2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress

The Weekend Warrior: How to build a genomic supercomputer in your spare time using Streams and Actors in Scala

Brendan Lawlor*† and Paul Walsh†

*Dept. of Computing, Cork Institute of Technology, Cork, Ireland
Email: firstname.lastname@mycit.ie
†NSilico Life Sciences Ltd., Cork, Ireland
Email: firstname.lastname@nsilico.com

***Abstract*—The cost of developing parallelised software is a significant bottleneck in the implementation of bioinformatics pipelines. This paper identifies the use of low-level threading abstractions as one cause for these elevated costs, and proposes alternatives which offer better value. We identify a core set of higher abstractions to accomplish low-cost multi-core and multi-server parallelization - Streams, Actors and the Scala language (StAcS). We present the performance results of a robust, parallelizable and scalable implementation of the Smith-Waterman algorithm developed using the StAcS approach at development costs inferior to equivalent systems written using current standard techniques.**

I. Introduction

The barriers to a wider and deeper use of our growing reserves of genomic data are as much economic as technological. Genomic data has become relatively cheap to create and is getting cheaper and more abundant. Multi-core processors are reducing processing speed limits for parallel software. Price barriers to deploying software on large-scale grid and distributed systems are lower thanks to cloud computing. But while the software to take advantage of these trends must be capable of extracting every last clock-cycle from multi-core processors distributed over many nodes, such systems have historically been expensive to develop, maintain and operate, and will remain so unless bioinformatics practitioners adopt new software architectures.

One important reason for the high cost of parallelization is that implementations typically use low levels of abstraction as exemplified by OpenMP and MPI. However such abstractions are widely acknowledged as time-consuming and error prone - two factors which increase development and maintenance costs [1]. However, commercial software engineering, faced with the same economic barriers, has begun to change its approach to parallelization. High-level programming techniques developed decades ago have been found to have enormous value in solving today's problems. We describe a set of these techniques in this paper, and demonstrate how they can be used in the particular setting of Smith-Waterman sequence alignment to achieve high levels of parallelization at low cost. The natural implication is that bioinformatics practitioners should learn from industry and adopt these technologies.

A. Lower versus Higher Abstractions

Software developers implement parallelization across multiple cores using multi-threaded code. Scientific software generally accomplishes this by means of low-level abstractions exemplified by OpenMP. What these systems have in common is the use of synchronization primitives to protect blocks of shared data from concurrent access. The problem with this approach is that it sharply increases the complexity and cost of code [2], obfuscates the underlying algorithms [3] and introduces an entire class of new problems like race conditions and deadlock/livelock, which are expensive to find and fix. The overuse of locks can furthermore end up serializing code that might otherwise be run in parallel [4].

Triggered by the impending demise of Moore's Law, researchers have sought more productive alternatives to thread-and-lock programming - language abstractions that can be mapped more efficiently onto parallel hardware, and which disentangle the logic of parallelization with the logic of the algorithm under development [4]–[7].

In this paper we will present a set of abstractions that can be employed to achieve multi-core *and* multi-server parallelization at a low cost of development, maintenance and deployment. Our goal is to demonstrate that despite the widely-held concern that higher level abstractions can reduce performance by removing opportunities for optimization, in practice this does not have to be the case, especially in so-called embarrassingly parallel contexts. On the contrary, when implementing large-scale parallelization across cores and servers, the kind of bottlenecks that arise (such as hardware-based I/O limitations) are best dealt with at similarly high levels of abstraction. As we will see in the next section, StAcS not only accomplishes multi-threading but thanks to the location-independence of *actors* it also acts as a suitable abstraction for creating clusters of cooperating network nodes, providing parallelization across multiple servers.

