Parallel Building with GNUstep Makefiles

Nicola Pero nicola.pero@meta-innovation.com

First Edition: March 2010

1 Introduction

In this tutorial we will learn how GNUstep make parallelizes building your project. You will learn how to organize your GNUmakefiles to maximize and control the parallelization of your build. This is an advanced topic; please refer to "Writing GNUstep Makefiles" and "More on GNUstep Makefiles" for an introduction to GNUstep make.

This tutorial only applies to gnustep-make 2.4.0 or later; parallel building was only partially supported before 2.4.0.

2 Parallel building

2.1 What is parallel building

By default, gnustep-make builds software in "serial" mode. Each step of the building process is executed in a well-defined order. A step won't start until the previous step has completed.

If you have more than one CPU, or if you have a modern multi-core CPU, you can speed up your build significantly by telling gnustep-make to build software in "parallel" mode. In this case, whenever possible gnustep-make will run a number of tasks in parallel.

There are a number of simple rules that gnustep-make follows to decide when tasks can be run in parallel, or when there is a sequential order to be respected. They are the main topic of this tutorial.

2.2 Doing a parallel build

To do a parallel build, you just need to pass the '-j' option to make. For example,

```
make -j
```

'-j' will try to run as many subprocesses as possible. You can limit it by adding a number, for example 'make -j 4' might be a good one.

If you try this out on a project containing many files to compile, and if you have a machine which can run multiple flows of execution in parallel, you should get a much faster build as the files are compiled in parallel.

In the rest of the tutorial we'll learn how gnustep-make decides which steps are done in parallel and which steps are done in serial order.

3 Single instance

In this section we look at building a “single instance”, that is, a single thing. For example, a single tool. We’ll cover building more than one thing (for example, two tools, or a library and a tool) in the following sections.

3.1 Compiling multiple files

GNUstep make automatically compiles multiple files for the same instance in parallel.

Let’s look at a very simple example: an Objective-C tool composed of two source files –

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = HelloWorld
HelloWorld_OBJC_FILES = HelloWorld.m main.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

In this case, if you use ’make -j’ you will see that the two files are compiled in parallel. So, the process is really composed of two steps:

```
Step 1. Compiling HelloWorld.m and main.m (in parallel)
Step 2. Linking them together
```

3.2 Adding before-xxx-all and after-xxx-all steps

If you need to add some custom steps before or after your instance is built, you can use before-xxx-all and after-xxx-all rules. These steps will always be executed exactly before and after the compilation of your instance. In other words, the building process for the HelloWorld tool is really composed of the following steps:

```
Step 0. Execute before-HelloWorld-all:: if it exists
Step 1. Compiling HelloWorld.m and main.m (in parallel)
Step 2. Linking them together
Step 3. Execute after-HelloWorld-all:: if it exists
```

For example, let’s say that for some reason you need to create a header file, HelloWorld.h, which is then included by your Objective-C files HelloWorld.m and main.m. Obviously this needs to happen before any compilation takes place.

To do so, in your GNUmakefile.postamble you would add the following:

```
before-HelloWorld-all::
	cp HelloWorld.h.in HelloWorld.h
```

Please note that there should be a TAB character before the ’cp’, and that if you don’t have a GNUmakefile.postamble, you can simply put this rule in your GNUmakefile, at the end of it, after all the gnustep-make makefiles have been included.

In the real world, this would probably be a more complicated rule, creating HelloWorld.h from HelloWorld.h.in by using sed or some other file editing tool. But whatever it is, you can simply have your code executed before any compilation is done by placing it in a before-HelloWorld-all:: rule.

Obviously if your tool is called HelloMoon, then you would place your code into a before-HelloMoon-all:: rule.

This code would always be executed serially, before the tool is compiled.

4 Multiple instances

4.1 Multiple instances of the same type

Multiple instances of the same type are built in parallel.

Let's say that you are building two tools instead of one. Your GNUmakefile might look like the following one:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = HelloWorld HelloMoon
HelloWorld_OBJC_FILES = HelloWorld.m main1.m
HelloMoon_OBJC_FILES = HelloMoon.m main2.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

When you build this in gnustep-make using 'make -j', the two tools will be considered independent and be built in parallel.

So, the top-level flow looks as follows:

```
Step 0. Execute before-all:: if it exists
Step 1. Build HelloWorld and HelloMoon in parallel
Step 2. Execute after-all:: if it exists
```

Building each of the HelloWorld and HelloMoon tools would follow the flow described in the previous section, meaning that the compilation of the files for each of them will be parallelized too! If you build with 'make -j' (and your CPU can make 4 concurrent flows of execution), for example, it is likely that all the 4 files will be compiled in parallel. The two tools are built in parallel, and then each of them fires the compilation of their 2 files in parallel. As a result, all of the 4 files are actually built in parallel. All the other stages of building the tools are also done in parallel; for example, the tools are linked independently, in parallel.

