Community Unified Post Processor (UPP) Code Management Plan

Month Day, 2018  
Point of Contact: Kate Fossell

# Introduction

The Unified Post Processor (UPP) software package was developed by the Environmental Modeling Center (EMC) of the National Centers for Environmental Prediction (NCEP). This package provides the Numerical Weather Prediction (NWP) community with a common tool to post-process output from multiple models, ranging from global to regional scales. Development of the UPP software package was based on NCEP’s Weather Research and Forecasting (WRF) Post Processor (WPP). UPP retains capabilities that were available in WPP while expanding beyond to offer additional features. UPP can be used to post-process both WRF Nonhydrostatic Mesoscale Model (NMM) and Advanced Research WRF (ARW) dynamic core output format. Currently, UPP is used in operations with the Global Forecast System (GFS), the North American Mesoscale (NAM) Model,, the Weather Research and Forecasting (WRF) Rapid Refresh model and the Hurricane Weather Research and Forecasting (HWRF).  
  
The Developmental Testbed Center (DTC) will serve to establish a link between the operational and research communities by maintaining a community UPP repository and providing regular public releases and user support for this software package. In order to benefit the entire NWP community, the community UPP repository must be maintained in a way that allows future updates and enhancements to be contributed by, and shared between, both the operational and research communities.

Supporting and managing a version of the UPP package applicable to community and operational users requires a plan for coordinating the sharing of code between developers with diverse needs. The DTC will be responsible for maintaining a community UPP package which is kept in sync with the operational UPP package (O2R). Community users may contribute modifications and enhancements to the UPP package; these modifications will be tested and made available to the operational community (R2O). This document outlines the policies and procedures used to maintain a robust and flexible UPP software package. The document contains descriptions pertaining to:

* Community UPP code repository
* Code Contributions
* Synchronizing the DTC Community UPP ⬄ NCEP UPP code repositories
* Release Schedule and User Support

# Community UPP Code Repository

## Physical Storage

The hardware, version control software, and repositories used to store and access the community UPP repository are maintained and administered by the National Center for Atmospheric Research (NCAR)/NCAR Earth Systems Laboratory (NESL)/Climate and Global Dynamics Division(CGD). NCAR/NESL/CGD repository data resides on a RAID unit configured to use Level 6 redundancy. The RAID unit can suffer two drive failures without loss of data. Two servers are configured for high availability. One system acts as the primary with the other in reserve. Roles for the servers are swapped for maintenance upgrades or in the event of hardware failure. Two sets of backups are kept. The first is a nightly rsync of the repositories to another RAID unit. This backup is over written daily. The second backup is the traditional disk-to-tape method. Incremental backups are performed nightly, with full backups performed once per month. These backups are kept for 90 days. Backups are considered as a recovery means and are not archival in nature.  
  
The version control system used is SVN with the database being configured under Rev 1.5. The community UPP package was initially derived from the NCEP operational UPP, NCEP SVN repository r9594 (08/31/2010).

## Software Directory Structure

The community UPP repository is comprised of two physical repositories linked using SVN configuration attributes. The main UPP source code and platform architecture support are in the repository: https://svn-dtc-unifiedpostproc.cgd.ucar.edu. The external supporting library code is in a separate repository: https://svn-dtc-external-lib.cgd.ucar.edu (Figure 1. UPP Repository Storage Structure).

NCAR/CGD  
Repository

External  
Libraries

UPP

Source & Support

Figure 1. UPP Repository Storage Structure

### External Libraries Repository

The external libraries are derivatives of the NOAA/EMC‘s repository <https://svnemc.ncep.noaa.gov/projects/nceplibs>. These libraries will strive to be identical except where the need for difference is determined to be necessary. A DTC preprocessor flag is used to facilitate the sharing of files with minor code variations. At the current time, the NCEP libraries are under going modifications to incorporate changes from several groups within EMC and to support the little endian format. At the DTC, an effort is under way to sync the community version of these libraries with EMC’s nceplibs once the unification process internal to EMC has been complete. Until this work is accomplished, these libraries are not advertised to produce an identical result between the community and operational versions.

