USING STANDARD TOOLS TO PACKAGE AND DISTRIBUTE SCIENTIFIC SOFTWARE C AND FORTRAN LIBRARIES: A DEMONSTRATION WITH THE GENERAL PURPOSE TIMING LIBRARY (GPTL) (REV. A)

Edward Hartnett [1,2,] James Rosinski [3,]

1 CIRES, University of Colorado, Boulder, CO 80309, USA
2 NOAA/ESRL/GSD, Boulder, CO 80305, USA
3 NCAR/JCSDA

ABSTRACT

Important scientific C and Fortran software packages are frequently distributed through use of home-grown approaches rather than the standard GNU tools autoconf, automake, and libtool. This results in wasted effort for the distributors of the software, as they struggle to achieve portability. Home grown approaches also tend to waste the time of all software installers, which have to struggle to correctly build the software. In this poster the conversion of a scientific software package, the General Purpose Timing Library (GPTL) from a home-rolled build system to a standard build is demonstrated. This results in nearly an order of magnitude decrease in build system complexity, measured by lines of code, and an increase in build system features for the user, including automatic shared library builds, standard configure options, and standard make options, as well as increased portability and ease of use. As demonstrated by major scientific software packages like netCDF and HDF5, these tools can be used very effectively to package software portably, including software that is built and run on High Performance Computing (HPC) systems.

DOCUMENT HISTORY

Date	Version	Changes
1/9/19	original	Presented at AMS annual meeting.
4/15/19	Rev. A	Added section 5.7. Clarified --enable-pmpi.

1. INTRODUCTION

1.1 Motivation

Complex software packages can require considerable effort to build and install. Scientific software, such as numerical models, make frequent use of math, I/O, compression, and other libraries, which must be installed, and their location communicated to the software build. Dependencies between software packages can also become complex; features may be used which require a specific version of a library, or specific information about the library's capabilities.

The GPTL legacy build system must determine the availability (or not) of several other libraries. It also has some code that only allies to x86 microprocessors. The build system was functional, but required manual intervention from the user.

In order to reduce the maintenance burden, increase portability, and support shared library builds, a new autotools-based build system has been developed.

1.2 Autotools

Standard Linux tools autoconf, automake, and (for libraries) libtool are used to configure and build packages in a standard way. Collectively these tools are referred to as "autotools".

Anyone who has built open-source software on Linux machines has encountered these tools. To build, the user first (optionally) specifies some standard compiler flags, and then runs the *configure* script. The *configure* script queries the target machine for information needed to correctly build the software. It then constructs Makefiles on the build system. The user runs *make* to build the code, *make check* to build and run tests, and *make install* to install the code.

To accomplish this, the developer creates an autotools build system, which includes one configure.ac file for the project, which is the basis of the configure script, and one Makefile.am file for each directory in the build. The Makefile.am files are used to construct the final makefiles.

1.3 GPTL

The General Purpose Timing Library(https://jmrosinski.github.io/GPTL) provides detailed timing information for an application by instituting calls to *start* and *stop* timers. These calls can be manually inserted by the user, or automatically generated through the use of flags available on most compilers (e.g. -finstrument-functions on GNU and Intel, -Minstrument=functions on PGI) . The library is thread-safe, and provides per-thread timing information. It also can provide optional summary information across

Corresponding author address: Edward J. Hartnett, edward.hartnett@noaa.gov

MPI tasks. There are other optional capabilities, such as integration with the PAPI performance counter library, and automatic generation of MPI stats if C-preprocessor flag ENABLE_PMPI is defined.

2. LEGACY BUILD SYSTEM

2.1 Makefiles

The GPTL libraries were built with *make*. A selection of different Makefiles were included for different systems and compilers. To build the code, the user first copies the makefile template that most matches the build system, edits it, then types *make.*

2.2 Combined C and Fortran Libraries

Like many science codes, the original GPTL library contains both C and Fortran libraries. Although these seem intimately connected to the science developer, they are best distributed and maintained as separate libraries, repositories, and distributions.

2.3 Setting Compilers and Flags for the User

GPTL, like many home-rolled build systems, attempts to set compilers and flags in the makefiles. That is, with certain compilers, the build attempts to use certain flags, and those flags are pre-programmed into the build system.

