CactusBuildSytemTour

June 10, 2024

1 Building Cactus on a new computer

1.1 Cactus build system

Cactus's build system is using `make` as the top-level interface, combined with code being generate from Cactus's `ccl` files that implement a domain specific language to describe the Cactus thorns (modules). To support different host operating systems, for example Linux on clusters and macOS on personal laptops, as well as different compilers, for example the GNU compiler collection, Intel's compiler, or LLVM / clang, the build system uses makefile variables. These are supplied via option list files, with those for public clusters being collected by tools like Simulation Factory or batchtools. A thornlist file is is used by the build system to decide which thorns to include in the compiled executable, with each thorn existing in its own subdirectory of the Cactus source tree.

Once compiled a simulation is the submitted to the cluster's queueing system for execution using a number of MPI ranks and reserving core for each MPI rank.

Tools like Simulation Factory and GetComponents try to assist in providing option lists and premade files to ease submitting simulations on clusters.

At the end of this presentation I hope you will be able to understand a bit what goes on in each step and this will help understand how to interpret error messages reported when building Cactus.

1.2 How to use this tutorial

This tutorial will not show all the possible ways to compile and use Cactus, nor does it aim to replace Cactus' and Simfactory's documentation. For those please consult the Cactus UsersGuide, the Configuring a new machine Einstein Toolkit wiki and / or the Simulation Factory User Guide, the Setting up Simfactory on a new machine ET Seminar including its notebooks and especially chapter "Configuration Options" of the Cactus UsersGuide.

Instead it provides a guide tour of how things work "under the hood" that will let you understand a bit better exactly what Cactus complains about when it fails to compile or a simulation fails to start. As such: your milage may vary!

1.3 Cactus directory structure

Cactus organizes thorns in thematically connected `arrangements`, like the `CactusNumerical` arrangement containing thorns that deal with numerical metrics. Only a single level nesting is allowed, so there are no arrangements of `arrangements`. All arrangements exist as subdirectories of the main `Cactus` directory.

Each thorn provides

- `configuration.ccl` that describes the capabilities that the thorn requires and provides (section D2.5 of UsersGuide)
- `interface.ccl` that describes the variables that the thorn uses (section D2.2 of UsersGuide)
- `param.ccl` that describes the runtime parameters that the thorn supports (section D2.3 of UsersGuide)
- `schedule.ccl` specifying which routines are to be called (section D2.4 of UsersGuide)
- `make.code.defn` (in the simplest case) list all the source files to be compiled (section C1.2.5 of UsersGuide)

Cactus' `Makefile` defines compilation targets that are using to compile the code, build documentation or clean up build artifacts.

The `configs` directory contains processed source code files, object files and processed copies of other input files.

```
Cactus
 +- Makefile
 +- arrangements
 |    +- Tests
 |         +- HeatEqun
 |         |   +- configuration.ccl
 |         |   +- interface.ccl
 |         |   +- param.ccl
 |         |   +- schedule.ccl
 |         |   +- make.code.defn
 |         |   +- src
 |         |       +- init.c
 |         |       +- evolve.c
 |         +- WaveEqun
 |             +- configuration.ccl
 |             +- interface.ccl
 |             +- param.ccl
 |             +- schedule.ccl
 |             +- make.code.defn
 |             +- src
 |                 +- init.c
 |                 +- evolve.c
 +- configs
     +- sim
         +- ThornList
         +- config-info
         +- config-data
             +- make.config.defn
         +- build
             +- HeatEqn
                 +- init.c
                 +- init.c.d
                 +- init.c.o
```

1.4 Downloading Cactus

The usual method is to use `GetComponents` which takes as input a thornlist with extra commands specific in ''components retrieval language'' (CRL). Here instead we will do things by hand to show the assembly.