The external libraries will be upgraded to remain current with EMC on a TBD basis. There are unit tests available from the EMC repository with each of these libraries. At this time the unit tests are not available in the DTC repository, but will be added in the future to ensure consistency.

The structure of the DTC external libraries repository follows:

Figure 2. Community External Library Repository Structure

build/

projects/

src/

arch/  
Makefile  
clean  
configure  
compile  
.externals

HWRF  
GSI  
UPP

bacio crtm  
g2 g2tmpl  
gfsio ip  
mersenne nemsio  
sfcio sigio  
sp w3  
wrfmpi\_stubs xml  
makefile

A user may create a branch as a workspace based on the *comm\_branch* directories. User

The contents of the build/ directory can be used to build a selection of libraries independently of any project. The projects/ directory contains a directory for each of the supported projects. Under each project are the build criteria for that project. The src/ directory contains a directory for each library. Within each of these libraries are the 3 standard SVN directories: trunk/, branches/, and tags/. The trunk/ directory of each library directory is the synchronization point for the NCEP source for that library. Unipost will use the branch name: <libname>\_upp for libraries that are not identical to the trunk of the specific library. Developers working within a library (ies) should use a branch of the effected library (ies). All branches should reside under the branches/ directory for each specific library. The tags/ directory is used to tag each release, check-ins that have the DC committee approval, and any notable milestones of development. No development work is done in the tags/ directory.

### Community UPP Repository

The UPP community repositories contain an image of the EMC post/trunk, svn externals links to the libraries src/, and DTC specific build support. The NOAA/EMC operational UPP repository mirrored is located at https://svnemc.ncep.noaa.gov/projects/post/trunk. The code within each repository will strive to be identical except where the need for difference is determined to be necessary or unavoidable. A DTC preprocessor flag is used to facilitate the sharing of files with minor code variations.

The structure of the DTC UPP community repository follows:

User1  
User2  
…  
UserN

branches/

trunk/

tags/

Figure 3 Source Code Structure of Community Code Repository

UPPv0.5/  
UPPv1.0/  
UPPv1.1/

clean  
configure  
compile  
arch/  
makefile  
src/  
parm/  
scripts/

lib  
unipost  
ndate  
copygb

Figure 4. Community UPP Repository Directory Content

The UPP trunk/ directory is the central location of the code needed to create executables of unipost and associated utility programs (i.e., ndate, copygb). The scripts: clean, configure, and compile are the command directives used to create the executables. These scripts use information in the arch/ directory to build a version of UPP which is compatible with the system/architecture it has been installed on. The trunk/src directory contains all the source code. The source code includes the lib/ directory which is made up of links to the external libraries repository. The main unipost code is in src/unipost/ and is a mirror image of the trunk in the NOAA EMC operational post repository. The utility programs, ndate and copygb ,are also located within the unipost repository under the src/ directory. These utilities were also developed by NOAA/EMC.

The contents of these directories are further expanded and defined in the following table.

|  |  |
| --- | --- |
| Directory Name | Description |
| trunk/clean | Script to remove compilation files |
| trunk/configure | Script to configure for compile |
| trunk/compile | Script to compile |
| trunk/arch/ | Architecture dependent support code |
| makefile | Initial makefile called by compile |
| trunk/parm/ | Output control parameter file |
| trunk/scripts/ | Scripts to run unipost |
| trunk/src/lib/bacio/ | binary I/O routines |
| trunk/src/lib/crtm2/ | Community Radiative Transfer Model |
| trunk/src/lib/ip/ | Interpolation Library |
| trunk/src/lib/mersenne/ | Random Number Generator |
| trunk/src/lib/sfcio/ | NCEP GFS Surface files I/O API |
| trunk/src/lib/sigio/ | NCEP GFS Sigma files I/O API |
| trunk/src/lib/sp/ | Spectral Grid Transform Library |
| trunk/src/lib/gfsio | GFS I/O routines |
| trunk/src/lib/nemsio | NEMS I/O routines |
| Trunk/src/lib/w3emc | GRIB1 code/decode Library |
| trunk/src/lib/w3nco | GRIB1 code /decode Library |
| trunk/src/lib/g2 | GRIB2 support library |
| trunk/src/lib/g2tmpl | GRIB2 table support library |
| trunk/src/lib/xml | XML support – GRIB2 parameter file |
| trunk/src/lib/wrfmpi\_stubs | Serial compilation support |
| trunk/src/unipost | Unified Post source code |
| trunk/src/copygb | Horizontal Interpolation Utility |
| trunk/src/ndate | Date formatting Utility |

