SUMMARY:
Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan.
Interpreting the data: Parallel analysis with sawzall.
Scientific Programming, vol. 13, number 4, pages 277-298, 2005

DATE: 22 February 2009

The objective of this paper is to present a specialized programming language called Sawzall. Sawzall is an interpreted language built over Google's MapReduce infrastructure with the purpose of providing programmers with a very easy and extremely concise alternative to doing map-reduce type jobs. The language defined is strongly typed and uses value types exclusively (no reference types are provided). It is however tailored for the kinds of data analysis typically done within Google using map-reduce. Thus types like time, arrays, tuples, collections, etc. are also provided. The language also abstracts away all concurrency, effectively allowing someone with a total lack of knowledge of parallel programming to design a program that will be run in parallel on thousands of machines. Each record in a large data set is individually run through the program written in Sawzall without any interaction with other records. The underlying MapReduce infrastructure transparently handles job allocations to machines without the involvement of the programmer. The results of each individual run is then aggregated first locally, then globally (possibly in multiple stages) by some aggregator function which is written in another language (like C++) and made available through libraries. The Sawzall programmer need only indicate the function used, and the system will take care of the rest, including parallelization and aggregator job scheduling.

Being an interpreted language doesn't have a large impact on performance, largely because the class of programs typically written for Sawzall will have an I/O bottleneck and little computing requirements. Still, Sawzall performs a lot better in benchmarks than other older interpreted languages like Python, Ruby and Pearl. It is however 1.6 times slower than interpreted Java and 51 times slower than compiled C code. Scaling is yet another strong point in the system which offered near optimal scaling in tests (each new machine added to the system contributed 0.98 machines worth of throughput).

SUMMARIZED BY: Cătălin-Alexandru Avram