[ ]:
```
%%bash

git clone --depth 1 https://bitbucket.org/cactuscode/cactus.git CactusBare
```

```
Cloning into 'CactusBare'...
remote: Enumerating objects: 391, done.
remote: Counting objects: 100% (391/391), done.
remote: Compressing objects: 100% (366/366), done.
remote: Total 391 (delta 38), reused 157 (delta 13), pack-reused 0 (from 0)
Receiving objects: 100% (391/391), 2.40 MiB | 7.07 MiB/s, done.
Resolving deltas: 100% (38/38), done.
```

[ ]:
```
cd CactusBare
```

[ ]:
```
%%bash

ls -F
```

```
CONTRIBUTORS  COPYRIGHT  Makefile  README.md  doc/  lib/  src/
```

[ ]:
```
%%bash

mkdir arrangements
```

[ ]:
```
cd arrangements
```

[ ]:
```
%%bash

git clone https://bitbucket.org/cactuscode/cactuswave CactusWave
```

```
Cloning into 'CactusWave'...
remote: Enumerating objects: 232, done.
remote: Counting objects: 100% (232/232), done.
remote: Compressing objects: 100% (204/204), done.
remote: Total 232 (delta 83), reused 85 (delta 16), pack-reused 0 (from 0)
Receiving objects: 100% (232/232), 118.69 KiB | 2.70 MiB/s, done.
Resolving deltas: 100% (83/83), done.
```

We will also need some of Cactus' infrastructure thorns:

[ ]:
```
%%bash

git clone https://bitbucket.org/cactuscode/cactusbase CactusBase
```

```
Cloning into 'CactusBase'...
```

```
remote: Enumerating objects: 8163, done.
remote: Counting objects: 100% (8163/8163), done.
remote: Compressing objects: 100% (2690/2690), done.
remote: Total 8163 (delta 6043), reused 7193 (delta 5322), pack-reused 0 (from 0)
Receiving objects: 100% (8163/8163), 1.53 MiB | 9.00 MiB/s, done.
Resolving deltas: 100% (6043/6043), done.
```

[ ]:
```
%%bash

git clone https://bitbucket.org/cactuscode/cactuspugh CactusPUGH
```

```
Cloning into 'CactusPUGH'...remote: Enumerating objects: 4769, done.
remote: Counting objects: 100% (4769/4769), done.
remote: Compressing objects: 100% (1561/1561), done.
remote: Total 4769 (delta 3103), reused 4571 (delta 2975), pack-reused 0 (from 0)
Receiving objects: 100% (4769/4769), 939.56 KiB | 8.62 MiB/s, done.
Resolving deltas: 100% (3103/3103), done.
```

[ ]:
```
%%bash

mkdir ExternalLibraries
git clone --depth=1 https://github.com/einsteintoolkit/ExternalLibraries-MPI␣
↪ExternalLibraries/MPI
```

[ ]:
```
cd ..
```

1.5 Makefile targets

At this point we have manually assembled a very basic Cactus checkout consisting of only the flesh the CactusWave arrangment. We are now ready to compile Cactus.

Cactus uses a number of makefile targets to control what is actually built, e.g., the executable, or documention or also phony targets used to clean up in case things go sideways.

The interesting targets are (details are in UsersGuide section B2.4):

- `<configuration>-config` which (re-)creates a configuration
- `<configuration>-clean` which deletes all compiled object and library files
- `<configuration>-realclean` which additinally deletes all dependency tracking files (use if things are pear-shaped)
- `<configuration>-cleandeps` which removes only the dependency tracking files (use if you eg do an OS update)
- `<configuration>-rebuild` re-runs CST (re-interprets ccl files), mostly useful when developing ExternalLibraries
- `<configuration>-build` compile only a single thorn and its dependencies (see below)
- `<configuration>-utils` compile the utilities included with the thorns

Additional informations passed to `make` as command line options

- `options` the name of an option list
- `THORNLIST` the thornlist file to compile

- `BUILDLIST` a space separated list of thorn to compile for the `<configuration>-build` target

We will require a thornlist and an option list to proceed. Simulation Factory comes with a database of option lists and the Cactus `utils/Scripts/MakeThornList` can be used to create a thornlist suitable for a given parameter file based on a template thornlist. Finally the tool `lib/sbin/ThornList.pl` can create a list of all the available thorns. In our case we will use a custom thornlist.