4.2 Instances of different types

Instances of different types are not built in parallel; they are built in the order in which the gnustep-make make fragments (tool.make, library.make, etc) are included.

This behaviour is backwards-compatible with previous releases of gnustep-make and it's important to know about it (later on we'll explain how you do a parallel build in this case by moving the instances into different subdirectories and using the new parallel-subdirectories.make makefile).

For example, if your GNUmakefile builds a library and a tool, they will be always be built separately and in serial order. If you include tool.make before library.make, the tool will be built first; if you include library.make first, the library will be built first.

Let's look at a quick example –

```
include $(GNUSTEP_MAKEFILES)/common.make

LIBRARY_NAME = HelloMoon
HelloMoon_OBJC_FILES = HelloMoon.m HelloMars.m

TOOL_NAME = HelloWorld
HelloWorld_OBJC_FILES = HelloWorld.m main1.m

include $(GNUSTEP_MAKEFILES)/library.make
include $(GNUSTEP_MAKEFILES)/tool.make
```

In this case, the HelloMoon library will be built before the HelloWorld tool because library.make is included before tool.make.

So, the building in this case will work as follows:

```
Step 0. Execute before-all:: if it exists
Step 1. Build HelloMoon
Step 2. Build HelloWorld
Step 3. Execute after-all:: if it exists
```

5 Subdirectories

You may have felt that the rules presented in the previous section were rather arbitrary. Instances of the same type are built in parallel, while instances of different types are not. What if you need something different ?

gnustep-make 2.4.0 introduces two new makefile fragments (serial-subdirectories.make and parallel-subdirectories.make) that allows you to have total control on the order and the parallelization in which instances are built. We will refer to them as the "subdirectories" makefiles. They are meant to replace the traditional aggregate.make in the long-term.

5.1 Examples

Let's say that you want to build a tool and a library. To use the new subdirectories makefiles, first of all you need to put the tool and the library into two separate subdirectories. For example, you could put the tool into a subdirectory called Tools, and the library into a subdirectory called Source (they could be called anything you want). Each of them will have its own standalone GNUmakefile.

At the top-level, you create a GNUmakefile that will drive the building of the subdirectories. Here is an example:

```
include $(GNUSTEP_MAKEFILES)/common.make

SERIAL_SUBDIRECTORIES = Source Tools

include $(GNUSTEP_MAKEFILES)/serial-subdirectories.make
```

This will cause gnustep-make to go into the SERIAL_SUBDIRECTORIES in order, and build them serially, one by one. So, it will first build Source (that contains our library) and then Tools (that contains our tool).

If you want the two subdirectories to be built in parallel, you just need to use parallel-subdirectories.make, as follows:

```
include $(GNUSTEP_MAKEFILES)/common.make

PARALLEL_SUBDIRECTORIES = Source Tools

include $(GNUSTEP_MAKEFILES)/parallel-subdirectories.make
```

This will tell gnustep-make that it should fork off two subprocesses, and build the Source and Tools subdirectories in parallel.

5.2 Advanced usage of subdirectories

If you now nest subdirectories at different levels, you can control exactly what you want to be built in parallel and what you want to be built in serial order.

For example, you may have a few frameworks that can be built in parallel, followed by a few tools that can be built in parallel, but the frameworks always need to be built before the tools.

In this case, you could have a Frameworks/ and Tools/ subdirectories, built using serial-subdirectories.make so that they are built in that order. Inside Frameworks/ you then have a GNUmakefile that uses parallel-subdirectories.make to build a number of frameworks in parallel (each of them in its own subdirectory), and similarly in Tools/ to build the tools in parallel.

The advantage of parallelizing the build of entire directories might not be apparent simply because your CPU might not support enough concurrency to do more than a few things in parallel. But if you simply organize your directory structure and use parallel-subdirectories.make and serial-subdirectories.make in a way that makes sense, you'll get the full benefit of more parallel-capable CPUs as soon as you get them. Your build will scale really well when you add CPUs and cores.

Moreover, when you parallelize building entire directories you not only parallelize the compilation steps, but all the other ones; linking and even before-all:: and after-all:: rules.

For example, if you have in one directory a before-all:: rule which takes 10 seconds to run, since it is executed serially it may prevent anything else from running for these 10 seconds. For 10 seconds all your other CPUs or cores might be sitting idle. But if you parallelize the build of that directory with the build of another directory, even during these 10 seconds, the other CPUs or cores might still be busy building things in the other directory.

