KGEN User's Guide for KGEN ver. 0.5.3

0. Changes from KGEN ver. 0.5.0

0.1 User Interface

- "--skip-intrinsic" and "--noskip-intrinsic" flags are discarded. Instead, please use "--intrinsic skip" and "--intrinsic noskip" each
- The syntax of numbers in "--invocation" flag changed. Please use colon(":") instead of comma(",") as a delimiter of numbers
- "--source" flag is added to inform KGEN with the Fortran source format.
- "common" section is renamed to "namepath" in exclusion INI file for "-e" flag
- The syntax of "namepath" is changed. Please see the section 2.3.1 for details
- Several actions are added in "namepath" section of exclusion INI file for "-e" flag. Please see the section 2.3.3 for details

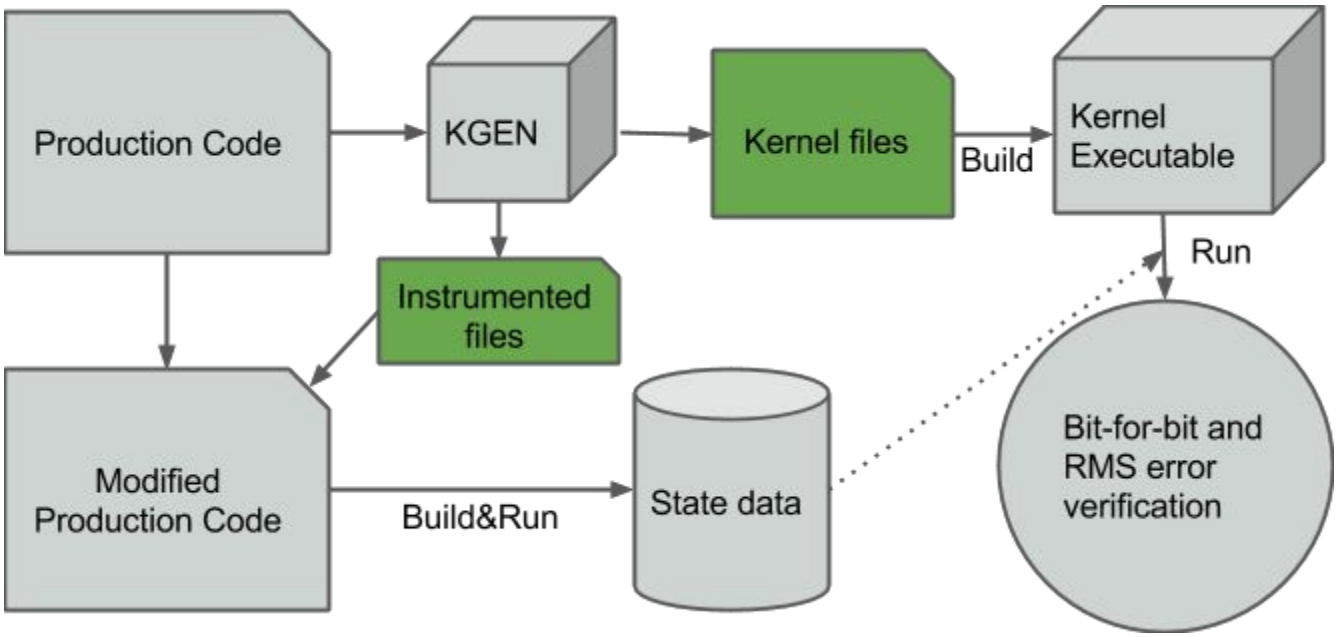
0.2 Major Improvements

- Better pruning of external library through additional exclusion actions.
- Support of analyzing Fortran External Subprograms

1. Introduction

KGEN is a software tool that extracts a Fortran subprogram from a larger software and make it as a stand-alone application, aka kernel. It also saves input data to and output data from the extracted part of source code that are used in the kernel for verification purpose.

General workflow for using KGEN is shown in following figure. User executes KGEN in command line with arguments that specify which subprogram to be extracted and where is a call site to the subprogram. As an initial step of kernel generation, KGEN creates kernel files as well as modified source file(s). The modified source files are used to generate input & output data. User replaces original source files with these modified source files and run the modified production code. Once input & output data are generated successfully, a kernel can be executed as a stand-alone application and its outcome from execution is automatically verified against the generated input & output data.



In practice, user may adopt iterative workflow in using KGEN. User first provides KGEN with minimum information and tries to generate a kernel. If it fails, in return, KGEN may provide an user with what information is missed. In addition, KGEN tries to create a kernel even though it can not find all required information and user may investigate the generated kernel further on missed information. Then user may re-try KGEN by providing additional information, and repeat the steps.