[ ]:
```
%%bash

cat <<"#EOF" >wavethorns.th
CactusBase/Boundary
CactusBase/CartGrid3D
CactusBase/CoordBase
CactusBase/Fortran
CactusBase/IOASCII
CactusBase/IOBasic
CactusBase/IOUtil
CactusBase/SymBase
CactusBase/Time
CactusPUGH/PUGH
CactusPUGH/PUGHReduce
CactusPUGH/PUGHSlab
CactusWave/IDScalarWave
CactusWave/IDScalarWaveC
CactusWave/WaveToyC
CactusWave/WaveToyF90
ExternalLibraries/MPI
#EOF
```

Next we need a very simple option list. Usually, in particular on clusters, these would be provided by a database of option lists in Simulation Factory. However Cactus actually has a built in set of defaults that it will use to fill in any missing option.

This functionality is implemented by the `known-architectures` files in `lib/make/known-architectures/`:

```
aix     darwin   hp-ux  linux-gnu      linux-gnulibc1  solaris  unicosmp           xt4-cray-ca
bgl     freebsd  irix   linux-gnuaout  openbsd         superux  uxpv4.1_ES         xt4-cray-li
cygwin  hiuxwe2  linux  linux-gnueabi  osf             unicos   xt3-cray-catamount
```

which are bash scripts that try to infere options, e.g., for OpenMP, based on the operating system and copmiler used.

For now we will use a very minimal set of options.

[ ]:
```
%%bash

cat <<"#EOF" >options.cfg
CC=gcc
```

```
CXX=g++
F90=gfortran
FPP=cpp
FPPFLAGS=-traditional
#EOF
```

There are actually 3 (common) ways to pass options to Cactus, in order of precedence:

- environment variables, e.g., `export DEBUG=yes`
- the option list, `make sim-config options=options.cfg` (and yes, `options` itself is an option)
- at the command line `make sim-config VERBOSE=yes` (this actually how we pass the `options` option)

Of these the environment option one can be surprising on clusters where an environment variable set by the `module` command may conflict with a Cactus option.

ExternalLibraries: Options passed to ExternalLibraries (e.g., HDF5, GSL) are an exception to this. For them environment options are only acted on during the CST (first stage of compiling) stage and override otion list values. This can and has cause problems on clusters.

1.6 Configuration options

[ ]:
```
%%bash

# if this hangs for you and does not react to input, add <<<"yes" at the end
make wave-config options=options.cfg THORNLIST=wavethorns.th
```

```
Setup configuration wave (yes)?
Setting up new configuration wave
Completely new cactus build.  Creating config database
Creating new configuration wave.
Using configuration options from configure line
  Setting THORNLIST to 'wavethorns.th'
End of options from configure line
Adding configuration options from 'options.cfg'...
  Setting CC to 'gcc'
  Setting CXX to 'g++'
  Setting F90 to 'gfortran'
  Setting FPP to 'cpp'
  Setting FPPFLAGS to '-traditional'
End of options from 'options.cfg'.
creating cache ./config.cache
checking host system type... x86_64-pc-linux-gnu
checking for mawk... mawk
Setting FPP to /lib/cpp
Setting FPPFLAGS to -traditional
checking whether make sets ${MAKE}... yes
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
```

```
checking whether we are using GNU C... yes
checking whether the C++ compiler (g++  ) works... yes
checking whether the C++ compiler (g++  ) is a cross-compiler... no
checking whether we are using GNU C++... yes
checking for nvcc... no
[...]
checking whether linker supports --whole-archive... yes
checking whether linker supports -filelist... no
creating cctk_Archdefs.h
creating make.arch.defn
creating cctk_Extradefs.h
creating make.extra.defn
updating cache ./config.cache
creating ./config.status
creating make.config.defn
creating make.config.deps
creating make.config.rules
creating cctk_Config.h
Determining number of fortran underscores...
Compiling test file with gfortran  ...
Lowercase - One trailing underscore
Compiling test file with gfortran  ...
Lowercase - One trailing underscore
Use make wave to build the configuration.
```

Cactus created a new configuration `wave` in `configs/wave` using the option list passed to make. It also reports having created a number of files and how to build the configuration.