Increased portability is attempted by providing a variety of makefile templates, each with different compilers and compiler flags. Although this works for a small number of cases, it is not a scalable solution.

It is not scalable because it is impractical for all compiler and compiler options to be supported in the makefiles. A new compiler or a new flag requires editing of the makefiles, and ultimately a new release to support the new compiler or flag.

Setting flags for the user is also not a long-lasting solution. Flags that are optimal with the state of the system today, may be counterproductive in a few years when the version of the compiler or the capabilities of hardware have changed. Over time, the flags that are built into the build system may become less relevant and less correct.

The autotools approach allows the user full control over compilers and flags. Only the user can select the optimal compiler and flags for their target system. Thus autotools-based builds can easily take advantage of new features or special performance options on the target system, even if these features are unknown to the GPTL developers. When a new compiler is introduced or new flags added to existing compilers, the autotools build system does not have to change to support the new compiler or new features.

The legacy build system also included a configure.ac, which built a single program: the suggestions script. This script helped with the manual creation of Makefiles by the GPTL developer.

3 C LIBRARY

3.1 Building the Build System

The build system consists of the configure.ac file and all the Makefile.am files, but these files cannot be directly run to build the GPTL libraries. Instead, these files are used as inputs to the autotool utilities *autoconf* and *automake.* These programs create the *configure* script and Makefile.in files.

When installing the package, the *configure* script is run, which uses the *Makefile.in* files, and information gathered from the target machine, to create *Makefile* files. These files are then used to build the package when the user types *make.*

To assist the developer in managing the build system, the *autoreconf* command will perform a complete build of the build system, running all necessary scripts and tools in the correct order.

3.2 Configuration

The configuare.ac file is where all configuration of the build system is managed. This file is in turn converted by *autoconf* to the *configure* script that is run by the end user.

The configure.ac script is in m4, a macro language that is part of the POSIX standard. It is a simple language, similar to the C preprocessor, but with more features. The configure.ac script also includes Bourne shell script commands.

The *configure* script must learn about the target build system, and must also provide options to the user. The --help option is automatically provided to help end-users decide what options are available.

3.2.1 Handling Configure Options

Many configuration options are provided by *autoconf,* with no developer effort required. For example, --enable-shared vs. --disable-shared, or the --prefix option, which allows the user to specify the installation directory.

Other options are specific to the GPTL libraries, like --enable-pmpi, which enables automatic profiling of application calls to MPI functions. This option is supported by the following code in *configure.ac*:

```
# Does the user want to turn on PMPI?
AC_MSG_CHECKING([whether PMPI is to be
enabled])
```

```
AC_ARG_ENABLE([pmpi],
[AS_HELP_STRING([--enable-pmpi],
              [build with PMPI capability])])
test "x$enable_pmpi" = xyes || enable_pmpi=no
AM_CONDITIONAL(ENABLE_PMPI, [test
x$enable_pmpi = xyes])
if test $enable_pmpi = yes; then
   AC_DEFINE([ENABLE_PMPI], [1], [enable
pmpi])
fi
AC_MSG_RESULT($enable_pmpi)
```

If --enable-pmpi is used, the library will automatically add MPI profiling. Automake conditional ENABLE_PMPI is set, and is used in the Makefile.am files to add a test if PMPI is enabled (from test/Makefile.am):

```
if ENABLE_PMPI
check_PROGRAMS += pmpi
TESTS += run_par_pmpi_test.sh
endif
```

The configure.ac code also sets a C preprocessor macro if ENABLE_PMPI is defined . This macro is output to a special header file, config.h, when the user runs the configure script. The config.h file contains many macros that describe the target build system. There are many macros that are provided by autoconf for all projects. Using the AC_DEFINE in configure.ac, we can add macros in support of the GPTL code, such as ENABLE_PMPI.

The config.h file must be included as the first header included in all library code and test code. This allows developers to control how code is built on systems with different capabilities.

