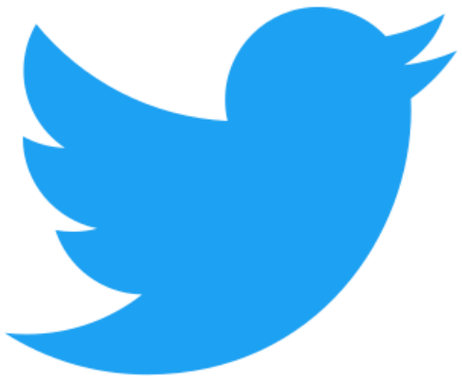

Getting Pants Performance for Free via Parallelism using Graal `native-image`

Danny McClanahan / @hipsterelectron / @pantsbuild
Twitter Inc.

Overview

1. native-image's usage in open-source
2. native-image for scalafmt
3. native-image for scalac

`native-image` for build tools: building user code

- Build user code with native-image:
  - Take ~uncontrolled user code, build it into an uberjar, then invoke native-image.
    - maven and sbt can do this for you.
  - There can be some difficulties in applying this to real applications, including the process of inferring or specifying reflective accesses.
    - *as discussed by Alibaba earlier today.*
- The coursier JVM resolve tool has support for publishing applications to a registry which are then downloaded and built with native-image on demand.
- This is becoming more and more feasible through a lot of open-source momentum for native-image.
  - native-image is becoming more widely used for command-line tools (including coursier itself!).

`native-image` for build tools: building JVM tools

- Build JVM tools with native-image:
  - These are tools that are used by a build tool to build user code somehow.
    - Scalac, Javac, Thrift, Scrooge, JUnit, ...
  - Code is mostly controlled
    - Can (temporarily!) fork the code if necessary
  - Have ~complete control of *how* and *where* the tool is invoked.
    - Can shard input over multiple instances of the tool.
    - Can invoke the tool via RPC.

`native-image` for JVM tools: formatting

- Applied native-image to the scalafmt tool:
  - The scalafmt tool has had specific open-source contributions to enable native-image.
    - Including reflection/resource json config.

  - Previously, required special support in Pants:
    - in order to build as a native-image on demand: https://github.com/pantsbuild/pants/pull/6893

- Now, it's published as a native-image directly:
  - can consume it in pants without requiring special native-image support: https://github.com/pantsbuild/pants/pull/8772.

`native-image` for JVM tools: formatting

- Described in 2019 Graal CGO workshop submission:
  1. Formatting is embarrassingly parallel.
  2. Using native-image improved performance over a warm JIT.

    - Spawning one native-image process per 150 input files on the open-source Finagle project achieved a 25% speedup over a warm JIT, without having to keep a warm JIT process alive using a GB of memory.

    - Having your cake and eating it too!

`native-image` for build tools: compilation

- Compilation is *not* embarrassingly parallel *(for most JVM-based languages)*.
  - C/C++ are -- can provide header files to the compiler all at once, and kick off compilations for each translation unit at once.

- Twitter uses Scala *(a lot)*, and Scala requires compilation in topological order.
  - The Scala compiler itself doesn't have a lot of parallelism (there is active work to improve this).

- There is a proprietary Scala compiler known as Hydra, from Triplequote, which has much more parallelism.
  - We're not interested in that, because we have an open-source alternative that makes use of rsc and native-image.

`native-image` for build tools: scala compilation with pants

- Pants (@pantsbuild) is an open-source build tool originally developed at Twitter.
  - Supports building python, scala, and many more.
    - Supports codegen with Thrift/Scrooge, (widely used at Twitter).

`native-image` for build tools: scala compilation with pants

- Pants builds Twitter Scala code by executing the zinc incremental scala compiler: https://github.com/sbt/zinc/.
  - Pants uses Nailgun to keep a warm JIT: http://www.martiansoftware.com/nailgun/background.html.
- Zinc takes a long time to start up, but is able to compile Scala code on Twitter developer laptops by using thread-based parallelism.
  - Blocked on topological ordering of Scala compilation.
  - Blocked by the amount of free resources on the developer's laptop.
  - Produces plenty of classfiles that are not necessary for the product that the user actually requires, but are used transitively by another target.

`native-image` for build tools: scala compilation with pants and rsc

- rsc allows embarrassingly parallel compilation of Scala code (https://github.com/twitter/rsc).
  - It can "header-compile" scala code by producing an "mjar" that has method stubs for all public members of Scala classes.
    - Scalac can then use these as if they were real Scala sources.
      - This initial run is in topological order, but is *extremely* fast.
        - The rsc runtime is heavily I/O-bound, actually.