Let's take look at the files present:

ls -F configs/wave/

```
ThornList  build/  config-data/  config-info  lib/  scratch/
```

Of these `config-info` is essentially a copy of the option list we passed in along with any options passed on the command line (except for those for ExternalLibraries):

[ ]:
```
%%bash

cat configs/wave/config-info
```

```
# CONFIGURATION  : wave
# CONFIG-DATE    : Fri Jun  7 11:47:49 2024 (GMT)
# CONFIG-HOST    : slurmjupyter
# CONFIG-STATUS  : 0
# CONFIG-OPTIONS :
CC=gcc
CXX=g++
F90=gfortran
FPP=cpp
FPPFLAGS=-traditional
```

```
THORNLIST=wavethorns.th
```

The complete set of all options used (again except ExternalLibraries) is in `configs/wave/config-data/make.config.defn`, one of the files that make created.

[ ]:
```
%%bash

cat configs/wave/config-data/make.config.defn
```

```
# /*@@
#   @file    make.config.defn.in
#   @date    Sun Jan 17 22:33:51 1999
#   @author  Tom Goodale
#   @desc
#            Make definition file for a configuration
#            Defines the compilers, etc.
#
#            Should be run through autoconf to produce make.config.defn
#   @enddesc
#   @version $Header$
# @@*/

# Include stuff defined by optional extra arrangements
include $(TOP)/config-data/make.extra.defn

# Include all required external libraries from configuration
-include $(TOP)/bindings/Configuration/make.link


# Compiler/executable info
export SHELL       = /bin/sh
export MKDIR       = mkdir
export CC          = gcc
export CXX         = g++
export CUCC        =
export F90         = gfortran
export F77         = gfortran
export CPP         = /usr/bin/cpp
export FPP         = /lib/cpp
export LD          = g++
[...]
# Are we building with all warnings switched on ?
# This option may be overwritten at compile time.
ifeq ($(WARN),)
  export CCTK_WARN_MODE = no
else
  export CCTK_WARN_MODE = $(WARN)
endif
```

```
ifeq ($(strip $(CCTK_WARN_MODE)),yes)
  CPPFLAGS  += $(CPP_WARN_FLAGS)
  FPPFLAGS  += $(FPP_WARN_FLAGS)
  CFLAGS    += $(C_WARN_FLAGS)
  CXXFLAGS  += $(CXX_WARN_FLAGS)
  CUCCFLAGS += $(CUCC_WARN_FLAGS)
  F77FLAGS  += $(F77_WARN_FLAGS)
  F90FLAGS  += $(F90_WARN_FLAGS)
endif

# Stuff specific to an architecture.
include $(TOP)/config-data/make.arch.defn
```

ExternalLibraries and how they interact with the build system if often a source of problems, not least because it is not obvious exaxtly which options they support.

The UersGuide chapter B2.2 describes how a generic ExternalLibrary is expected to behave, yet, since they all grew organically and were not designed at the same time, some (most of of) ExternalLibraries deviate from this baheviour in various ways.

On the technical level all ExternalLibraries use Cactus' ''capabilities'' functionality and `configuration.ccl` to declare the option they support to the Cactus build system and report any make options they require as output of a custom configuration script.