5.3 Backwards compatibility between subdirectories and aggregate.make

If you need your software to build with gnustep-make ¡ 2.4.0, then you can't really use parallel-subdirectories.make or serial-subdirectories.make because they weren't available in older versions of gnustep-make. You need to use aggregate.make.

aggregate.make will always do a serial build in older versions of gnustep-make. In newer versions, it will do a serial build by default, but will do a parallel build if you set

```
GNUSTEP_USE_PARALLEL_AGGREGATE = yes
```

So, here is how to build a set of subdirectories in serial order –

```
include $(GNUSTEP_MAKEFILES)/common.make

SUBPROJECTS = Source Tools

include $(GNUSTEP_MAKEFILES)/aggregate.make
```

and here is how to build them in parallel oder –

```
include $(GNUSTEP_MAKEFILES)/common.make

SUBPROJECTS = Source Tools

GNUSTEP_USE_PARALLEL_AGGREGATE = yes
include $(GNUSTEP_MAKEFILES)/aggregate.make
```

This last example will simply do a parallel build on newer gnustep-makes, and a serial build on older ones that don't support it.

aggregate.make will be slowly phased out in the following years, so if you don't need to support older versions of gnustep-make, you could simply use serial-subdirectories.make and parallel-subdirectories.make.

6 Subprojects

6.1 Why do subprojects exist ?

Before version 2.4.0, gnustep-make did not support having source files in subdirectories. For example, you couldn't write:

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = HelloWorld
HelloWorld_OBJC_FILES = Subdirectory/HelloWorld.m

include $(GNUSTEP_MAKEFILES)/tool.make
```

This works if you use gnustep-make 2.4.0 (or later); but wouldn't work with previous versions of gnustep-make, because HelloWorld.m is in a subdirectory. So, people who wanted to organize their source files in different subdirectories had to use "subprojects". You created a GNUmakefile in each of the subdirectories (which would build using subproject.make), then specifies that the tool had some subprojects. So, the tool's GNUmakefile would have been –

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = HelloWorld
HelloWorld_SUBPROJECTS = Subdirectory

include $(GNUSTEP_MAKEFILES)/tool.make
```

and in the subdirectory you'd have had a GNUmakefile such as the following one –

```
include $(GNUSTEP_MAKEFILES)/common.make

SUBPROJECT_NAME = Subdirectory
Subdirectory_OBJC_FILES = HelloWorld.m

include $(GNUSTEP_MAKEFILES)/subproject.make
```

You'll appreciate how easier it is to specify source files in subdirectories directly in the tool's GNUmakefile (as allowed by gnustep-make 2.4.0), without having to have an additional GNUmakefile in each subdirectory.

6.2 Most of the times, you do not need subprojects any more

As explained, in recent versions of gnustep-make, there is no longer any need for subprojects just to organize your source files in subdirectories. You can simply put your source files into subdirectories, then list them all in your GNUmakefile. The project is simpler, and gnustep-make is able to parallelize as much as possible since it will (potentially) be able to compile all the files in parallel.

6.3 Subprojects and paralell building

If you ever need to use subprojects in a parallel build, you need to know that, for backwards-compatibility, subprojects are built in the order they are specified, and are built before the rest of the source files are compiled.

So, in the example above of a tool HelloWorld with a Subdirectory subproject, the build flow to build the tool is as follows:

```
Step 0. Execute before-HelloWorld-all:: if it exists
Step 1. Build Subdirectory
Step 2. Build HelloWorld
Step 3. Execute after-HelloWorld-all:: if it exists
```

if you had more than one subproject, they would be built in the specified order. Most likely, they are independent though, and should be built in parallel. You can very simply obtain that result by getting rid of subprojects altogether and listing all files in the tool's GNUmakefile. This will also get rid of the intermediate linking steps for subprojects, speeding up your build even more.

7 Disabling parallel building

If parallel building is causing problems, you can always turn it off. There are three ways of turning it off:

1. globally, by configuring gnustep-make with ./configure –disable-parallel-building. This sounds a bit extreme, but it's an option if you want everything to be always built serially.

2. for a specific GNUmakefile, by adding

   ```
   GNUSTEP_MAKE_PARALLEL_BUILDING = no
   ```

   to the GNUmakefile, just before including common.make. This is probably the most useful trick; if for any reason you find a GNUmakefile is not working with parallel building and you don't know how to fix it, you can disable parallel building in that GNUmakefile while still leaving it on for everything else.

3. for a specific compilation run, by simply not using '-j' when invoking make. :-)