Table 1: Unipost Repository Directories

A user may create a branch as a workspace based on the trunk. User branches should exist under the *branches/* directory and named such that they do not conflict with other branches. An important note is that when a user makes a copy of the trunk as a branch, the external links to the library repositories are NOT branched, unless they are explicitly changed. A script utility will be developed such that a user can make a branch which will include all repositories from which they can work without fear of committing code back to the trunk. An associated utiliy will also be developed to remove the branch.

The *tags/* directory is used to store snapshots of the community UPP package when a version is officially released to the user community. These releases include the formal annual release and any bug fixes that are posted as a code revision. In the event of numerous user contributions at any given time the *tags* directory may be used to help insure the order of the code changes to the repository. A user may tag a branch however no development is to occur in the tags/directory. A script to ensure all linked repositories are tagged is under development.

# Code Contribution Outline

## **Access to Repository**

A developers’ committee (DC) exists which will govern the contributions made to the community UPP repository. The DC will approve read access to the repository. Read/write access will be granted to a subset of the DC based on need. Initial membership of the DC will include representatives from NOAA/EMC, DTC, NOAA/Global Systems Division (GSD), and the Air Force Weather Agency (AFWA). A community UPP repository email list (commUPP\_developers@rap.ucar.edu), which includes all members of the DC, will be used to distribute correspondence related to the status of the community UPP repository. Individuals wishing to gain access to the community UPP repository should contact the DTC. The DTC will bring the request to the DC for approval. The DC will meet on an as needed basis.

## Working in the Repositories

DC members with read/write access are encouraged to make a branch(es) under the branches/ directory to work on changes they intend to make available to the community repository(ies) at a future date. This will facilitate the user in keeping current with the modifications that occur to the community code while they are developing their changes. The changes made under a user branch(es) are not used by the general community. This is an area where a group or individual may create a quasi-permanent UPP version which contains changes their project requires, but are not ready to be incorporated into the community code.

## Request to Accept Change into Community UPP Repositories

To check code modifications into the community repository, a DC member must request a code review. A code review meeting will be scheduled and should be attended by a majority of the DC members. The DC members in attendance have the authority to accept or reject the proposed changes. The procedure to follow in order to prepare code for review is outlined below.