For the `MPI` ExternalLibrary has a `configuration.ccl` file that looks like this:

[ ]:
```
%%bash

cat arrangements/ExternalLibraries/MPI/configuration.ccl
```

```
# Configuration definitions for thorn MPI

PROVIDES MPI
{
  SCRIPT src/detect.pl
  LANG perl
  OPTIONS MPI MPI_DIR MPI_INC_DIRS MPI_LIB_DIRS MPI_LIBS MPI_INSTALL_DIR HWLOC_DIR
}

# Pass configuration options to build script
REQUIRES MPI

OPTIONAL hwloc
{
}
```

This indicates that ExternalLibraries/MPI `PROVIDES` the capability `MPI` and supports options

- `MPI`
- `MPI_DIR`
- `MPI_INC_DIRS`

- MPI_LIB_DIRS
- MPI_LIBS
- MPI_INSTALL_DIR
- HWLOC_DIR

where the last one (HWLOC_DIR) is actually from ExternalLibraries/hwloc that MPI can use. This one may not actually be require here to make MPI work.

MPI can optionally (if present at compile time) use hwloc, but will work without.

configuration.ccl and thus MPI's options are processed during the "CST" stage of the Cactus build when all the ccl files are being parsed. This happens *after* the make wave-config stage, and is why ExternalLibrary options behave differently from "core" Cactus option list options.

Let's take a look what happens.

[ ]:
```
%%bash

make wave-build BUILDLIST=MPI
```

```
Cactus - version: 4.15.0
Building thorns 'MPI' of configuration wave
Reconfiguring thorns
Reading ThornList...
Parsing configuration files...
   Boundary
            Provides:          Boundary
[...]
   WaveToyF90
           11 schedule blocks.
Running any thorn-provided configuration scripts...

********************************************************************************
Running configuration script for thorn FORTRAN:
Found a traditional Fortran cpp
Finished running configuration script for thorn FORTRAN.

********************************************************************************
Running configuration script for thorn MPI:
MPI selected, but MPI_DIR is not set. Computing settings...
Found MPI compiler wrapper at /usr/bin/mpic++!
Successfully configured MPI.
Finished running configuration script for thorn MPI.
Checking consistency...
Creating Thorn-Flesh bindings...
   Creating implementation bindings...
   Creating parameter bindings...
   Creating variable bindings...
   Creating schedule bindings...
   Creating function bindings...
```

```
   Generating arguments...
CST finished.
Checking status of thorn MPI
```

So the configuation script for MPI was ran, it found a `mpic++` compiler in `/usr/bin/mpic++` and used it to determine the correct values for `MPI_DIR` etc. that I had not specified in my option list.

Where did those values go though?

They are not in `configs/wave/config-info` or even `configs/wave/config-data/make.config.defn`.

Instead there is now a `bindings` directory:

[ ]:
```
%%bash

ls -F configs/wave
```

ThornList bindings/ build/ config-data/ config-info lib/ piraha/ scratch/

and inside that a sub-directory `configs/wave/bindings/Configuration/Capabilities`:

[ ]:
```
%%bash

ls -F configs/wave/bindings/Configuration/Capabilities/
```

```
cctki_BOUNDARY.h    cctki_MPI.h           make.CARTGRID3D.deps  make.IOUTIL.defn  make.PUGH.dep
cctki_CARTGRID3D.h  cctki_PUGH.h          make.COORDBASE.defn   make.IOUTIL.deps
cctki_COORDBASE.h   make.BOUNDARY.defn    make.COORDBASE.deps   make.MPI.defn
cctki_FORTRAN.h     make.BOUNDARY.deps    make.FORTRAN.defn     make.MPI.deps
cctki_IOUTIL.h      make.CARTGRID3D.defn  make.FORTRAN.deps     make.PUGH.defn
```

and `make.MPI.defn` is a makefile fragment that contains the actual values used by MPI:

[ ]:
```
%%bash

cat configs/wave/bindings/Configuration/Capabilities/make.MPI.defn
```

```
INC_DIRS +=  $(MPI_INC_DIRS)
INC_DIRS_F +=  $(MPI_LIB_DIRS)
MPI_BUILD       =
MPI_INSTALL_DIR =
HWLOC_DIR       =
CCTK_MPI     = 1
MPI_DIR      = /usr
MPI_INC_DIRS = /usr/lib/x86_64-linux-gnu/openmpi/include/openmpi /usr/lib/x86_64-linux-gnu/oper
MPI_LIB_DIRS = /usr/lib/x86_64-linux-gnu/openmpi/lib
MPI_LIBS     = mpi_cxx mpi
HAVE_CAPABILITY_MPI = 1
```