3.2.2 Finding Libraries

Autoconf is good at finding libraries or header files. Here's how we can check for a library (from configure.ac):

```
# Check for pthread library.
AC_CHECK_LIB([pthread], [pthread_mutex_init])
if test
"x$ac_cv_lib_pthread_pthread_mutex_init" =
xyes; then
   AC_DEFINE([PTHREADS], [1],
      [pthreads library is present])
fi
```

3.3 Makefiles

Makefiles are generated when the user runs the configure script. The configure script runs many tests on the system, and can set automake conditionals to control how the makefiles are constructed. The automake conditionals are used in the Makefile.am files.

Each directory in the project gets a Makefile.am file. This file specifies what is built in that directory.

In most cases, Makefile.am files are simple. Here is the Makefile.am that builds the GPTL C library (including shared library):

```
libgptl_la_CPPFLAGS = -I$(top_srcdir)/include

# This is our output. The GPTL library.
lib_LTLIBRARIES = libgptl.la

# These are the source files.
libgptl_la_SOURCES = f_wrappers.c    \
getoverhead.c gptl.c gptl_papi.c     \
hashstats.c memstats.c memusage.c pmpi.c \
print_rusage.c pr_summary.c util.c
```

3.3.1 Target *make all*

The *all* target builds the library, but no tests are built.

3.3.2 Target *make check*

The *check* target first runs the *all* target, then builds and runs the tests. Tests are simply programs that use the library, and return 0 for success or any other value for failure.

If all tests build and pass, then *make check* returns 0, otherwise it fails.

3.3.3 Target *make install*

The *install* target first runs the *all* target, then installs the library in /usr/local, or elsewhere, if the user used the --prefix configure option. An *uninstall* target is also provided automatically.

Note that the *install* target does not run the tests.

3.3.4 Targets *make dist/distcheck*

The *dist* target builds a tarball with all files needed to build the GPTL C library. Users of the tarball do not have to have any autotools software installed, nor do they run autoreconf on their system (the autotools files, configure.ac and the Makefile.am files, are provided in the tarball for completeness, but the tarball also contains the created configure script, and the Makefile.in files that are converted to Makefiles when the configure script is run.)

The *distcheck* target first builds a distribution tarball, and then unpacks it and does a *make check*, and a *make distclean*, then ensures that everything is deleted. The *distcheck* target, as its name implies, allows you to check that your distribution tarball is complete and correct.

3.3.5 Target *make clean/distclean*

The *clean* target works as expected. The distclean target does an extra level of cleaning, returning the build to a state before the configure script was run.

4 FORTRAN LIBRARY

4.1 Configuration

As with the configuration of the C library, necessary libraries and headers are found when the user runs the configure script. One such library is the GPTL C library, which must be built and installed before the GPTL-Fortran library is built. This is done in configure.ac:

```
# Find the GPTL C library.
AC_CHECK_LIB([gptl], [GPTLinitialize], [],
                 [AC_MSG_ERROR([Can't find or
link to the GPTL C library.])])
```

As is good practice in configure scripts, error out if required supporting libraries or tools cannot be found.

4.2 Makefiles

The automake file to build the Fortran library is in the src subdirectory. It is slightly more complex than the one used to build the C library (src/Makefile.am):

```
libgptlf_la_FCFLAGS = -I$(top_srcdir)/include
if HAVE_PAPI
libgptlf_la_FCFLAGS += -DHAVE_PAPI
endif
if HAVE_MPI
libgptlf_la_FCFLAGS += -DHAVE_MPI
endif
# This is our output. The GPTL-fortran
# library.
lib_LTLIBRARIES = libgptlf.la
libgptlf_la_SOURCES = gptlf.F90 \
printmpistatussize.F90 \
process_namelist.F90
# Install these in the include directory.
include_HEADERS = gptl.mod
```

Note the .mod file, which will be installed in the include directory.

The Fortran build does not use the config.h file, but fortran 90 does support preprocessor ifdefs, and those are supported by setting macros with the -D option in FCFLAGS.

4.2.1 Standard Targets

As with the C library, standard targets are supported, including *all*, *check*, *clean*, *install*, *dist*, *distcheck*, and *uninstall*.

5 COMBINED DISTRIBUTION CONTAINING C AND FORTRAN LIBRARIES

The original GPTL distribution included both C and Fortran libraries. We have broken these into separate projects.

