Epoch parallelism: one execution is not enough

Jessica Ouyang, Kaushik Veeraraghavan, Dongyoon Lee,
Peter M. Chen, Jason Flinn, and Satish Narayanasamy
University of Michigan

1 Introduction

The conventional approach for using multiprocessors requires programmers to write correct, scalable parallel programs. Unfortunately, writing such programs remains a daunting task, despite decades of research on parallel languages, programming models, static and dynamic analysis tools, and synchronization primitives.

We argue that it is futile to expect programmers to write a program that is both correct and scalable. Instead, we propose a different style of parallel execution, called *epoch parallelism*, that separates the goals of correctness and scalability into *different* executions. One execution (called the epoch-parallel execution) runs a program that is aimed at achieving correctness (or some other desirable property) but need not scale to multiple processors (e.g., the program may be single-threaded). The other execution (called the thread-parallel execution) runs a program that is aimed at scaling to multiple processors but need not always be correct. We then combine these two executions to achieve both correctness and scalability. The epoch-parallel execution counts as the "real" execution, and the thread-parallel execution speeds up the epoch-parallel execution by allowing multiple epochs to run at the same time.

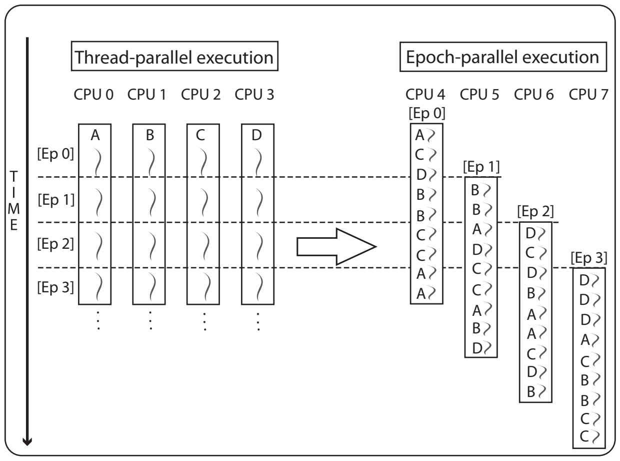

Figure 1: Epoch parallelism. In this example, the epoch-parallel execution timeslices four threads (A,B,C,D) onto a single processor to reduce the occurrence of races, enable deterministic replay, and provide sequential memory consistency. The thread-parallel execution generates checkpoints at the end of each epoch, which makes it possible to speculatively execute future epochs of the epoch-parallel execution before prior epochs complete.

2 Epoch parallelism

The goal of epoch parallelism is to take one execution that provides correctness (but may be slow or unscalable) and another execution that provides performance (but may be occasionally incorrect), and to combine these to provide both performance and correctness.

Epoch parallelism combines these two executions in the following manner. The epoch-parallel execution counts as the "real" execution, i.e. it generates the results that are output by the program. The thread-parallel execution exists merely to speed up the epoch-parallel execution. The thread-parallel execution could speed up the epoch-parallel execution in many ways (e.g., prefetching data); we focus on using the thread-parallel execution to generate checkpoints that allow multiple epochs of the slow, epoch-parallel execution to run in parallel.

Figure 1 shows an example of how this works. In this example, the epoch-parallel execution timeslices multiple threads onto a single processor. Timeslicing threads onto a single processor has several benefits: it reduces the chance that certain race conditions will occur; it makes it easier to replay deterministically; and it provides sequential memory consistency even on processors that do not support strong consistency across cores. However, timeslicing all threads onto a single processor loses all parallelism, so the epoch-parallel execution is slow. To recover this lost parallelism, the thread-parallel execution runs ahead and generates checkpoints speculatively. These checkpoints are used to speculatively start later epochs of the epoch-parallel execution before prior epochs finish. When an epoch of the epoch-parallel execution completes, it checks the checkpoint predicted by the thread-parallel execution with its own state. If these differ, the system cancels the epochs that depended on incorrect checkpoints, and the program restarts at the last matching state. To increase the chance that the epoch-parallel and thread-parallel executions match, we log the order of synchronization operations issued by the thread-parallel execution and replay these in the epoch-parallel execution.

The main cost of epoch parallelism is higher CPU utilization. In the above example, each original instruction executes twice, so the system uses twice as many cores. We believe the advent of many-core processors makes this a worthwhile tradeoff: the number of cores per computer is expected to grow exponentially, and scaling applications to use these extra cores is notoriously difficult. In addition, we may be able to leverage the similarity between the thread-parallel and epoch-parallel executions to reduce the cost. For example, the thread-parallel execution may help prefetch values or predict branches for the epoch-parallel execution.

3 Uses of epoch parallelism

The main advantage of epoch parallelism is that it no longer requires a single program and execution to satisfy the multiple goals of correctness and speed. Instead, one execution targets correctness (or other useful properties, e.g., replayability, sequential consistency, etc.) and the other targets speed. We can leverage this separation of concerns in several ways, which we categorize by by the amount of programmer effort needed.

No new code. In many uses of epoch parallelism, the programmer writes a single program, which is used by both thread-parallel and epoch-parallel executions. The epoch-parallel executes in a safer, more correct manner that allows us to check the results of the faster, though not always correct, thread-parallel execution.

Uniprocessor execution is one example of how the epoch-parallel execution can run multi-threaded programs more safely and slowly. Because multiple threads are timesliced onto a single processor and never access shared data concurrently, we can easily provide sequential memory consistency and deterministic replay, both of which are difficult and expensive to provide when threads execute in parallel. Executing on a uniprocessor also allows us to control the order of threads more precisely, and this allows us to choose schedules (e.g., non-preemptive scheduling) that avoid or detect certain types of races. The thread-parallel execution runs the same program, but executes all threads on different processors concurrently.

Another way to use epoch parallelism without writing new code is to *elide lock operations* in the thread-parallel execution. The thread-parallel execution will run faster but may experience harmful data races and atomicity violations. The epoch-parallel execution guarantees correct program behavior because it continues to execute lock operations. This is a form of optimistic concurrency control (similar to transactional memory), but checks for conflicts by comparing state between the thread-parallel and epoch-parallel executions, rather than by monitoring memory operations.

Some new code. In other uses of epoch parallelism, the programmer may add code to the epoch-parallel execution to make it more robust. Examples of such extra code include assertions on data structure integrity, security checks, or software rejuvenation. Or the programmer may modify code in the thread-parallel execution to make it faster (at the potential cost of correctness), for example by using fast, approximate algorithms to compute results.

Completely new program. The most general use of epoch parallelism is to write two completely different programs, one buggy and fast, the other slow and correct. In the extreme case, the programmer would write two completely different programs, such as a single-threaded program (easy to write correctly, but slow) and a multi-threaded program (fast, but difficult to write correctly). These two programs would run using epoch parallelism style to provide the speed of the multi-threaded program and the correctness of the single-threaded program, thus freeing the programmer from ever writing a fast, correct multi-threaded program. The main problem with this approach is that it becomes difficult for the thread-parallel execution to generate checkpoints that can be used to start future epochs of the epoch-parallel execution. One way to circumvent this problem is to start epochs at program points with minimal state, by saving the persistent program state to a file at the end of an epoch, then importing that file as the starting state of the next epoch.