KGEN is being developed actively as of writing this document. It would be very helpful if you can share your experience on using KGEN by sending relevant information to "kgen@ucar.edu".

2. How to use

2.1 Installation

KGEN is written in Python as an extension of a Fortran parser distributed in F2PY python package.

2.1.1 Requirements

KGEN is developed mainly in Python 2.6.6, which is a minimum required for using KGEN. A Fortran parser of F2PY is extensively used in KGEN and is a part of KGEN distribution. Therefore, there is no additional work needed for installing the F2PY Fortran Parser.

Once you have unzipped KGEN distribution file, then you can run KGEN as below:

python $PATH_TO_KGEN/kgen.py --help

If you are an user of Yellowstone cluster of NCAR, you can use a pre-installed KGEN at "/glade/u/tdd/asap/contrib/kgen/std"

KGEN requires either Fortran preprocessor ("fpp") or C preprocessor("cpp"). Currently fpp is default.

Makefiles are generated by KGEN to make it easy to generate state data and to build/run an extracted kernel.

2.2 User interface

KGEN will support two ways of user-interface: command-line and KGEN directive.

KGEN directives are inserted in source codes to directs what KGEN to do. Command-line inputs overwrites KGEN directives.

2.2.1 KGEN directive user interface

In general, the syntax of KGEN directive follows OpenMP syntax. Case is not sensitive if not specified.

SYNTAX: !$kgen directive name/clause(s)

directive specifies the type of setting. name or clause(s) is additional information for the directive. Name is a non-space ascii string and clause is a name with having parenthesis.

Continuation of kgen directive is supported as follow:

!$kgen directive name/clause(s) &
!$kgen& [additional name/clause(s)]

"callsite" is the only KGEN directive implemented in this version.

Callsite directive

syntax: callsite name

meaning: callsite directive specifies the location of callsite Fortran statement in a source file. If this directive is specified in source file, user does not have to provide “namepath” on command-line. “namepath” is a colon-separated names. Please look at command line user interface for details

The directive should be placed just before the callsite line. However, blank line(s) and other comment lines are allowed in-between. name is a subroutine or a function to be extracted as a kernel.

example:

```
!$kgen callsite calc
CALL calc(i, j, output)
```

2.2.2 KGEN command-line user interface

Once you have KGEN files in an accessible directory, you can run generate a kernel using KGEN by issuing following command without adding KGEN directive in a source file.

>> python $PATH_TO_KGEN/kgen.py [KGEN flags] <filepath>:<callsite identifier>

“filepath” is a path to a source file that contains a call to the kernel subprogram to be extracted.” “callsite identifier” tells KGEN which Fortran line is the callsite. The simplest way to set “callsite identifier” is to use the name of subroutine or function name. For example, if the callsite is “CALL calc(i, j, output)”, then “callsite identifier” is “calc”. If there are multiple of the name in the file, user can specify a hierarchy of name path, which is explained in section 2.3.1.

2.3 Supporting kernel generation for large-sized software

As we are interested in large-sized code, it is likely that the software adopts modular approaches with multiple source files. KGEN can correctly analyze source codes only when all required paths for searching modules and correct set of macro definitions are provided.

2.3.1 Namepath

To resolve possible name conflict among different levels of namespace, KGEN introduces a hierarchical representation of a name, called “namepath.”

Namepath is consecutive names with colons between them as a separator. For example, if name B is inside of A, the it can be represented by “A:B”. In practice, it is used to specify “ identifiers” in KGEN. For example, callsite identifier is a namepath that locates “callsite” to kernel.

an example of callsite identifier:

“module_A:function_B:subroutine_C”

In above example, user specified a Fortran line that contains “subroutine_C”, whose upper namespace is “function_B”, and again one more upper level is “module_A”.

If there is no name conflict, for example that there is only one “subroutine_C” exists in the given file, user can only specify “subroutine_C” without upper-level names.

The separator of colon can be used as a metacharacter meaning of “any” similar to “*” in “ls” linux command.

Leading colon means any names can be placed before a name placed next to the colon. For example, “:name_a” means any namepath that ends with “name_a”.

Similarity, colon at the end of a namepath means any names can be followed after a name placed before the colon. For example, “name_a:” matches to any namepath that starts with “name_a”.