978-1-5090-2771-2/16 $31.00 © 2016 IEEE
DOI 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.32

B. Streams and Actors in Scala (StAcS)

So how do we avoid the drawbacks of low-level thread-and-lock abstractions, and at the same time capitalize on the promise of inter-sequence parallelization? For the purposes of this research, we have selected what we consider to be a core set of tools to accomplish low-cost, multi-core and multi-server parallelization: Streams, Actors and the Scala language. For brevity, we will refer to this combination as StAcS. In this section we describe these three elements, explaining how each one addresses the needs and concerns of parallel systems conveyed in the previous section. We then outline a proof-of-concept implementation of Smith-Waterman which demonstrates that it is possible to create robust, parallel and scalable systems using the StAcS approach with lower costs than equivalent systems written using more traditional techniques. Note that the same approach could be applied to any alternative alignment algorithm that takes two sequences and returns a score.

1) Scala: By coding in functional programming languages (which transform immutable data rather than manipulate shared mutable data), and using intuitive abstractions such as Actors to achieve multi-threading and server-clustering, it becomes easier - and therefore quicker and cheaper - to develop robust parallel systems. Such systems are not prone to the errors normally associated with multi-threading mentioned above. Scala in particular offers some advantages over other functional languages. Firstly, it runs on the JVM, which automatically means a reduction in code complexity due to the JVMs Memory Model. It also makes deployment costs cheaper by running on any commodity hardware platform, allowing owners a wider choice of platforms. Secondly, Scala is not only a functional language, it also Object Oriented. This makes it more accessible to programmers familiar with imperative OO languages such as C++ and Java, reducing the costs of adoption. Thirdly, Scala comes with a rich set of productivity tools including highly sophisticated Integrated Development Environments like *Eclipse* and *IntelliJ*.

2) Actors: As described by Karmina et al. [6], the actor abstraction is a programming model that is *inherently* parallel. The model is made up of actors and messages, where messages are passed between actors asynchronously, and each actor processes its message to completion in a single-threaded fashion. Actor systems do not suffer from the obfuscation effect of the thread-and-lock approach, but instead enable a compositional design which simplifies reasoning. As Agha et al. [8] have found, actor systems easily express a wide range of computational paradigms, and provide a natural extension of programming into concurrent (parallel) systems. This naturalness and ease of expression means that programs to solve complex problems do not add even *more* complexity of their own. This direct relationship of the code to the problem domain also makes it easier to optimize algorithms based on knowledge of that domain.

3) Reactive Streams: The Achilles heel in actor-based systems is the mailbox: Every actor receives messages into its own mailbox and processes those messages when it is allocated resources. This can lead to mailbox overflow if the amount of data arriving into a system is greater than that system's capacity to process it. What is required is a simple and intuitive pattern to buffer incoming messages and signal data producers to wait before sending any more messages. This signal is known as 'back-pressure' and has been implemented as part of networking protocols for decades. Generating back-pressure is a low-level activity which, similarly to the thread-and-lock pattern, will obfuscate code if implemented directly. Reactive Streams [9] offer a way of implementing back-pressure in a transparent way. They work well with actor systems by avoiding the message buffer overflow problem in a succinct and elegant way.

In addition to solving the message overflow problem, reactive streams allow systems to become more robust and to scale *down* as well as up. By transparently propagating 'back-pressure' from consumers to producers of data, they allow systems to react gracefully and intelligently to limitations in processing power (either temporary or systemic) without failing.

4) Smith-Waterman using StAcS: With specific reference to sequence alignment, parallelization efforts can be divided into two kinds: *intra-sequence* and *inter-sequence*. The Smith-Waterman alignment algorithm is used to compare large numbers of sequences, but *only two at a time*. Each such pairwise comparison requires a matrix with one sequence as the row index and the other sequence as the column index. Each value in the matrix is calculated by finding the maximum of 4 possible integer values. The value in a given cell is based on the previously calculated values in the 3 neighbouring cells above and to the left. An *intra-sequence* approach to parallelization would exploit the fact that it is theoretically possible to calculate cell values in parallel along the minor diagonal of the matrix. *Inter-sequence* parallelization, by contrast, makes use of the fact that no data dependencies exist between individual alignments of any given pair and simply seeks to run as many of them in parallel as available resources will allow. The StAcS approach lends itself more naturally to inter-sequence parallelization, but does not exclude the use of intra-sequence optimizations, as will be explained below.