1. Obtain a working copy of the code by checking out the HEAD of the community UPP repository (https:/svn-dtc-unifiedpostproc.cgd.ucar.edu/trunk). Users may wish to create a branch in the branches/ directory to aid merging back to the community branch.
2. Add modifications, enhancements or additions into branch. All code changes proposed for inclusion to the community branch of the repository must meet the coding standards described in Appendix A.
3. Run available regression tests (Appendix B) on modified code, as well as any specific computing and scientific tests the developer may have to ensure the modifications are behaving as intended and have not adversely impacted existing UPP capabilities.
4. Upon successful completion of testing (3), a proposal of the modifications should be submitted to the DC. The developer’s proposal must include a detailed description of the modifications to the system, a summary of the completed testing with results, instructions on usage of new feature(s), files modified, changes to program input/output, and code differences (unified diff format). The DTC will schedule a review meeting and notify the DC membership. The complete developer’s proposal must be received by the DC no later than 24 hours prior to the scheduled review meeting.
5. DC members should review all proposals for system commits prior to the scheduled review meeting. During the DC meeting, all code modifications are described and defended to the DC review members. The DC members may accept or reject any set of proposed changes. (Note: The successful completion of the regression tests is a necessary, but not sufficient, condition for acceptance of proposed changes.)
6. Approved system modifications may be committed to the trunk of the community UPP repository. In the event that multiple proposals are submitted prior to the same review meeting, the system changes will be prioritized by the attendees of the review meeting and the order of system upgrades will be determined. Upon successfully synchronizing to the HEAD of the trunk, testing the new code and committing it to the trunk, the first developer posts a notification that they are done (SVN auto-generated email will satisfy this obligation). The next developer must then merge the new repository HEAD with their changes, test, commit, and post notification of completion. This process continues until all developers have placed their system modifications in the Community UPP repository.
7. After all system modifications are in the trunk, the DTC will retrieve the trunk*,* compile UPP and supporting utilities, and run regression tests to ensure functional status. If the tests do not complete successfully the DTC will notify the party(ies) involved and work to correct the system error(s). This may include removal of a previously accepted modification. When the DTC determines the community UPP repository is in a working state a notification will be made to the DC membership to that effect, including any change removals that may have occurred. Developers who have their modifications removed may correct the noted error and resubmit their proposal to the DC.
8. The developers whose system modifications have been accepted and verified should delete the branch they made in the branches/ directory of the repository. If development is anticipated to continue on this branch it may remain with no required intervention.
9. The accepted changes not initiated by NCEP will be made available to NCEP to determine if the community modifications can be incorporated into the operational UPP.

# Synchronizing the DTC Community UPP ⬄ NCEP UPP code repositories

To maintain a strong connection between the research and operational communities, the DTC community UPP repository must be kept synchronized with the NCEP operational UPP repository. This will require the DTC to merge all NCEP operational changes into the community UPP repository, as well as present the community changes accepted by the DC into the community repository to NCEP for possible inclusion in the operational repository. A primary and secondary point of contact will be designated from the DTC staff who will work with the NCEP primary code manager to accomplish this goal. The remainder of this section outlines the communication required for repository synchronization.

## DTC ⇒ NCEP

The DTC will submit non-NCEP initiated changes accepted by the DC into the community UPP repository to NCEP for acceptance into the operational UPP repository. The documentation submitted to initiate the DC approval process will act as the documentation provided to NCEP. The DTC will open an EMC TRAC ticket for each request submitted to NCEP. For larger changes a branch can be made to the EMC repository and code transferred through the branch.

## NCEP ⇒ DTC

NCEP will send notification of updates via email messages. These email messages may be generated by the operational SVN configuration system or the NCEP principal code manager. There are three forms of updates the DTC will receive from NCEP regarding the Community UPP repository. These include:

1. New operational UPP software capabilities (via SVN)
2. Acceptance of community contributions to the operational repository (via TRAC)
3. Refusal of community contributions to the operational repository (via TRAC)

### Operational UPP Software Modification

The DTC point of contact receives email notification that operational UPP software has been modified. The SVN or TRAC logs shall describe the files which changed and provide a description of the change. The DTC will proceed as follows:

* The DTC will make a branch based off the community trunk(<https://svn-dtc-unifiedpostproc.cgd.ucar.edu/trunk>).
* The NCEP changes will be incorporated in this branch.
* The DTC will run regression tests.
* Upon successful completion of the regression tests, the modification will be brought before the DC for acceptance into the trunk.
* The details in Section 3.3 (Request to Accept Change into Community UPP Repositories) further explain the remaining procedure. The DTC will act as the originator during this procedure.

### Acceptance of a Community Contribution

The DTC point of contact receives email notification that a community contribution has been accepted in to the operational UPP repository. The DTC determines if the accepted operational implementation matches the initial submission or if modifications have been made. Variations to white space (blanks, tabs, and newlines) are not considered a modification for this purpose. If the change is identical there is no work to be done in the repository. The DC will be notified that the change was accepted by NCEP. The DTC will then update the tracking database as discussed in Section 4.1, DTC ⇒ NCEP, to reflect the acceptance of the contribution. White space changes will be reconciled by the DTC if necessary to keep the code as close as possible with the NCEP operational trunk. The DTC may use the procedure outlined in Section 3.3 to modify the community repository with these changes.