Finally, double colons between names means any names can be placed between the two names. For example, “name_a::name_b” matched to any namepaths that starts with “name_a” and ends with “name_b”

2.3.2 User-provided inclusion information

KGEN accepts an INI-format file with “-i” command-line option. In the INI file, user can provide KGEN with additional information including macro-definitions, include directories and others. Details of using the INI file are explained in following sections.

Command line option format: “-i <user-providing INI format file>”

Syntax of the INI file follows conventional INI file syntax. Brackets are used to specify sections. In a section, an option is added in a line or over multiple lines. Each option has a format of key and value pair with a separator of “=”. Value part can be missed depending on the type of option.

2.3.2.1 Information for each source file

Some information has to be provided per each source file separately. As of this version, there are two types of information are identified in this category: macro definition and include directory. Following convention is used to provide these information in the INI file.

[Path-to-source-file]
include = [directory path]:[directory path]:...
macro_name = macro_value
...

example) When “program.F90” uses a module in “./module” directory, and “program.F90” needs macro definition of “N=10”

[program.F90]
include = ./module
N = 10

There can be multiple macro_name options but only one “include” option is allowed per each file.

By providing these options, user allows KGEN to analyze source code correctly. In addition, KGEN will analyze files specified in this INI file first, which may help to reduce analysis time.

2.3.2.2 Common information to all source files

There are several types of information that can be applied to all source files that KGEN analyzes.

Common macro definitions and include directories

In some cases, all source files may share the same macro definitions and/or include directories. In the cases, instead of specifying the information per each source file sections, user can use following sections.

[macro]
macro_name = macro_value
…

These macros will be added to each source file during KGEN analysis.

[include]
include_path1 =
include_path2 =
…

These include_paths will be added to each source file during KGEN analysis. Note that each path should be specified per each line, which is different from the syntax of separate section for each file. Value part of each option should be blank for this version and is reserved for later use.

2.3.2.3 importing additional information

User can provided additional information through “import” section in a INI file.

[import]
information_object = information_type

As of this version, “source” information_type is the only type supported. An “source” type option in import section provides KGEN with paths to additional files to be analyzed before starting main parsing tasks.

[import]
/path/to/source = source

While KGEN tries to collect information required to analyze the source codes, there are some cases that those tasks can not be done automatically. One of them is to search Fortran external subprogram. Fortran external subprogram can be called from any parts of Fortran code without the need of using Fortran use statement, which make it hard for KGEN to discover the importing connection.

2.3.3 User-provided exclusion information

KGEN accepts an INI-format file with “-e” command-line option. In the INI file, user can provide KGEN with information of names to be excluded during name search. Details of using the INI file are explained in following sections.

Command line option format: “-e <user-providing INI format file>”

Syntax of the INI file follows conventional INI file syntax. Brackets are used to specify sections. In a section, a option is added in a line or over multiple lines.

2.3.3.1 namepath section

When “namepath” is specified in a section of INI file, “actions” specified in an option are applied to namepath in the option.

[namepath]

namepath = action(s)

“namepath” in an option line specifies target of actions. The syntax of “namepath” is explaiend in section 2.3.1. “action(s)” will be applied the namepath in the same line.

Regardless of actions specified in an option line, any name in Fortran source codes that matches to namepaths will be skipped from name resolution in KGEN.

There is four “action” are defined in this version.

When “comment” is specified as “action”, KGEN will comment-out Fortran statements that contain the namepath in generated kernel files.

When “skip_module” is specified as “action”, KGEN will not use a module specified by the namepath during name resolution tasks.

When “remove” is specified as “action”, KGEN will not use save state data specified by the namepath.

When “use_object” is specified as “action”, KGEN will copy an object files into “kernel” folder under output directory. This is useful if the production code use identifiers in object files that are written in different languages. The syntax of the action is “use_object(/path/to/object/file)”

2.4 Kernel extraction examples

2.4.1 simple example

In this section, an example of generating a kernel of “calc” from “calc_mod.F90” source file is explained. This example shows a basic workflow of using KGEN with information on using essential KGEN flags. You can find this example in “simple” folder of KGEN distribution.

program.F90

```
PROGRAM demo
  USE update_mod, &
      only : update
  INTEGER t
  DO t=1,10
    CALL update
  END DO
END PROGRAM
```

update_mod.F90

```
MODULE update_mod
  USE calc_mod, only : calc
  PUBLIC update
CONTAINS
  SUBROUTINE update()
    INTEGER :: i, j
    INTEGER :: output(COL,ROW)
    DO i=1,COL
      DO j=1,ROW
        !$kgen callsite calc
        CALL calc(i, j, output)
      END DO
    END DO
  END SUBROUTINE
END MODULE
```

calc_mod.F90

```
MODULE calc_mod
  PUBLIC calc
CONTAINS
  SUBROUTINE calc(i, j, output)
    INTEGER, INTENT(IN) :: i, j
    INTEGER, INTENT(OUT), &
      dimension(:,:) :: output
    output(i,j) = i + j
  END SUBROUTINE
END MODULE
```

In this example callsite is “CALL calc(i, j, output)” in update_mod.F90. Callsite should be placed in a subprogram of a module.