II. Methods

The purpose of this paper is to demonstrate that using high level abstractions to achieve parallelization is a functioning alternative to the commonly used low-level abstractions described above, and in fact is preferable due to its relatively modest costs. In order to test that hypothesis we developed a proof of concept using the Streams and Actors described in the previous section, verified their efficacy, and measured the related costs.

We verified the ability of the code to parallelize efficiently across the 4 available cores of a single network node by running a Smith-Waterman algorithm library without Actors or Streams and measuring the speedup factor when the same library was run on the same node using StAcS. We then verified that parallelization across a cluster of nodes was

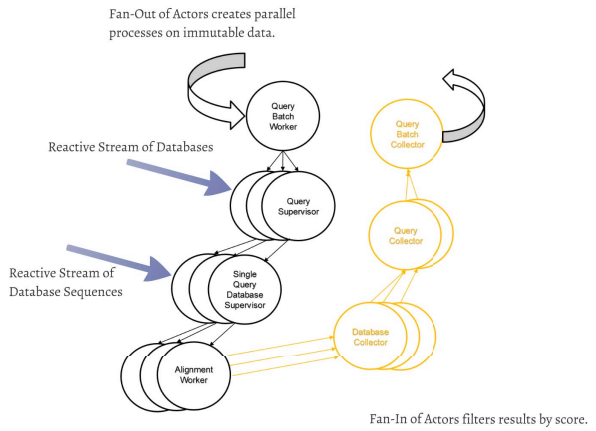

Fig. 1. Actors on the Worker Node

linearly scalable by measuring the time to completion as more nodes were added to that cluster. For practical reasons, rather than maintaining a fixed number of database sequences and expecting a reduced time to completion, we increased the number of database sequences by 10,000 for every new node and expected a constant time to completion.

Costs are difficult to measure directly, so we used the proxy measurements of development time, lines of code and cloud computation expenses to represent development, maintenance and operational costs respectively.

A. *Proof of Concept*

What follows is a necessarily simplified description of the proof of concept[1].

The system runs across a number of network nodes, each executing the same 'worker' program. One lightweight 'master' node distributes tasks to the workers, and receives the results. (All nodes participate in a cluster which is managed by the same framework which implements both the actor model and the reactive streams (Akka). The actors are unaware of their physical location as they communicate with each other.) The actors operating within a single worker are shown in figure 1.

With respect to figure 1:

- Every circle in this figure represents an actor instance and the small arrows between actors represent the passing of messages between them.
- The Alignment Worker at the bottom of the hierarchy is where the pairwise alignment of the query sequence with a single database sequence is done. The alignment is handled by calling an external library which uses SIMD techniques (and so benefits from intra-sequence parallelization).
- The Query Batch Worker at the top of the hierarchy is the point of entry to each worker node and is responsible for splitting a number of incoming queries (a batch) into individual queries to be processed in parallel.
- Every actor level in the hierarchy between the entry point and the Alignment Worker breaks the work up further into parallel tasks; first by database (a group of known sequences) and then by individual database sequence.
- The speed at which this proliferation of parallel tasks is performed is managed by two types of incoming reactive streams: a stream of database names and a stream of database contents.
- The Alignment Workers, which are doing the CPU-intense task of performing the Smith-Waterman alignment, create the necessary 'back-pressure' which is transparently propagated back up the hierarchy, throttling the rate at which new actors are created.

Note that there is a reverse hierarchy on the right, which gathers in all results and performs local comparisons of the scores, passing up only the highest score from each database, and then each query. This allows for parallelization of the comparison, and also dramatically reduces the number of messages exiting the system. The master node receives only the 'winning' database sequence for each query.

The Smith-Waterman library we used was developed by Zhao et al. [10] and is a single-threaded, highly-optimized SIMD library available also to the JVM through a JNI[2] library provided by that team.

A useful image to employ in order to understand the interplay of Actors and Reactive Streams in this implementation is that of a Formula 1 engine with intelligent fuel injection. The Actors spin as quickly as the processor speed and core numbers will allow. The Reactive Streams inject exactly the right amount of data - not so little as to stall the engine, and not so much as to flood it. This allows the engine to rev up and down as processing resources (nodes) are added and taken away.