1.7 Compiling thorn source code

So far we have not actually compiled any thorn source code. In this section, we will dig down a bit to understand the steps involved. Let's look at what happens if we compile the `IDScalarWaveC`

thorn:

[ ]:
```
%%bash

make wave-build BUILDLIST=IDScalarWaveC
```

```
Cactus - version: 4.15.0
Building thorns 'IDScalarWaveC' of configuration wave
Checking status of thorn IDScalarWaveC
COMPILING CactusWave/IDScalarWaveC/src/InitialData.c
COMPILING CactusWave/IDScalarWaveC/src/CheckParameters.c
COMPILING configs/wave/bindings/build/IDScalarWaveC/cctk_ThornBindings.c
Creating /home/rhaas80/CactusBare/configs/wave/lib/libthorn_IDScalarWaveC.a
```

Which is actually quite short. The thorn contains only two source code files:

- `InitialData.c`
- `CheckParameters.c`

located in its source directory `arrangements/CactusWave/IDScalarWaveC/src`.

There is also an auto-generated file `cctk_ThornBindings.c` which handles registration of a thorns grid-variables and scheduled function with the Cactus flesh. It was created as part of the earlier `CST` stage of the build.

As is usual for a thorn, the source files to compile are all listed in `make.code.defn`:

[ ]:
```
%%bash

cat arrangements/CactusWave/IDScalarWaveC/src/make.code.defn
```

```
# Main make.code.defn file for thorn IDScalarWave
# $Header$

# Source files in this directory
SRCS = InitialData.c CheckParameters.c

# Subdirectories containing source files
SUBDIRS =
```

and there are no subdirectories with additional source code files.

Since we did not really see what was going on in the terse output, we will redo compiling one file with `VERBOSE=yes` set.

[ ]:
```
%%bash

touch arrangements/CactusWave/IDScalarWaveC/src/InitialData.c
make wave-build BUILDLIST=IDScalarWaveC VERBOSE=yes
```

This (unsurprisingly) produces much more output to screen, and shows exactly which commands `make` executes.

```
[...]
Cactus - version: 4.15.0if test "xIDScalarWaveC" = "x"; then \
[...]
  echo Building thorns \'IDScalarWaveC\' of configuration wave; \
  cd /home/rhaas80/CactusBare/configs/wave; \
  make -f /home/rhaas80/CactusBare/lib/make/make.configuration TOP=/home/rhaas80/CactusBare/co
fi;
Building thorns 'IDScalarWaveC' of configuration wave
make[1]: Entering directory '/home/rhaas80/CactusBare/configs/wave'
Checking status of thorn IDScalarWaveC
--------------------------------------------------------------------------
make[2]: Entering directory '/home/rhaas80/CactusBare/configs/wave/build/IDScalarWaveC'
if [ ! -d ./ ] ; then mkdir -p ./ ; fi
cd ./ ; make CCTK_TARGET=make.checked TOP=/home/rhaas80/CactusBare/configs/wave CONFIG=/home/r
make[3]: Entering directory '/home/rhaas80/CactusBare/configs/wave/build/IDScalarWaveC'
gcc -E -M   -pipe -std=gnu99 -O3 /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWave
perl -pi -e 's{^\s*\QInitialData.o\E\s*:}{InitialData.c.o InitialData.c.d:}; s{\s+\S*[/\\](CPa
1717767988.34154 Preprocessing /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/
{ if test no = 'yes'; then echo '#line 1 "'/home/rhaas80/CactusBare/arrangements/CactusWave/ID
1717767988.37144 Compiling /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src/
current_wd=`pwd` ; cd /home/rhaas80/CactusBare/configs/wave/scratch ; gcc   -pipe -std=gnu99 -
1717767988.58111 Postprocessing /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC
--------------------------------------------------------------------------
echo "" > make.checked
make[3]: Leaving directory '/home/rhaas80/CactusBare/configs/wave/build/IDScalarWaveC'
[...]
make[1]: Leaving directory '/home/rhaas80/CactusBare/configs/wave'
```