In the event the modifications made by NCEP include variations to the initial DTC submittal the DTC will notify the DC of the changes made by NCEP. If any questions arise from the DC, the DTC will communicate those back to NCEP and iterate, as necessary, until a final version is agreed upon. Any code changes from the initial DTC submission to NCEP will terminate the acceptance of the current approval and reinitiate the process with the most recent modifications. Upon acceptance of the modifications, the DTC will follow the outline found in Section 4.2.1. All communication shall be documented in the EMC TRAC ticket initially opened.

### Decline of a Community Contribution

NCEP has full authority over what is accepted into the operational UPP repository. They may, for any reason, decide to not accept a community contribution. An explanation of refusal shall be included in the notification email sent to the DTC. Any explanation of why the contribution was declined, including possible changes which would make the modification acceptable, will be supplied to the DC. The EMC TRAC tracking system will be updated to identify the contribution as existing in the community UPP repository only (submitted and declined).

The compile flag COMMCODE may be used to allow code to exist in the DTC repository while EMC is not ready or willing to accept the change.

# Community UPP Release and Support

The DTC will be responsible for regular code releases to the community. The DTC, along with other developers, will participate in pre-release testing and evaluation which will go beyond the limited regression tests used for on-going maintenance. The release procedure will consist of:

* The trunk/of the community UPP repository will be placed in a “frozen” state.
* During the “freeze” period the DC will only review bug fix contributions.
* Regular release meetings will be held during the freeze period. At the first release meeting, the additions/modifications available in the up-coming UPP software package will be determined and documented. This document will be made available before or in conjunction with the release.

Community UPP users can utilize the helpdesk email account [wrfhelp@ucar.edu](mailto:wrfhelp@ucar.edu) to submit questions regarding any aspect of UPP. The DTC will serve as the frontline for this task. Although the DTC will be responsible for responding to user inquiries, developers will need to assist with inquiries that go beyond the expertise of the DTC staff. Thus, once any new feature is released, the owners of those additions will automatically become part of the support team (helpdesk) who may need to answer questions from users. Code contributors will also be responsible for supplying pertinent documentation upon submission of any new code. The DTC will be responsible for reviewing and updating official UPP documentation with each new release.

1. Code Practices

Preamble

The reason for putting this document together, apart from establishing some minimum standard for code quality from developers outside EMC or JCSDA, is to provide a basis for consistency amongst the many UPP developers.

One thing to remember other people will be reading and trying to understand your code – be nice to them.

Style

* Use free format syntax.
* Indentation: begin in first column for statements such as program, module and contains, and recursively indent all subsequent blocks by *at least* two spaces.
* Do not use tab characters – they are not part of the Fortran character set.
* Name ends fully, including the program unit name.
* When creating new code (this includes refactoring[[1]](#footnote-1) old code), use the style guidelines above within the context of your personal style. If you use a syntax sensitive editor, as an experiment, turn off the syntax coloring to see if your code is still easily readable.
* When modifying old code, adhere to the style of the existing code.

Comments/Documentation

* For cryptic variable names, state description in a comment immediately preceding declaration or on end of the declaration line.
* For procedures and modules, insert a contiguous documentation block immediately following its declaration containing a *brief* overview followed by an optional detailed description.
* Ensure procedure argument documentation in the doc block is consistent with additions and/or deletions from the calling list.
* Procedure argument documentation in the doc block should briefly describe what are the arguments and their units. In some cases, this level of documentation may be unnecessary (e.g. the arguments to a generic interpolation procedure.) If in doubt, err on the side of documenting the argument list.
* Document any modifications made by using a short, but descriptive, log message when checking the modified code into the software repository. Don’t just say *what* has changed – since differencing versions provides that information – but *why*.
* Do not document changes within the code with comments that include the user’s name or initials.

Variable Declarations

* Declare all variables (IMPLICIT NONE)
* Use meaningful, understandable names for variables and parameters.
* Do not use Fortran intrinsic function names for variable names.
* Declare intent on all dummy arguments.
* Declare dimension attribute for all non-scalars.
* Line up attributes within variable declaration blocks.
* Any scalars used to define extent must be declared prior to use.
* Declare a variable name only once in a scope, including use module statements.