III. Results

A. *Parallelization Over Multiple Cores*

We measured the average time to completion over 10 runs of the Smith-Waterman alignment as it looped through 10,000 database sequences which were read from a number of local files, on a Linux server with an Intel i7 (quad-core) processor. We then measured the average time to completion for the StAcS implementation on the same machine, reading the same number of sequences from the same files. The results are shown in figure 2.

Our expectation is that on a quad-core processor, a multi-threaded implementation should be four times faster than single-threaded code. In fact the observed speedup using StAcS was 4.87, slightly more than might be expected based simply on the number of cores available. This demonstrates that the use of high-level abstractions did not hinder the expected speedup - quite the opposite. We believe the extra speedup was due either to a more efficient I/O due to the buffering inherent in Reactive Streams, or to the influence of

[1]Our implementation made extensive use of the Akka framework which provides both Actors and Reactive Streams through Scala.

[2]Java Native Interface: a mechanism that allows code on the JVM to invoke native code

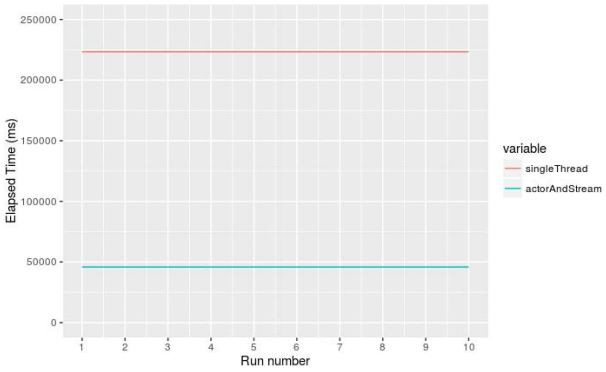

Fig. 2. Single thread vs actors on single node.

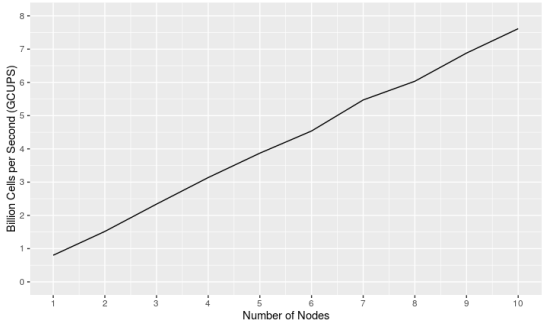

Fig. 3. Linear scalability over multiple nodes on cloud.

the i7 processor's hyper-threading technology, but this will need to be investigated further. In any case, the results tell us that the use of Actors and Streams on Scala do not present any measurable overhead in their ability to provide parallelization, despite their high levels of abstraction, and in fact may make it easier to achieve some efficiency benefits. We discuss the significance of these results below.

B. Parallelization Over Multiple Nodes

Using a commercial cloud computing service, we deployed the *worker* agents described above on increasing numbers of Linux nodes and for each new node we measured the time to completion of a Smith-Waterman query over ten runs and took the average. For practical reasons, rather than maintaining a fixed number of database sequences and expecting a reduced time to completion, we increased the number of database sequences by 10,000 for every new node and expected a constant time to completion. The results are shown in figure 3 [3].

Allowing for some noise based on variable performance from individual nodes, the results show a linear increase in throughput for every extra node added. This perfect scalability should be expected given the embarrassingly parallel nature of the problem, and the StAcS approach has demonstrated its ability to meet that expectation. As we will discuss below, it meets these expectations while at the same time bringing some important advantages of its own.

[3]The GCUPS unit is a common way of measuring performance for Smith-Waterman and means Billion Cell Updates Per Second.

C. Costs

We used measurements of development time, lines of code and cloud computing costs as proxy indicators of overall costs.

Development time: During the development of the proof-of-concept presented here we recorded the time spent designing and coding. We found that the development effort was 22.5 person days.

