Dynamic parallel message processing with transactional memory in the actor model

Yaroslav Hayduk, Anita Sobe, Pascal Felber

1 Introduction

The actor model, initially proposed by Hewitt [5], is a successful message-passing approach that can be instrumental in addressing limitations of performance/energy gains caused by current scaling trends at the CPU level. An actor is an independent, asynchronous object with an encapsulated state that can only be modified locally based on the exchange of messages. The actor model introduces desirable properties such as encapsulation, fair scheduling, location transparency, and data consistency to the programmer.

While the data consistency property of the actor model is important for preserving application safety, it is arguably too conservative in concurrent settings as it enforces sequential processing of messages, which limits throughput and hence scalability.

In our previous work [4], we addressed this limitation by proposing a mechanism to boost the performance of the actor model while remaining faithful to its semantics. Our key idea was to apply speculation, as provided by transactional memory (TM), to handle messages concurrently as if they were processed sequentially. However, while reaching substantial performance benefits for non-contended workloads, the performance of parallel processing drops close to or even below the performance of sequential processing under high contention. To improve performance especially for mixed workload scenarios, we propose a combination of two approaches: (1) relaxing the atomicity and isolation for specific read-only operations and (2) determining the optimal number of threads at runtime. Many rollbacks are caused by read-only transactions that cannot commit due to conflicts with update transactions; these rollbacks can be avoided in cases when applications can tolerate values read from a non-consistent snapshot. ScalaSTM (based on CCSTM [1]) provides an *unrecorded read* facility, in which the transactional read does not create an entry in the read set but bundles all metadata in an object. At commit the automatic validity check is omitted; without it, the result of the unrecorded read might not be fully consistent but can often safely be used for tasks such as heuristic decisions or approximate computations. By using unrecorded reads for message processing within the actor model, we can can process a large amount of read-only messages, while not conflicting with the rollback behaviour of messages processed in regular transactions (TM messages).

On the other hand TM messages are still the source of high number of rollbacks. Didona et al. [2] argue that the performance of an application is dependent on the level of concurrency, and propose to dynamically determine the optimal number of threads depending on the workload. The read-only and the regular TM messages may require different levels of concurrency. Following this observation, during high contention phases, we reduce the number of threads processing regular TM messages, which in turn allows us to increase the number of threads processing read-only messages.

2 Evaluation

We show the applicability of our approach by performing measurements on a real-world application. Multiple-point geostatistics [3] is a prominent tool shown to be effective for performing geostatistical simulations. At its core, the technique analyses the relationships between multiple variables in several locations at a time. To perform multiple-point geostatistics, we use the direct sampling [6] method. A simulation consists of a training image (TI) and a simulation grid (SG). The algorithm's task is to simulate unknown points on the SG by matching patterns from the TI. Given that the SG and the TI are usually of a large size, the simulation is performed in parallel. Besides that researchers usually evaluate the final result by visualising the SG. As the simulation times vary, it is advantageous to be able to perform intermediate snapshots.

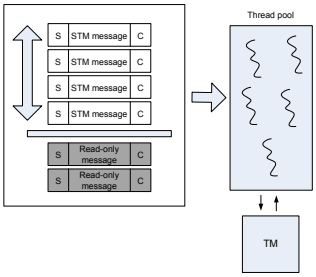

Figure 1: Dynamic concurrent message processing (S=start, C=commit).

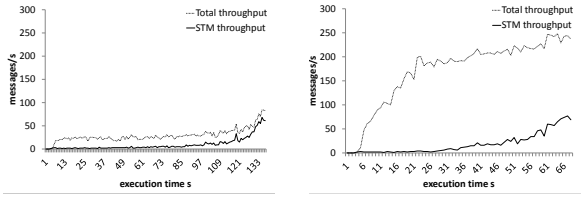

Figure 2: Message throughput for (a) static number of threads (b) dynamic number of threads.

The actor-based simulation considers the (1) *main actor*, storing the SG and the TI, (2) *SG actors* request the main actor to simulate points, and (3) *snapshot actors* request the current state of the SG from the main actor. Each SG actor claims a number of points to be simulated and sends for each of them a request for simulation to the main actor.The simulation ends once all the points are simulated. Snapshot actors also send messages to the main actor, requesting it to send a copy of the SG for visualisation. Snapshot actors repeat their request on response of the main actor. The main actor differentiates between the processing of read-only (snapshot) messages and the STM (simulation) messages. Figure 1 provides an overview of the proposed message processing. The basic principle is that we consider a pool of size $n+m$, where n STM messages are concurrently processed with m read-only messages by the same thread pool. Note that $n : m$ can be static or dynamic. We execute the benchmark on a 48-core machine equipped with four 12-core AMD Opteron 6172 CPUs running at 2.1GHz. Each core has private L1 and L2 caches and a shared L3 cache. The thread pool uses 32 threads. There are 32 simulation actors and 64 snapshot actors sending messages to one main actor. In a first scenario we consider a static ratio of $(n : m)$ 90:10% for STM messages vs. read-only messages. In a second scenario we vary the number of STM messages according to a rollback-to-commit ratio: if it is lower than a certain threshold α, we divide the number of threads processing STM messages by two; if it is higher, we multiply them by two. After, the rest of the available threads are assigned for processing of read-only messages. To avoid starvation, we never process less then 10% of messages of one type. For our experiments we set $\alpha = 1.2$. In high contention phases, only 8-10 out of 32 threads are used for processing STM messages. The remaining threads are assigned to process snapshot messages. As shown in Figure 2, we considerably improved the overall message throughput as well as reduced the algorithm's total execution time.

3 Conclusion

We proposed a method for improving message processing in the actor model by optimising the resource usage for different types of messages to be processed concurrently by an actor. We showed that different message types can be handled with different levels of concurrency, while not conflicting with each other.

References

[1] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*, 2010.

[2] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the optimal level of parallelism in transactional memory systems. In *International Conference on Networked Systems (NETYS)*, pages 233–247, 2013.

[3] F. B. Guardiano and R. M. Srivastava. Multivariate geostatistics: Beyond bivariate moments. In *Geostatistics Tróia 92*, volume 1, pages 133–144. Springer, 1993.

[4] Y. Hayduk, A. Sobe, D. Harmanci, P. Marlier, and P. Felber. Speculative concurrent processing with transactional memory in the actor model. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2013.

[5] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, 1973.

[6] G. Mariethoz, P. Renard, and J. Straubhaar. The Direct Sampling method to perform multiple-point geostatistical simulations. *Water Resources Research*, 46(11):1–14, 2010.