University of St Andrews | CS4204 | School of Computer Science KY16 9SX

Practical 2 - Parallel Skeletons in Erlang

Student ID: [redacted]

April 25, 2025

1 Introduction

Growing computational demands have made single-core systems inadequate for many applications. Parallel programming addresses this by leveraging multicore architectures to distribute workloads [1]. Erlang provides a robust model for parallelism, featuring a functional core, lightweight processes, and message-passing concurrency. It offers fine-grained control over process creation and communication, while abstracting scheduling and resource management through high-level constructs [2].

This project explores algorithmic skeletons—reusable high-level patterns—for parallelising matrix multiplication in Erlang. Skeletons such as *farm* and *pipeline* encapsulate common parallel paradigms and expose them via composable interfaces [3]. Unlike low-level concurrency techniques, such as manual process spawning and message-passing, skeletons abstract thread management and synchronisation, simplifying implementation without compromising performance.

Matrix computations are central to scientific simulation, machine learning, and computer graphics. Effective parallelisation must address load balancing, granularity, and communication overhead [4]. This work implements *farm* and *pipeline* skeletons and evaluates their performance on matrix multiplication against the *Skel*[1] library to assess the efficiency of the parallelisation.

2 Related Work

Skeleton-based parallelism is supported across various frameworks. FastFlow [3] targets shared-memory multicore systems using lock-free coordination, while Muesli [5], a C++ library, integrates task and data-parallel skeletons across MPI, OpenMP, and CUDA. In Erlang, the *Skel* library offers composable skeletons like *farm* and *pipeline*, built on its lightweight actor-based model.

In contrast to manually managing processes and information flow, skeletons provide reusable abstractions that simplify reasoning about parallel structure and reduce implementation overhead. This work builds on these principles by implementing custom *farm* and *pipeline* skeletons in Erlang, benchmarking their performance on matrix multiplication tasks of varying sizes and core counts. While *Skel* has already achieved a similar solution, this work has achived comparable results, as shown in the remainder of this paper.

3 Implementation and Experimental Setup

The implementation of farm and pipeline skeletons in Erlang was evaluated on a matrix multiplication example, where the processing load was chunked to allow better load balancing. All experiments were conducted on a Linux system with an 11th Gen Intel® Core™ i5-11400 CPU @ 2.60 GHz (12 cores) and 30 GiB of RAM. Erlang/OTP 26 with NIF 2.17 was installed. The benchmark consisted of parallel matrix multiplication using row-wise operations defined in `matmul.erl` on different matrices. For timing the functions, `timer:tc/3` is used to time the computation

[1] https://github.com/ParaPhrase/skel

(wall-clock time) of the actual skeletons, as well as the sequential function as a baseline, listed in Table 2.

The skeletons we designed to be modular. The farm skeleton implemented in `farm:start_farm/2` takes the worker function and a list of tasks, the the length of the list denotes the number of worker processes will be started. The pipeline (`pipeline:start_pipe/2`) takes the stages, a list of functions and an input, that matches the function signature. The stage output does need to match the signature of the next stage input. The worker function used to employ the skeletons was `multiply/2`, as it works as long as $Cols(A) == Rows(B)$, so the chunks can be easily passed to it, as well as `dot_product/2` for utilising the farm within each pipeline state in the hybrid approach.

Matrix Size	50, 100, 150, 200, 300, 450, 600
Number of Chunks	2, 3, 5, 6, 10, 12
Number of Cores	2, 3, 4, 5, 6, 10, 12
Patterns	farm, pipe, pipe(farm), farm(pipe)
Trials	20 iterations per combination

Table 1: Configuration Arrays and Skeletons

The workloads are listed in Tab. 1, and the first input matrix A is split into a specific number of segments: Let $A \in \mathbb{R}^{n \times n}$ be a matrix. A is partitioned row-wise into c submatrices $A_i \in \mathbb{R}^{\frac{n}{c} \times n}$, for $i = 1, \ldots, c$, where each A_i consists of $\frac{n}{c}$ consecutive rows of A, assuming c divides n evenly. If the row-dimension of the matrix is not divisible by the number of chunks specified, the last chunk may be smaller than the previous ones, which introduces some asymmetry.