Lines of code: The proof of concept consisted in approximately 1000 lines of Scala code.

Computing cost: The cloud computing solution we used charged $0.119 per node, per hour for the quad-core servers we used to perform the multi-node scalability experiment on.

IV. Conclusion

A. Parallelization Over Multiple Cores

The central finding from the results of the multiple cores experiment is that using higher abstractions to achieve multi-threading does not impair the expected speedup. There is a prevailing opinion that low-level programming is the only way to achieve efficient parallelization, and this opinion inhibits bioinformaticians from using alternatives which may be more productive. Our results demonstrate that this prevailing opinion is not sustained by the data. We submit that the advantages of recent progress in commercial software development are not being exploited enough in the bioinformatics community in general. Opportunities to reduce the barriers of costs described in the introduction are being lost.

B. Parallelization Over Multiple Nodes

Our purpose in showing these results is to demonstrate that the higher abstraction of StAcS does not compromise that expected scalability and instead simplifies it: The same abstractions that allowed Smith-Waterman invocations to be scaled across the cores of a single node, scaled also across the nodes of a cluster. Rather than learning how to use both OpenMP and MPI, bioinformaticians can distribute computations across local *and* remote processing cores using the same high-level abstraction - the Actor.

C. Costs

1) Development: The figure of 22.5 person days is quite moderate considering the robustness of the design and its demonstrated performance and scalability. The effort corresponds to less than three months work by a 'Weekend Warrior' - somebody who uses only their free time on a project. Allowing for the once-off research that was required as part of the development, we would expect future developments using StAcS to be even less costly. We believe that developers of intermediate skill level would enjoy these levels of productivity, once they had familiarized themselves with the Actor abstraction and the syntax of Scala.

2) Maintenance : The ISO 9126 model of software quality decomposes maintainability into 5 sub-characteristics:

- Analysability: how easy or difficult is it to diagnose the system for deficiencies or to identify the parts that need to be modified?
- Changeability: how easy or difficult is it to make adaptations to the system?
- Stability: how easy or difficult is it to keep the system in a consistent state during modification?
- Testability: how easy or difficult is it to test the system after modification?
- Maintainability conformance: how easy or difficult is it for the system to comply with standards or conventions regarding maintainability?

The proof of concept presented here has approximately 1000 lines of Scala code, and uses abstractions that express the parallelization problem directly in a coherent and readable way. The low line count and high readability are direct consequences of choosing high-level abstractions like Actors and Streams, as well as the Scala language itself which is demonstrably less verbose than C++ [11]. These qualities have a strongly positive effect on maintainability, impacting directly on the analysability, changeability and testability characteristics from the ISO model above.

3) Operation: Because the system runs on the JVM and does not rely on specialized processors, a wider choice of runtime platforms is available. This flexibility keeps the cost of running the software low as one can "shop around" for a competitively priced platform provider. In our case we chose a cloud provider which charged $0.119 per node, per hour for a node with 4 CPU cores (virtual). Based on that price and the established performance results, the cost on this same platform for comparing a 3000 character sequence against 1×10^7 database sequences of the same length would be $3.71. Extrapolating our experiment to a 2000 node cluster - well within proven parameters of the Akka framework - the proposed architecture would yield performance of around 1500 GCUPS. Note the fact that because of the demonstrated linear scalability of the StAcS software, the *cost per query* remains constant at less than $4 even if we scale up to these supercomputer levels of speed (assuming the system was constantly 'fed' with query sequences).

To date, the Akka framework has been tested successfully in clusters of 2400 nodes and its development team does not believe this to be a hard limit. Assuming only quad-core processors were available on each node, this configuration would still effectively constitute a 10,000 core computation fabric, and would provide supercomputer levels of performance as for a price per query as indicated above.

D. Further Work

In this proof-of-concept implementation, the database sequences are maintained locally on each worker node, with different nodes holding different sequences. The Reactive Streams read database sequences from these local files. More sophisticated approaches to storing and retrieving database sequences might include using distributed file systems such as HDFS[4] or by employing distributed databases. Most importantly, any such solution can be abstracted as a Reactive Stream. We feel this topic deserves attention in a separate study.