To extract “calc” subroutine and make it stand-alone application, following KGEN command is used.

```
>> python ${KGEN} \
        -D ROW=4,COL=4 \
        -I ${SRC_DIR} \
        --timing repeat=1000 \
         --invocation 1:2:3 \
        --kernel-compile FC=ifort,FC_FLAGS='-O3' \
        --state-build cmds="cd ${SRC_DIR}; make build" \
        --state-run cmds="cd ${SRC_DIR}; make run" \
        ${SRC_DIR}/update_mod.F90
```

Each lines of the command are explained below.

“>> **python ${KGEN}**” : This invokes KGEN. It is assumed that ${KGEN} has valid path to “kgen.py” Python source file in KGEN distribution. KGEN is developed using Python 2.6.6.
“-D ROW=4,COL=4 \” : “-D” KGEN flag specifies the macro definitions that are used for preprocessing Fortran source files. In this example, “ROW” is defined as “4” and “COL” is defined as “4”. This information will be used by “fpp” or “cpp” preprocessors before KGEN starts to parse input source files.

“-I ${SRC_DIR}” : “-I” KGEN flag specifies the directories for KGEN to search Fortran source files. In this example, it is assumed that ${SRC_DIR} has a valid path to a directory that contains all of three input Fortran source files, “program.F90”, “update_mod.F90” and “calc_mod.F90”

“--timing repeat=1000”: “--timing” flag specifies information for timing measurement. “repeat” sub-flag specifies how many calls to be made to kernel to measure timing. Larger number increases the resolution of measurement but increase time to measure.

“--invocation 1:2:3”: “--invocation” flag specifies when to save state data in terms of calls to the kernel. In this example, state data will be saved first, second, and third calls to the kernel.

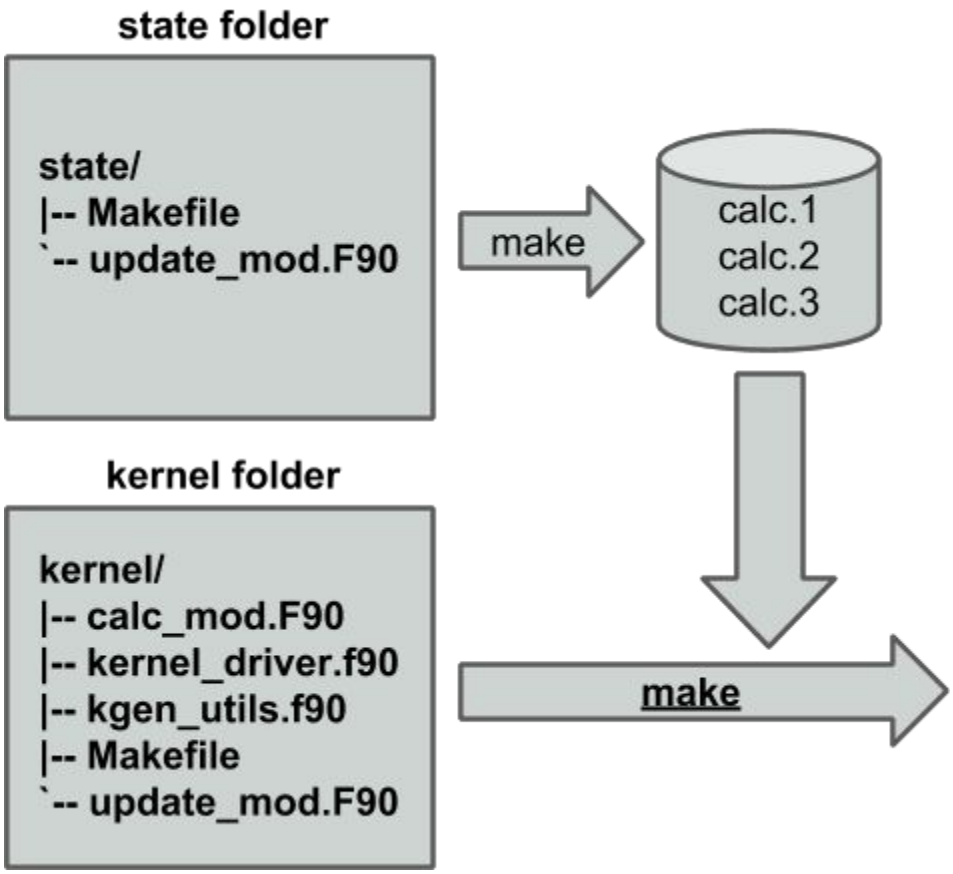
“--kernel-compile FC=ifort,FC_FLAGS='-O3' \”: “--kernel-compile” KGEN flag provides the Fortran compiler and compiler flags to compile generated kernel. It is prefered to use the same compiler and compiler flags to original production compilation.