There's two interesting lines in this mess.

```
1717767988.34154 Preprocessing /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/
{
  if test no = 'yes'; then
    echo '#line 1 "'/home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src/Initial
  fi;
  cat /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src/InitialData.c;
} | perl -s /home/rhaas80/CactusBare/lib/sbin/c_file_processor.pl \
        -line_directives=no \
        -source_file_name=/home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src/
        /home/rhaas80/CactusBare/configs/wave/config-data \
> InitialData.c
```

and

```
1717767988.37144 Compiling /home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src/
current_wd=`pwd` ;
cd /home/rhaas80/CactusBare/configs/wave/scratch ;
gcc   -pipe -std=gnu99 -O3 -c -o $current_wd/InitialData.c.o \
      $current_wd/InitialData.c \
      -I"/home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src" \
```

```
-I"/home/rhaas80/CactusBare/configs/wave/config-data" \
-I"/home/rhaas80/CactusBare/configs/wave/bindings/include" \
-I"/home/rhaas80/CactusBare/src/include" -I"/home/rhaas80/CactusBare/arrangements" \
-I"/home/rhaas80/CactusBare/configs/wave/bindings/Configuration/Thorns" \
-I"/home/rhaas80/CactusBare/configs/wave/bindings/include/IDScalarWaveC" \
-I"/home/rhaas80/CactusBare/arrangements/CactusWave/IDScalarWaveC/src" \
-I"/home/rhaas80/CactusBare/configs/wave/bindings/include/IDScalarWaveC"  \
-DCCODE
```

which I have broken down into individual shell commands. Interstingly what is copiled by `gcc` is not the source file in `arrangements/CactusWave/IDScalarWaveC/src/InitialData.c` instead a file in `current_wd` which ends up being `configs/wave/build/IDScalarWaveC`.

This is due to Cactus preprocessing both C and Fortran source code, in C/C++'s case only to support the `CCTK_FNAME` "macro", but in Fortran's case to expand `DECLARE_CCTK_FOO` and other preprocessors macros. So error messages produced by the compiler will refer to the "wrong" source file.

For C/C++ files not using `CCTK_FNAME` there are (usually) no differences, so at least line numbers match, not so for Fortran files.

diff -suw arrangements/CactusWave/IDScalarWaveC/src/InitialData.c configs/wave/build/IDScalarWaveC/InitialData.c

```
Files arrangements/CactusWave/IDScalarWaveC/src/InitialData.c and configs/wave/build/IDScalarWa
```

For Fortran code there are large differences due to the `CCTK_ARGUMENTS` and `DECLARE_CCTK_FOO` macros being expanded.

[ ]:
```
%%bash

make wave-build BUILDLIST=IDScalarWave
```

```
Cactus - version: 4.15.0
Building thorns 'IDScalarWave' of configuration wave
Checking status of thorn IDScalarWave
COMPILING CactusWave/IDScalarWave/src/InitialData.F77
COMPILING CactusWave/IDScalarWave/src/CheckParameters.F77
COMPILING configs/wave/bindings/build/IDScalarWave/cctk_ThornBindings.c
Creating /home/rhaas80/CactusBare/configs/wave/lib/libthorn_IDScalarWave.a
```