- rsc is an *extremely* compact and well-written compiler developed by Eugene Burmako during his time at Twitter, now developed by Win Wang at Twitter.

`native-image` for build tools: [scala compilation with pants, native-image, and remote execution]

- native-image: can invoke with
  - *relatively deterministic performance,*
  - *no startup time!*

- This makes it *extremely* interesting to invoke *remotely*:
  - No need to "warm up" and keep track of warm nodes to be able to serve compilation requests.
  - No difficulty with responding to "bursty" input.

`native-image` for build tools: difficulty building zinc and the scala compiler with native-image

- The scala compiler is somewhat easy to build with native-image: this was solved during ScalaDays 2018: https://github.com/graalvm/graalvm-demos/tree/master/scala-days-2018/scalac-native.
  - However, zinc is a *lot* of code from sbt which required a lot of manual configuration (in reflect-config.json), along with some substitutions.
- Scala macros are implemented via reflection -- these need to be specified in reflect-config.json.
  - We used the native-image-agent for a while, but it would only cover macros that happened to be used in that compilation run, and was very hard to automate.
  - Eventually we realized we could scan classfiles and pull macro definitions from them using the scala compiler API, which allowed us to automate this.

`native-image` for build tools: scala compilation with pants, rsc, native-image, and remote execution

- Once we got zinc building with macros working, we had a fully functional compilation pipeline.
- As the backend, we used Scoot (https://github.com/twitter/scoot), which implements the Bazel remote execution API.
- Tried many, many different execution modes, mixing local and remote execution, along with compilation via rsc or not.
  - Used current pants execution of zinc on the local machine with a nailgun as a control.

`native-image` for build tools: scala compilation with pants, rsc, native-image, and remote execution

- No charts here :(
- No fully-remote method was able to achieve the same performance as local execution.
  - The difficulty wasn't in native-image runtime, but rather in the time taken to ship classfiles back and forth across the network.
  - Any compilation method which has other compile jobs waiting on sending bytes back and forth over the network is going to have a compounding slowdown at deeper levels of the dependency graph.

`native-image` for build tools: scala compilation with pants, rsc, native-image, and remote execution

- No charts here :(

- Many recent optimizations made in rsc would show a greater runtime speedup when running rsc in a JVM, as opposed to native-image.
  - Especially optimizations made to symbol interning which relied on hashmap access, for some reason.

- As a result, we decided to try a hybrid approach which used rsc in a warm JVM on the local machine, and to do "real" scalac compiles remotely, completely in parallel.
  1. Avoided blocking any jobs on network requests.
  2. Made better use of the rsc optimizations which work better on HotSpot for some reason.
  3. All blocking on the network was blocking on longer compile jobs (*not* rsc), which reduced the relative time taken by sending inputs and outputs over the network.
  4. All blocking on the network was done entirely in parallel to further work with rsc, which meant we were able to saturate the network interface while doing local I/O.

`native-image` for build tools: scala compilation with pants, rsc, native-image, and remote execution

- No charts here :(
- The hybrid approach of:
  - running rsc in a JIT locally
  - spreading non-header compiles across remote compile job requests
- ...was ~2x as fast as the current state-of-the-art (invoking zinc in a JIT).
  - Slides from Stu Hood's talk at ScalaDays will show charts with numbers for these!
  - Also shows a Zipkin trace which demonstrates that we were able to saturate the network with parallel compile requests.

`native-image` for build tools: *Conclusions*

- native-image is actually super approachable for open-source JVM tools right now, and is becoming more popular, with build tools offering it as a build option for user applications.

- native-image can be specifically useful for multiple different types of JVM tools in a build tool, and can unlock wildly different execution modes:
  - remote execution with zinc
  - embarrassingly parallel local execution with scalafmt

`native-image` for build tools: *Future Work*

- Does invoking rsc with the Graal JIT compiler improve performance more?
  - What leads to the lesser magnitude of rsc optimizations when run in native-image?

- What kind of scheduling algorithm for local and remote compile jobs will lead to optimal compile performance?
  - How do we appropriately weight, simulate, and optimize this for an application (pants) which invokes JVM tools as subprocesses?

- What kind of tools can we contribute to the community to improve the ability to automatically build things with native-image?
  - Like extracting macros from the classpath.

Questions?

@hipsterelectron: Loves pants and open-source build tooling!

@pantsbuild: The object of my affection!