“--state-build cmds="cd ${SRC_DIR}; make build"” : “--state-build” KGEN flag provides command how to build instrumented original production application. KGEN will generate Makefile that contains the content of “cmds” sub-flag as a part of building task. In this example, it is assumed that Makefile is provided in original application.

“--state-run cmds="cd ${SRC_DIR}; make run" \” : “--state-run” KGEN flag provides command how to run the instrumented original production application. KGEN will generate Makefile that contains the content of “cmds” sub-flag as a part of execution task. In this example, it is assumed that Makefile is provided in original application.

“**${SRC_DIR}/update_mod.F90**” : This line specifies the path to a source file that contains “callsite”. In this example, the exact location of callsite is specified by KGEN “callsite” directive in source file.

Once KGEN completes kernel generation successfully, two subfolders will be created in current working directory as shown below



Next step is to generate data from executing original production code with replacing some of files with KGEN-generated files in “state” folder. This process is automated by Makefile in the “state” folder. Run “make” in “state” folder. Once it completes with success, three data files, “calc.1”, “calc.2”, and “calc.3” in this example, will be created in “kernel” folder.

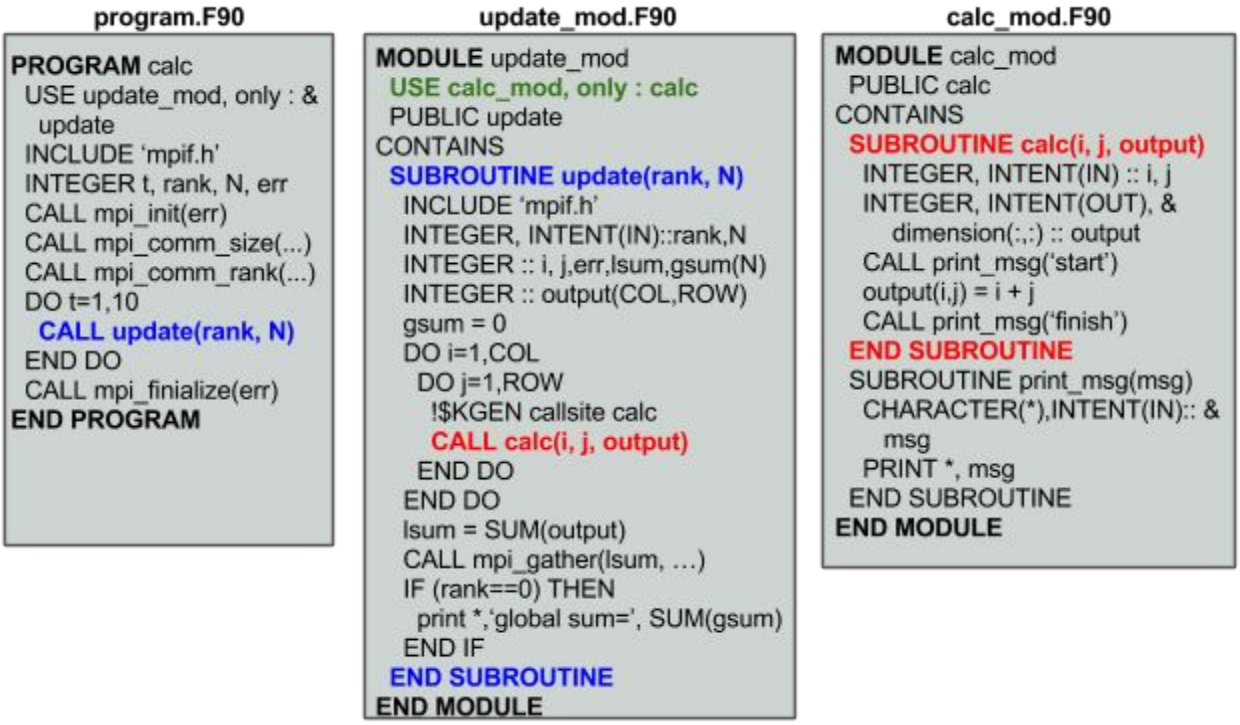
Final step is to build and run the “calc” kernel. Move to “kernel” folder and run “make”. If it completes with success, you will see something similar to following figure on screen.

```
>> make
ifort -O3 -c -o kgen_utils.o kgen_utils.f90
ifort -O3 -c -o calc_mod.o calc_mod.F90
ifort -O3 -c -o update_mod.o update_mod.F90
ifort -O3 -c -o kernel_driver.o kernel_driver.f90
ifort -O3 -o kernel.exe kernel_driver.o
  update_mod.o kgen_utils.o calc_mod.o

./kernel.exe

 ** Verification against './calc.1' **

 calc: Tolerance for normalized RMS:   9.99E-015
 calc: Number of variables checked:          1
 calc: Number of Identical results:          1
 calc: Number of warnings detected:          0
 calc: Number of fatal errors detected:          0
 calc: Verification PASSED

 calc : Time per call (usec):   6.00E-003

 ** Verification against './calc.2' **
...
 ** Verification against './calc.3' **
...
 ***************************************************************
 calc summary: Total number of verification cases: 3
 calc summary: Total time of all calls (usec):   1.69E-002
 ***************************************************************
```

