# Parallel builds in Maven 3

Maven 3.x has the capability to perform parallel builds. The command is as follows:

mvn -T 4 clean install # Builds with 4 threads
mvn -T 1C clean install # 1 thread per cpu core
mvn -T 1.5C clean install # 1.5 thread per cpu core

This build-mode analyzes your project's dependency graph and schedules modules that can be built in parallel according to the dependency graph of your project.

The above mentioned thread per cpu core means the number of cores is used as a multiplier

## Experimental feature for 3.0!

The parallel build feature has been subject to extensive testing, but the maven ecosystem is diverse so there **will** be undiscovered issues. We recommend that users of the parallel build feature establish their own reference as to how well this works for their project, preferably starting with everyday builds as opposed to final production releases.

The parallel build functionality is brand new, and although they are tested with quite a few projects they do not have the general wisdom accumulated by running on multiple project types on multiple platforms within the community. So take a little care.

## What performance boost can be expected ?

This depends greatly on your module structure, but the following observations have been made:

- 20-50% speed improvement is quite common.
- Distributing tests among your modules is likely to improve performance, putting all your tests in one module decreases it - unless you run one of the parallel surefire test providers.
- Running tests in parallel within a single surefire-instance is a little different from running multiple surefire-runs (from separate projects), since there will be different classloaders. Remember that tcp/ip ports and files are still singletons.

## Plugin/Settings compatibility

The functionality within the Maven3 core is thread safe and well behaved, but the maven ecosystem consists of a large number of subsystems, and a lot of plugins have a large number of dependencies. Not all of these plugins/libraries were written with thread safety in mind. As of beta-2 maven 3 will warn noisily of any plugins present in the build that are not @threadSafe.

The following plugins/settings are KNOWN to have incompatibilities when running any of the parallel modes:

- Surefire with forkMode=never, surefire [2.6,) asserts this.
- maven-modello-plugin, fixed in [1.4,)
- All maven-archiver clients (EAR, EJB, JAR, WAR etc), see https://issues.apache.org/jira/browse/MSHARED-148 related/links section. EAR, EJB, JAR and WAR are fixed in latest version.

## Known non-thread safe libraries

Known thread safety problems have been *fixed* in the following library versions:

plexus-utils 2.0.5
maven-archiver 2.4.1
plexus-archiver 1.0
plexus-io 1.0

## Thread safe plugins and libraries

- Maven Clean Plugin 2.4.1
- Maven Compiler Plugin 2.3.1
- Maven Install Plugin 2.3.1
- Maven Resources Plugin 2.4.3
- Maven Surefire Plugin 2.6
- Maven EAR Plugin 2.4.2
- Maven EJB Plugin 2.3
- Maven JAR Plugin 2.3.1
- Maven WAR Plugin 2.1
- Maven Shade Plugin 1.3.3
- Maven Changes Plugin 2.4
- Maven Checkstyle Plugin 2.6
- Maven Antrun Plugin 1.4
- Maven Assembly Plugin 2.2.1
- Maven GPG Plugin 1.1
- Maven Plugin Plugin 2.7

- Maven Remote Resources Plugin 1.2.1 (1.2 is *not* threadsafe)
- Maven Source Plugin 2.1.2
- maven Enforcer Plugin 1.0.1

## Known issues

It is not required to report jiras for these issues:

The console output of both parallel modes is not sorted in any way, which can be a bit confusing. https://issues.apache.org/jira/browse/MNG-2727

## Mojo thread safety assertion checklist

Sometimes it can be hard to determine if a plugin and the underlying libraries are thread-safe, so when adding @threadSafe the following checklist can be used:

- Check all static fields/variables in plugin/plugin code are not subject to threading problems.
  You might want to pay special attention to static member variables of the subclasses of "java.text.Format" (NumeberFormat, DateFormat etc), most of which
  are not threadsafe and cannot be shared as static variables.
- Check any plexus components.xml; if the components defined are singletons they need to be threadsafe.
- Check for presence of known tainted libraries.
- Check thread safety of any other third party libraries. This last item can be a bit hard, but inquiries on mailing lists can get you a long way.

This checklist qualifies for a "simple thread safety" review of a mojo.

If you need to learn more about multithreading and the java memory model it is probably wise to start off with a book like "Java Concurrency In Practice" or similar.