However, we would still like to have a combined distribution which includes both C and Fortran libraries. There is an easy way to accomplish this using autoconf.

We start by creating a new, third repo, which will hold the configuration files for the combined distribution.

5.1 Use of git Submodules

The new combined repository also offers an improvement in programmer workflow. Using git submodules we can achieve a combined development environment, while still maintaining separate C and Fortran library repositories.

We start by cloning the combined distribution project, and using the following commands to add submodules:

```
git submodule add
git@github.com:jmrosinski/GPTL.git
git submodule add
git@github.com:NOAA-GSD/GPTL-fortran.git
```

This creates directories GPTL and GPTL-fortran, which contain the contents of the C and Fortran library repositories.

Now the C and the Fortran libraries my be edited and tested as a unit. A separate git commit is required for each of the three repositories which have been changed. But the developer can make changes in either library, and then use the combined build to ensure that both libraries still work, and still work together.

5.2 Configuration

The configure.ac file for the combined distribution project is small. After initializing autoconf, automake, and libtool, the script launches the C and Fortran library configuration scripts.

An additional option, --enable-package-build, has been added to the Fortran library builds. Setting this option notifies the Fortran library build that this is a combined library build, so it can find the C library.

```
# Build the GPTL C library.
AC_CONFIG_SUBDIRS([GPTL])

# Add this arg for the fortran build, to tell
# it to use the C library we just built.
ac_configure_args="$ac_configure_args \
--enable-package-build"

# Build the GPTL Fortran library.
AC_CONFIG_SUBDIRS([GPTL-fortran])
```

```
AC_OUTPUT
```

5.3 Top-Level Makefile.am

The Makefile.am file for the combined release launches the build in each of the two subdirectories:

```
SUBDIRS = GPTL GPTL-fortran
```

This will cause first the C, and then the Fortran GPTL libraries to be built.

5.4 Fortran Configuration Changes

The Fortran test directory needs to link to the C library. In stand-alone Fortran library builds, the location of the C library is specified in CPPFLAGS and LDFLAGS, and the configure script adds the -lgptl to the link lines.

But with a combined build, the C library is not going to be available in that way, and is instead to be found in the GPTL/src directory.

To accommodate this, the following was added to the configure.ac:

```
AC_ARG_ENABLE([package-build],
  [AS_HELP_STRING([--enable-package-build],
    [Set internally for package builds, \
     should not be used by user.])])
test "x$enable_package_build" = xyes || \
enable_package_build=no
AM_CONDITIONAL([BUILD_PACKAGE], [test \
"x$enable_package_build" = xyes])

# Find the GPTL C library, unless this is a
# combined C/Fortran library build.
if test $enable_package_build = no; then
   AC_CHECK_LIB([gptl], [GPTLinitialize], [],\
      [AC_MSG_ERROR([Can't find or link to \
                     the GPTL C library.])])
fi
```

This adds the --enable-package-build option to the Fortran build. When used, this option tells the configure and Makefiles that the C library is located as part of the combined package, rather than already being installed.

5.5 Fortran Test Makefile.am Changes

Within the Fortran library Makefile.am, the follow change allows the build to find the C library, in combined library builds:

```
# For combined C/Fortran builds, find the C
# library.
if BUILD_PACKAGE
LDADD = ${top_builddir}/../GPTL/src/libgptl.la
endif
```

5.6 Installing the Combined Package

The combined package is built exactly like the individual libraries. The options given to configure will be passed to both C and Fortran builds. (Unknown configuration options are ignored.)

5.7 Use of git Submodules in Development

The use of git submodules requires that developers be able to check out and work with the GPTL-all project, which includes the GPTL (the C library) and GPTL-fortran (the Fortran library).

To work with the combined project, developers first do:

```
git clone --recursive \
https://github.com/NOAA-GSD/GPTL-all.git
```

This clones the top-level project into directory GPTL-all. In this directory are directories containing the C and Fortran libraries, which are cloned from their repositories. One git clone command, with the --recursive option, clones all three repositories.