This shows that the kernel is compiled with success and the correctness check shows that kernel generated data is identical to data generated from original production application. This verification repeated per each of three data files. It also shows that the kernel execution took 6.00E-003 usec for the first verification and 1.69E-002 usec for all three verifications.

2.4.2 simple-MPI example

In this section, an example of generating a kernel of “calc” from “calc_mod.F90” source file in simple-MPI folder is explained. This example shows a workflow of using KGEN for MPI applications. Especially, this example includes the usage example for using “-e” flags which is an essential part of supporting large-sized application.



In this example callsite is “CALL calc(i, j, output)” in update_mod.F90. Callsite should be placed in a subprogram of a module.

Please change FC and FC_FLAGS in Makefile and src/Makefile for your compiler and MPI environment.

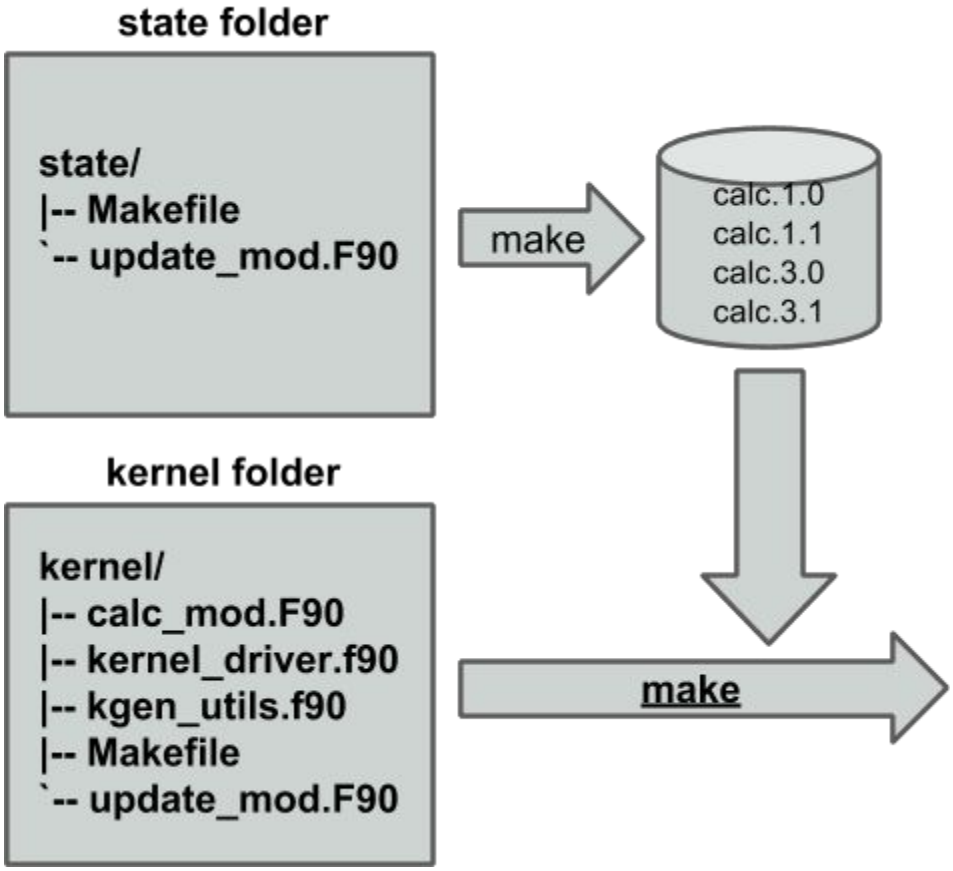
To extract “calc” subroutine and make it stand-alone application, following KGEN command is used.

```
>> python ${KGEN} \
        -D ROW=4,COL=4 \
        -I ${SRC_DIR}:${MPI_INC} \
        --timing repeat=1000 \
        --mpi ranks=0:1 \
         --invocation 1:3 \
        --kernel-compile FC=ifort,FC_FLAGS='-O3' \
        --state-build cmds="cd ${SRC_DIR}; make build" \
        --state-run cmds="cd ${SRC_DIR}; make run" \
        ${SRC_DIR}/update_mod.F90
```