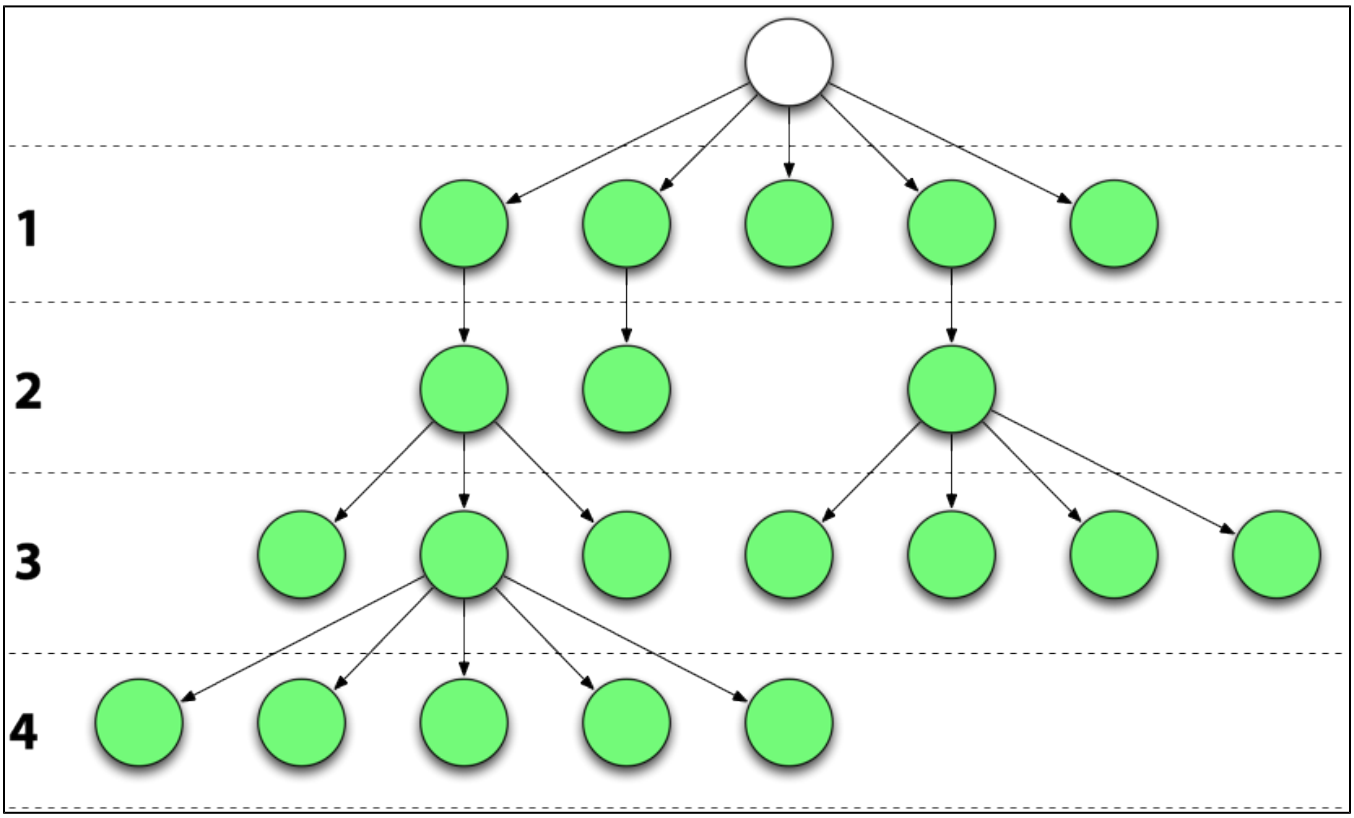
If a mojo uses a known-non-threadsafe external dependency, you may want to do something like this:

```
public class MyMojo
    extends AbstractMojo
{

  private static final Object lock = new Object();

  public void execute()
  {
     synchronized( lock)
     {
         // Main mojo code
     }
  }
}
```

# How Execution is evaluated




Each node in the graph represents a module in a multi-module build, the "levels" simply indicate the distance to the first module in the internal reactor dependency graph. Maven calculates this graph based on declared inter-module dependencies for a multi-module build. Note that the parent maven project is also a dependency, which explains why there is a single node on top of most project graphs. Dependencies outside the reactor do not influence this graph.

For simplicity; let's assume all modules have an equal running time. This build should have level 0 running first, then a fanout of up to 5 parallel on level 1. On level 2 you'll be running 3 parallel modules, and 7 on 3, 5 on level 4.

This goes by declared dependencies in the pom, and there is no good log of how this graph is actually evaluated. (I was hoping to render the actual execution graph, but never got around to finding a cool tool/way to do it - plaintext ascii in the -X log would be one option).

Of course, in real life your modules do not take equal amounts of time. Significant gains are common when the project has one or more "api" modules and dependencies on the "api" modules (and just bring the "impl" version of the module into the actual assembly that will be started). This design normally means your big chunky modules depend on lightweight "api" modules that build quickly.

The parallel build feature rewards "correct" modularizations. If your project has degenerated inter-module dependencies (execessive dependencies inside reactor), you will probably see gains by cleaning up the dependencies.