Although this yields a very convenient way of checking out all three projects, developers must remain aware that three git projects are involved. Changes committed under the GPTL subdirectory will affect the C library repository, changes under the GPTL-fortran directory will affect the Fortran library repository, and changes in the GPTL-all directory will affect the GPTL-all repository.

When creating branches, programmers must understand which git project is involved. Creating a branch under the GPTL directory will create a branch on the C library repository, but not the GPTL-fortran and GPTL-all repositories. If work needs to be done on all three repositories for a feature, then three branches must be created, one on each repository.

More commonly, work will be isolated to either the C or the Fortran libraries. Developers will work in the GPTL or GPTL-fortran directories, creating branches in the C or Fortran library repositories, as needed.

To use branches, developers cd to the directory of the C or Fortran GPTL library, and use:

```
git checkout master
```

This will checkout the current master for the library. Using the usual commands, the developer can create branches:

```
git branch ejh_new_branch
git checkout ejh_new_branch
git push -u origin ejh_new_branch
```

6 RESULTS

6.1 Greater Portability

The previous build system for GPTL supported a variety of build platforms, but the approach of specifying compilers and flags in the makefile is fundamentally limited to those that have been previously encountered by the development team.

The new build system is capable of building on many more systems, and leaves the selection of compiler and flags in the hands of the end user. As a result, it will be able to build with compilers and compiler options that are not known to the GPTL developers. Even future compilers and compiler options can be supported.

6.2 Additional Features

The autotools packages provide some build and configure features that were not supported in the previous build system:

- Shared library builds.
- Standard make targets dist, distcheck, uninstall.
- Standardised help setting specific configure options with *configure --help*.

6.3 Reduction in Complexity

Significant reduction in complexity has been achieved. The total number of files required to support the build has dropped from 30 to 14. The number of lines of code (contents of make and configure files) has been reduced by almost an order of magnitude.

build system	**total files**	**lines of code**
legacy	30	4807
autotools (combined C/Fortran build)	14	593

Reduction in complexity reduces maintenance costs, enables changes to be made quickly, and reduces the chance of serious errors in the build system.

6.4 Friendly to Linux Package Management Systems

Using autotools, and separating the C and Fortran libraries, allows the GPTL C and Fortran libraries to work within the standard Linux package management systems. This will allow end users to install the library using yum/apt package management tools, which is the easiest and simplest way to install software on a Linux system.

6.5 Developer and User Workflow

6.5.1 User Workflow

The end-user of the GPTL libraries does not need to know anything about the autotools: autoconf, automake and libtool. The end user does not even have to have these tools installed on their system.

The GPTL libraries are installed from the distribution tarball. This contains the configure script, and the Makefile.in files that it uses to construct the Makefiles on the build system.

To install the libraries, the user unpacks the tarball, then runs *configure* and *make install*.

6.5.2 Developer Workflow

Developers of GPTL will start from the git repository, instead of a distribution tarball. The repository does not (and should not) contain the intermediate files generated by autoconf/automake. Instead, the repository contains the source files that autoconf and automake need to generate the build system.

After cloning the repository, developers must run *autoreconf -i* to build the build system. This command must also be run whenever the configure.ac file is changed, to ensure that a new configure script is generated.

When running autoreconf -i, many intermediate files will be generated.These intermediate files must not be committed to the repository. They will change slowly over time, as the autotools packages release new versions. Using autoreconf -i on the clean repository clone will always result in a correct and consistent set of intermediate files. Committing any of these files to the repository may cause future subtle bugs if the autotool that generates or needs that intermediate file is upgraded. In such a case, the file that has been committed to the repository may be incorrect for the new version of the tool.

Developers must never edit the intermediate files. Developers interact with the build system through the configure.ac file (one per project) and the Makefile.am files (one per subdirectory). These are the only build system files that are ever edited by the developer.

7 ACKNOWLEDGMENTS

Jim Rosinski of NCAR is the author of the GPTL. His assistance in the transition to the autotools build system was invaluable.

8 REFERENCES

GPTL GitHub Site, https://github.com/jmrosinski/GPTL

GPTL-Fortran GitHub Site, https://github.com/NOAA-GSD/GPTL-fortran

GPTL-all GitHub Site, https://github.com/NOAA-GSD/GPTL-all