Tested patterns include using only the farm skeleton on different configurations, as well as only the pipeline, with one stage per chunk. Other configurations tested were nesting the farm into the pipeline stages, by passing `dot_product/2` as the worker function to the farm created in each pipeline stage (`matmul:start_hybrid1/2`), as well as using the pipeline inside each worker process in the farm (`matmul:start_hybrid2/2`), and Listing 1 shows an example output of the latter, where the 2 chunks are numbered, and inner lists represent the rows.

```
1> matmul:start_hybrid2(4,2).
[{1,[[[10,20,30,40],[10,20,30,40]]]},
 {2,[[[10,20,30,40],[10,20,30,40]]]}]
```

Listing 1: Example output of using the pipe(farm) hybrid on a 4×4 matrix and 2 chunks.

4 Evaluation

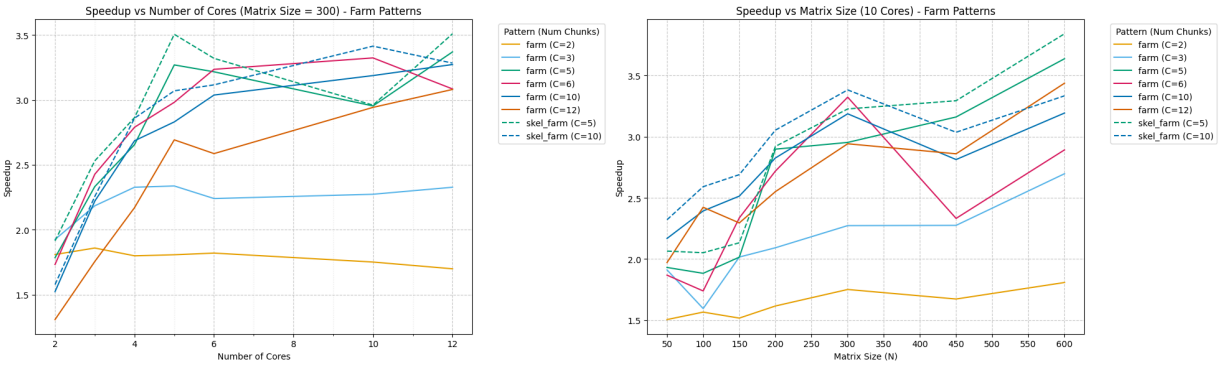
Benchmarking was performed across varying matrix sizes, chunk counts, and core configurations. Each setup was executed 20 times, yielding a coefficient of variation (CV) below 10% for larger matrices. Table 2 shows a consistent decrease in the CV for the farm skeleton as matrix size increases, supporting the notion that greater task granularity improves timing stability. Memory usage also rose with chunk count, as the second matrix is transmitted to each worker. Fig. 1 presents the result of the *farm* skeleton, as hybrid implementations showed a 4.20% performance drop for the *farm(pipe)* pattern relative to the farm skeleton, and more than 100% degradation for *pipe(farm)* compared to the pipeline. These findings are further discussed in the *Limitations* section. Speed-ups for the *pipe* are shown in Tab. 2 and stay fairly constant.

Speed-ups and Granularity To analyse the effect of the core number on speed-up, a matrix size of 300 was selected, as it had a CV close to 5%, which allowed reliable results and it is divisible by all chunk and core numbers. Fig. 1a shows that the relation of chunks and available cores is important to gain speed-ups. The local maximum for $C = 5$ at 5 cores underscores this, as well as the global maximum for 12 chunks at equal core numbers. The latter shows

matrix_size	cv_percent		speed-up		sequential_time (ms)	memory (mb)	
	Farm	Pipeline	Farm	Pipeline		Farm	Pipeline
50	35.436	31.039	1.751	0.680	3.727	0.49	17.92
100	32.335	14.056	1.879	0.879	15.790	2.89	141.49
150	14.535	5.600	1.938	0.804	23.274	7.73	540.42
200	9.797	3.698	2.144	0.806	40.729	10.66	1154.31
300	5.483	1.939	2.443	0.907	100.785	17.47	4466.66
450	3.944	8.375	2.149	0.849	333.036	29.82	7464.22
600	2.109	5.482	2.500	0.884	717.799	66.11	21646.34

Table 2: Summary Statistics Comparing Farm and Pipeline Skeletons

interesting behaviour, as it has a local maximum for 5 cores, suggesting the Erlang scheduler working efficiently to distribute the workload. In Fig. 1b, we see that the speed-up steadily increases with the matrix size, and this could be explained by the low memory usage listed in Tab. 2 being far from hitting any hardware limitations. The results for the lower matrix sizes have to be taken with a grain of salt, as Tab. 2 highlights, that the CV is higher than the general trend observed, and we saw many outliers in the data, as will be discussed when talking about the issues later in the paper.