Only “--mpi” flag is explained in this section. Please see a section for “simple” example for the other flags.

“--mpi ranks=0:1”: “--mpi” flag specifies MPI-related information. “ranks” sub-flags specifies the mpi ranks that will save state data. In this example, state data will be saved from MPI ranks 0 and 3.

Once KGEN completes kernel generation successfully, two subfolders will be created in current working directory as shown below



Next step is to generate data from executing original production code with replacing some of files with KGEN-generated files in “state” folder. This process is automated by Makefile in the “state” folder. Run “make” in “state” folder. Once it completes with success, three data files, “calc.1.0”, “calc.1.1”, “calc.3.0”, and “calc.3.1” in this example, will be created in “kernel” folder. The encoding of the two number of the state file is “invocation number” and period and “MPI rank number”.

Final step is to build and run the “calc” kernel. Move to “kernel” folder and run “make”. If it completes with success, you will see something similar to following figure on screen.

```
>> make
ifort -O3 -c -o kgen_utils.o kgen_utils.f90
ifort -O3 -c -o calc_mod.o calc_mod.F90
ifort -O3 -c -o update_mod.o update_mod.F90
ifort -O3 -c -o kernel_driver.o kernel_driver.f90
ifort -O3 -o kernel.exe kernel_driver.o
  update_mod.o kgen_utils.o calc_mod.o

./kernel.exe

 ** Verification against './calc.1.0' **

 calc: Tolerance for normalized RMS:   9.99E-015
 calc: Number of variables checked:           1
 calc: Number of Identical results:           1
 calc: Number of warnings detected:           0
 calc: Number of fatal errors detected:           0
 calc: Verification PASSED

 calc : Time per call (usec):   6.00E-003

 ** Verification against './calc.1.1' **
...
 ** Verification against './calc.3.0' **
...
 ** Verification against './calc.3.1' **
...

 ***********************************************************************
 calc summary: Total number of verification cases: 4
 calc summary: Total time of all calls (usec):   2.19E-002
 ***********************************************************************
```

This shows that the kernel is compiled with success and the correctness check shows that kernel generated data is identical to data generated from original production application. This verification repeated per each of three data files. It also shows that the kernel execution took 6.00E-003 usec for the first verification and 2.19E-002 usec for all four verifications.

2.5 Syntax checker

Current version of KGEN supports a subset of Fortran 95 specification and a very limited support for Fortran 20003 specification. Therefore, it is worthy to check whether current KGEN can parse an application correctly before one invests major time and effort for kernel extraction.

To run the syntax checking feature of KGEN, user can run kgen as following:

python $PATH_TO_KGEN/kgen.py ${OPTS} -s [source folders | source files]

OPTS: KGEN options that are compatible to this feature: -i -e -I -D intrinsic
source folders: folders that contains Fortran source files to be checked. KGEN will check subfolders recursively
source files: Fortran source files to be checked
Once the syntax checking is completed, it will show if there is any statements in the specified directories or files that KGEN can not correctly parse and analyze for kernel generation. However, KGEN may still be able to generate kernel correctly even though the syntax checking shows any non-supported statements. It is because the non-supported statements may not be part of generated kernel so that KGEN does not need to parse them at all.

2.6 Command line options

The syntax of each options generally follows the following convention:

General KGEN option syntax:
-[-]<option-name> [<suboption-name>=<suboption-value>[,[<suboption-name>=]<suboption-value>]]

If there are multiple information in <suboption-value>, each information would be separated by colon, ":". Double or single quotation marks can be used to use some of the separation symbols, such as equal sign, comma, colon, in option value.

