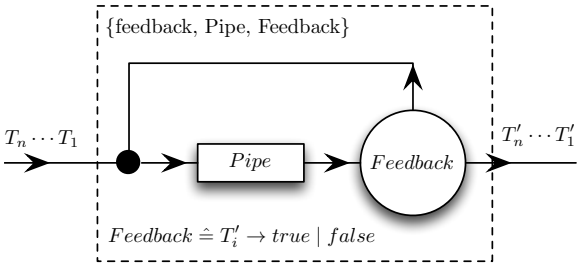


**Fig. 9.** The **Feedback** Skeleton

work out what to forward. If they received two unit values, they forward the unit value. If they received a part and a unit value, they forward the part, and only if they received two parts do they combine them using the reduce function, before forwarding on the result of the computation. Before forwarding an input, the reducer decrements the count of steps that the part has to go through. When this count gets to zero, the reduce label on the data message is taken off, to preserve the invariant that the label stack is the same on leaving a skeleton as it is when it entered it. When a reducer receives two end-of-stream messages, it forwards a single end-of-stream message to the next reducer.

**Feedback** The Feedback skeleton sends items through a pipeline, `Pipe`, and then checks a predicate, `Feedback`, to find out whether it should send that input through the pipeline again or forward it to the next skeleton. A schematic is shown in Figure 9, and an example is shown in Figure 10.

The only complication in this skeleton comes in the race condition between an end-of-stream message coming in, and items that are going through the feedback loop. Because we cannot prioritise one "stream" over another in Erlang (which would let us prioritise the feedback queue, and just stop receiving from the previous skeleton), we instead maintain two counters of items in two parts of the skeleton. The counters are stored in a separate process, due to the lack of any shared state in Erlang, which also allows us to make any operations on them atomic, and allows the receiver to subscribe to updates anyone makes to the counters. The first counter keeps a count of items in the pipeline, and the second keeps a count of items in the pipeline. Once an end-of-stream message is received, the receiver process (the black circle in Figure 9) continues receiving,

```
skel:run([{feedback, [{seq, fun(X) -> X+1 end}],
                     fun(X) -> X < 5 end}],
         [1,2,3,4,5,6,7,8,9,10]).
% -> {sink_results,[5,6,7,8,9,10,11,5,5,5]}
```

**Fig. 10.** An example of the **Feedback** Skeleton

but also waits to be told when both counters get to zero. When both counters reach zero, the end-of-stream message can be forwarded without fear of race conditions.

## 4 Related Work

Since the nineties, the "skeletons" research community has been working on high-level languages and methods for parallel programming [4, 5, 3, 6, 2, 9]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterized to execute problem-specific code. Skeletons do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, map and reduction.

Early proposals of pattern-based parallel programming frameworks have been mainly focused on distributed memory platforms, such as clusters of workstations and grids [13, 11]. All these skeleton frameworks provide several parallel patterns covering mostly task and data parallelism. These patterns can usually nested to model more complex parallelism exploitation patterns according to the constraints imposed by the specific programming framework. Recently skeletons gained renewed popularity with the arrival of multi-core platforms, the consequent diffusion of parallel programming frameworks, and their adoption in some programming frameworks, such as FastFlow [1], Intel Threading Building Block (TBB) [10] and to a limited extent the Microsoft Task Parallel Library [12]. Google MapReduce [7] brings to the mainstream of out-of-core data processing the map-reduce paradigm. The main features of these frameworks, as well as many other experimental ones, are surveyed in [8].

## 5 Conclusions and Future Work

## Acknowledgements

This work has been supported by the European Union Framework 7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", `http://www.paraphrase-ict.eu`.

## References

1. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: High-level and Efficient Streaming on Multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2012.
2. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. $P^3L$: A Structured High Level Programming Language and its Structured Support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.
3. G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 243–252. IEEE Computer Society Press, 1996.
4. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.
5. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.
6. J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for Structured Parallel Composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*, 1995.
7. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.
8. H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.
9. M. Hamdan, P. King, and G. Michaelson. A Scheme for Nesting Algorithmic Skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.
10. Intel Corp. *Threading Building Blocks*, 2011.
11. H. Kuchen. A Skeleton Library. In B. Monien and R. Feldman, editors, *Proc. of 8th Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 620–629, Paderborn, Germany, Aug. 2002. Springer.
12. D. Leijen and J. Hall. Optimize Managed Code for Multi-Core Machines. *MSDN Magazine*, Oct. 2007.
13. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.