(a) Speed-up over core numbers.

(b) Speed-up over matrix sizes.

Figure 1: Speedup of the farm skeleton

Limitations The pipeline implementation did not outperform the sequential baseline, which can be partly attributed to increased memory usage shown in Tab. 2 and potential inefficiencies in data handling. Notably, the `pipe(farm)` pattern led to a further performance degradation of over 100% relative to the standalone pipeline, averaged across cores and chunk configurations. In no tested configuration did either hybrid skeleton achieve performance comparable to their non-nested counterparts—even when using large matrices that led to computing chunks of 50 rows. This drop in performance is likely due to a combination of factors. Nesting farms within pipeline stages increases message-passing overhead and introduces additional process scheduling complexity. Each stage in `pipe(farm)` spawns multiple worker processes, potentially over saturating the Erlang scheduler and leading to contention or idle time across cores. Moreover, misalignment between chunk counts and core availability may result in underutilised or overloaded schedulers. Tab. 2 highlights the pipeline's disproportionately high memory usage, which increases with core count. This trend suggests a possible memory leak or inefficient data handling in the implementation. The pipeline passes `Input = {MatA, MatB, Result}` through each stage, which may result in unnecessary data duplication. Further work is required to ensure that large data structures are passed efficiently and that intermediate results are not inadvertently retained across stages.

Outliers, where the CV is more than 2 standard deviations from the mean are listed in Tab. 3, and this explains partly why we see so much variety in the speed-ups for low matrix sizes and

outliers especially occur in high core numbers, as the fine task granularity leads to large deviations. Increasing the number of trials to 30 and 50 did not lead to significant improvements. As the workload gets bigger, the measurements and reproducibility of the results increased significantly.

matrix_size	chunk_size	core_count	cv_percent
50	12	12	186.352
50	10	2	163.747
100	6	10	71.343
200	10	10	47.010

Table 3: Notable High CV Benchmark Outliers

Comparison to Skel The comparison between the *farm* and *Skel* implementations reveals consistent performance characteristics. To compare, `skel:do/2` was used to implement the same stages and workers as were used in Sec. 4 The library demonstrates a moderate performance advantage, with execution times averaging around 1.1x faster than the *farm* implementation. This suggests that *Skel* maintains its advantage. Both implementations exhibit similar stability in their performance, as indicated by comparable CV values. The speed-up ratios between the two implementations remain relatively stable across different configurations, indicating that the performance characteristics are consistent and predictable. Fig. 1 shows dashed lines for chunk sizes 5 and 10 that illustrate the general trends, showing the performance advantage of the *Skel* implementation.

5 Conclusion

This work demonstrated the use of algorithmic skeletons—specifically farm and pipeline patterns—to parallelise matrix multiplication in Erlang. The custom implementations showed that farm-based parallelism can yield significant speed-ups, especially as problem size increases, due to better granularity and lower communication overhead. While pipeline and hybrid skeletons introduced additional memory costs and underperformed, the project highlighted key design trade-offs in skeleton composition. Comparing results to the *Skel* library confirmed the viability of custom skeletons for scalable parallel computation in Erlang, offering a practical approach for exploring concurrency in the domain of matrix multiplication.

References

[1] F. Wrede and S. Ernsting, "Simultaneous CPU–GPU execution of data parallel algorithmic skeletons," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 42–61, Feb. 1, 2018.

[2] F. Cesarini and S. Thompson, *Erlang programming*, 1st ed. Beijing ; Cambridge [Mass.]: O'Reilly, 2009, 470 pp.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, *et al.*, "Fastflow: High-level and efficient streaming on multicore," in *Programming multi-core and many-core computing systems*, S. Pllana and F. Xhafa, Eds., 1st ed., Wiley, Jan. 24, 2017, pp. 261–280.

[4] A. M. Schmidtobreick, "Parallel asynchronous matrix multiplication for a distributed pipelined neural network," 2017, Publisher: Heidelberg University Library.

[5] H. Kuchen, "Parallel programming with algorithmic skeletons," in *The Art of Structuring: Bridging the Gap Between Information Systems Research and Practice*, K. Bergener, M. Räckers, and A. Stein, Eds., Cham: Springer International Publishing, 2019, pp. 527–536.