[-i, --include]
meaning: provides information of what to be included during KGEN analysis. Information includes macro definitions and include directories.
example) -i include.ini

[-I]
meaning: provides include path information where KGEN search files. This information applies to all source files
example) -I /path/to/directory/a:/path/to/directory

[-D]
meaning: provides macro definitions. This information applies to all source files
example) -D USE_A=1

[--invocation]
meaning : specifies Nth kernel invocation to save input/output data
example) --invocation 1:10:100

[--mpi]
meaning : Turns on MPI supports in KGEN. There are several sub-options: “ranks”, “comm”, “use”, and “header”. “ranks” specifies the MPI ranks on which data is collected. Default “ranks” is 0. “comm” specified the names of variable that is used when MPI call is made. Default “comm” is “MPI_COMM_WORLD”. “use” specifies Fortran module name whose name is inserted in additional Fortran use statement. There is no default value for “use”. “header” specifies the path to MPI header file. Default “header” is “mpif.h”.
example) --mpi ranks=0,comm=mpicom,use="spmd_utils:mpicom"

[--intrinsic]
meaning : flags to let KGEN skip searching for names of intrinsic-procedures. At minimum, one of “skip” or “noskip” should be provided. With “except” sub-flag, user can specify which “namepath” should be considered as exception. With “add-intrinsic” sub-flag, user can add new intrinsic function names. default: --intrinsic skip
example) --intrinsic skip,except=mod_A.subr_B.sum

[--source]
meaning : this flags specifies information related to source file. “format” sub-flag specifies the Fortran format of source files “fixed” is used for F77 format and “free” used for F90 and later format. With this sub-flag, KGEN forces to use the specified format regardless of file extension. “strict” format let parser of KGEN informe to apply format strictly or not. Default is of the sub-flag is “no”, “alias” sub-flag create path alias. This is useful if you have one file physical location but has two different paths that points the same physical path.
example) format=free,strict=no,alias=/path/A:/path/B

[--kernel-compile]
meaning : compile-specific information used in generating Makefile for kernel. Two sub-flags are defined in this version: “FC” and “FC_FLAGS”. User can choose which Fortran compiler to be used in the kernel makefile with “FC” flag..
example) --kernel-compiler FC=ifort,FC_FLAGS=-O3

[--kernel-link]
meaning : compile-specific information used in linking-phase. Three sub-flags are defined in this version: “pre_cmds”, “include”, and “lib”. Values of “pre_cmds” sub- flags will be executed before the generated Makefile builds the kernel. “include” sub-flags provides include paths for libraries. The paths of this flag will be added after “-L” compiler flag. “lib” sub-flags provides the name of libraries. The names of this flag will be added after “-l” compiler flag.
example) --kernel-link include=”./”,lib=mylib

[--state-build]
meaning : build-specific information used in generating Makefile for original program. One sub-flags are defined in this version: cmds. User can run Linux command specified in cmds.

example) --state-build cmds="cd ${WORK_DIR}; module load mkl; make"

[--state-run]
meaning : execution-specific information used in generating Makefile for original program. One sub-flags are defined in this version: cmds. User can run Linux command specified in cmds.
example) --state-run cmds="cd ${WORK_DIR}; make run"

[--state-switch]
meaning : provides information how to use KGEN-generated instrumented source files during compiling original production code. Two sub-flags are defined in this version: "type" and "cmds". Two string values, "replace" and "copy" are valid for "type" sub-flag. "replace" let KGEN replace the files in original production files with instrumented files. "copy" let KGEN copies instrumented files into a folder. When "copy" is used, the other sub-flag, "cmds" should be also provided that actually copy the instrumented files a folder.
example) --state-switch type=copy,cmds="cd ${WORK_DIR}; rm -f ./*; cp -f ${GEN_DIR}/*.F90 ./"

[--outdir]
meaning : KGEN output directory
example) --outdir /path/to/output/directory

[--timing]
meaning : provides information about performance measurement. One sub-flag is defined in this version: repeat. "repeat" provides the number of invocations to the kernel subprogram to enhance the measurement resolution.
example) --timing repeat=1000

[--debug]
meaning: This flag specifies information related to debugging KGEN. One useful option is "printvar", which let KGEN add codes for displaying the content of specified variables during kernel execution as well as original production code execution with instrumented codes.
example) --debug printvar=var_a,var_b