[ ]:
```
%%bash

diff -suw arrangements/CactusWave/IDScalarWave/src/InitialData.F77 configs/wave/
  ↪build/IDScalarWave/InitialData.f | more
```

```
--- arrangements/CactusWave/IDScalarWave/src/InitialData.F77    2024-06-07 11:22:13.143378415 +
+++ configs/wave/build/IDScalarWave/InitialData.f       2024-06-07 14:21:36.640103919 +0000
@@ -1,60 +1,1104 @@
[...]
-      subroutine IDScalarWave_InitialData(CCTK_ARGUMENTS)
[...]
```

```
+      subroutine IDScalarWave_InitialData(cctk_dim,cctk_gsh,cctk_lsh,cct
+     &k_lbnd,cctk_ubnd,cctk_level,cctk_patch,cctk_npatches,cctk_componen
+     &t,cctk_tile_min,cctk_tile_max,cctk_ash,cctk_alignment,cctk_alignme
+     &nt_offset,cctk_from,cctk_to,cctk_bbox,cctk_delta_time,cctk_time,cc
+     &tk_delta_space,cctk_origin_space,cctk_levfac,cctk_levoff,cctk_levo
+     &ffdenom,cctk_timefac,cctk_convlevel,cctk_convfac,cctk_nghostzones,
+     &cctk_iteration,cctkGH,cctk_ash1,cctk_ash2,cctk_ash3, coarse_dx,coa
+     &rse_dy,coarse_dz,phi,phi_p,phi_p_p,r,x,y,z)

       implicit none

-      DECLARE_CCTK_ARGUMENTS
-      DECLARE_CCTK_PARAMETERS
-      DECLARE_CCTK_FUNCTIONS

-      CCTK_REAL  one
+      INTEGER cctk_dim
+      integer, parameter :: cctki_use_cctk_dim = kind(cctk_dim)
+      INTEGER cctk_gsh (cctk_dim)
+      integer, parameter :: cctki_use_cctk_gsh = kind(cctk_gsh)
[...]
```

In this I have actually remove most changes for the sake of brevity. It is clear though that much more is going on than for just C code. This changes line numbers so that, unless one takes measures, line numbers in Fortran error messages are off by many hundreds of lines even for simple thorns. Further since this is the file actually compiled, using a debugger like `gdb` also shows *this* source code and not the one in `arrangements`.

Aside: Cactus, and compilers, offer workarounds for this issue by adding special instructions to the source file that tell the compiler which file and line number to report. In Cactus this done via the

```
C_LINE_DIRECTIVES = yes
F_LINE_DIRECTIVES = yes
```

option list options.

Further Aside: Preprocessing of Fortran files is not done for files with extension of `.f`, `.f77`, or `.f90` (details in the UsersGuide C1.2.4), though I have never seen that done in Cactus.

1.8 Finish compiling the Cactus executable

[ ]:
```
%%bash

make -j2 wave
```

```
Cactus - version: 4.15.0
Building configuration wave
Checking status of thorn Boundary
Checking status of thorn CoordBase
```

```
COMPILING CactusBase/CoordBase/src/Domain.c
[...]
Creating cactus_wave in /home/rhaas80/CactusBare/exe from CactusBase/Boundary CactusBase/CartGi
Done creating cactus_wave.
All done !
```

This produces the main Cactus executable `exe/cactus_wave`:

[ ]:
```
%%bash

ls -f exe
```

```
cactus_wave  ..  .
```

If you are used to Simulation Facory then you may wonder why there are no utilities being built. This is because there is a specific target `wave-utils` for building them.

[ ]:
```
%%bash

make wave-utils
```

```
Building utilities for wave
Copying mpirun from /usr/bin/mpirun to /home/rhaas80/CactusBare/exe/wave
Copying mpirun.openmpi from /usr/bin/mpirun.openmpi to /home/rhaas80/CactusBare/exe/wave
Copying ompi-clean from /usr/bin/ompi-clean to /home/rhaas80/CactusBare/exe/wave
Copying ompi-server from /usr/bin/ompi-server to /home/rhaas80/CactusBare/exe/wave
Copying ompi_info from /usr/bin/ompi_info to /home/rhaas80/CactusBare/exe/wave
```

which produces (copies in this case of using a pre-compiled MPI thorn) utilies in `exe/wave`:

[ ]:
```
%%bash
ls -fR exe
```

```
exe:
cactus_wave  ..  .  wave

exe/wave:
ompi_info  mpirun.openmpi  ..  mpirun  ompi-clean  .  ompi-server
```