Although we present Actors as a high-level abstraction *with respect to Threads and Locks* we recognize that in different contexts, Actors may be considered low-level abstractions on which ever more sophisticated constructs can be built. An example of this would the be the Resiliant Distributed Dataset (RDD) of the Apache Spark project which uses Scala and Akka [12], and Akka Streams themselves which are implemented in terms of Actors. While the purpose of this paper was to advocate for the use of Actors over Threads, further work to investigate these higher abstractions for their applicability to bioinformatic problems would be beneficial.

E. Summary

Our motivation for this paper was our concern that high-throughput software in the bioinformatics field was in danger of becoming an economic bottleneck due to high costs of development, motivation and execution. We emphasised high-level abstractions over low-level abstractions to achieve the parallelization necessary in high-throughput systems. Our viewpoint was informed by the general idea that bioinformatics should learn from trends in the software industry rather than adhering to older technologies [13].

As a demonstration of this approach, we developed an inter-sequence alignment system based on Actors and Streams in the Scala language and measured its cost directly. The result was a powerful, low-cost Smith-Waterman aligner that can run on commodity servers in the cloud and scale up to supercomputer levels of performance at a low fixed cost per query.

Based on these outcomes we conclude the following:

- The StAcS architecture has clear cost advantages and lower performance penalties compared to older technologies such as OpenMP and MPI and low-level thread-and-lock programming. Given this, such architectures should be considered by bioinformatic practitioners as a better alternative to the traditional means of parallization.
- A parallel and massively scalable implementation of the Smith-Waterman algorithm, which runs on cheap commodity hardware in the cloud, was developed using the StAcS architecture in a timescale corresponding to an amateur hobbyist's efforts. This bodes well for accelerating development by full-time bioinformaticians.
- Highly-scalable, high-throughput software for bioinformatic applications can be build for lower costs than might be expected if bioinformatic software practitioners adopt the new technologies and architectures already in use in commercial software development.

[4]Hadoop Distributed File System

ACKNOWLEDGMENT

Brendan Lawlor and Paul Walsh received funding support for this research through the EU H2020 MSCA programme for the SageCare project, http://www.sage-care.eu/.

REFERENCES

[1] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer, "Design of an empirical study for comparing the usability of concurrent programming languages," *Information and Software Technology*, vol. 55, no. 7, pp. 1304–1315, 2013.

[2] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 69–84.

[3] C. D. Krieger, A. Stone, and M. M. Strout, "Mechanisms that separate algorithms from implementations for parallel patterns," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010, p. 11.

[4] J. Hong, K. Hong, B. Burgstaller, and J. Blieberger, "Streampi: a stream-parallel programming extension for object-oriented programming languages," *The Journal of Supercomputing*, vol. 61, no. 1, pp. 118–140, 2012.

[5] S. Nanz, S. West, K. Soares Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 183–192.

[6] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20.

[7] A. Ricci and A. Yonezawa, "Away from the sequential paradigm tarpit: modelling and programming with actors, concurrent objects and agents," in *Proceedings of the Second International Workshop on Combined Object-Oriented Modelling and Programming Languages*. ACM, 2013, p. 1.

[8] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997.

[9] S. Khare, K. An, A. Gokhale, S. Tambe, and A. Meena, "Reactive stream processing for data-centric publish/subscribe," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015, pp. 234–245.

[10] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "Ssw library: An simd smith-waterman c/c++ library for use in genomic applications," 2013.

[11] R. Hundt, "Loop recognition in c++/java/go/scala," *Proceedings of Scala Days*, vol. 2011, p. 38, 2011.

[12] M. A. Zaharia, "An architecture for and fast and general data processing on large clusters," 2013.

[13] B. Lawlor and P. Walsh, "Engineering bioinformatics: building reliability, performance and productivity into bioinformatics software," *Bioengineered*, vol. 6, no. 4, pp. 193–203, 2015.