Modules

* Use modules to group related procedures and/or shared data.
* Use the only attribute on use statements as required.
* Declare implicit none.
* Include a private statement and explicitly declare public attributes.

Subroutines and Functions

* Group all dummy argument declarations first, followed by local variable declarations.
* Declare intent on all dummy arguments.
* To avoid null or undefined pointers, pointers passed through an argument list must be allocated.

Control Constructs

* Name control constructs (e.g., DO, IF, SELECT CASE) which span a significant number of lines or form nested code blocks.
* No numbered do-loops.
* Name loops that contain cycle or exit statements.
* Do not use goto.
* Use Fortran95-style relational symbols, e.g., >= rather than .ge., /= rather than .NE..
* For multiple selection tests, use case statements with case defaults rather than if-constructs wherever possible.

Miscellaneous

* Always initialize pointer variables in their declaration statement using the NULL() intrinsic, e.g.

**INTEGER, POINTER :: x => NULL()**

* Use modules (not common blocks) for sharing large segments of data.
* Remove unused variables.
* Do not use, *if at all possible*, compiler specific functions or calls. Doing so limits portability of the code. If compiler specific functions or calls must be used, localize the impact by wrapping the compiler extensions within a generic procedure and call that generic procedure. Document the potential portability problem in the calling code.
* Always use generic, not specific, intrinsic functions, e.g. COS rather than DCOS.
* Remove code that was used for debugging purposes once the debugging is complete.
* A standard naming convention has been adopted for variables and routines which refer to tangent linear and adjoint values, use suffixes “\_tl” and “\_ad” respectively.

Appendix A was taken from the Boulder Community GSI Code Management Plan (2009)

1. Regression Testing

A suite of regression tests will be made available for users to run while doing development and prior to requests for committing system modifications back into the community UPP repository. These regression tests will be broken down into code functionality, allowing users to better match their modifications to an appropriate test. To aid developers in understanding the available tests and their application a brief explanation of the software architecture has been included here.

The UPP begins by reading in a file named ITAG. This file specifies the input filename, model, and input data format. There is other information in this file that will not be discussed here. Based on the model and input format, the input data is read. There is minimal calculation done at this point. The code returns to a central flow to read in the control file (***wrf\_cntrl.parm***) which specifies which outputs are requested. The computation of the output variables begins at this point. This processing is broken into parameter type. Output of the data occurs as the data is processed. Figure 4. UPP Code Process Flow shows the UPP code process flow.

Process Input

Read ITAG

Read Input File

Read Control File

Output Requested Values

Figure 4. UPP Code Process Flow

Table 2. UPP Input Types shows the actual subroutines called to process each model and input type. UPP is currently set up to ingest WRF binary and NEMSIO binary files. They are binary files with WRF and NEMSIO headers respectively to identify what model variables each record represents. UPP also has two options to read WRF binary files, the sequential read in INITPOST\*BIN and record independent MPI IO read in INITPOST\*MPIIO. When a user makes a modification based on an input type the modifications should be propagated to all routines which are affected. When testing, all routines should be verified to ascertain that their functionality has not been compromised.

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Input Format | UPP “itag” input value | Read Routine |
| ARW / RAPR | netCDF | netcdf | INITPOST |
| ARW/RAPR | WRF binary | binarympiio | INITPOST\_BIN\_MPIIO |
| NMM | netCDF | netcdf | INITPOST\_NMM |
| NMM | WRF binary | binarympiio | INITPOST\_NMM\_BIN\_MPIIO |
| GFS | Grib | grib | INITPOST\_GFS |
| GFS | NEMSIO binary | binarynemsio | INITPOST\_GFS\_NEMS |
| NMM | NEMSIO binary | binarynemsio | INITPOST\_NEMS |

Table 2. UPP Input Types

Upon successfully reading the input into the UPP data structures, processing to compute desired outputs occurs. Table 3. UPP Calculation Routines outlines a high level representation of where variables are calculated and output.

|  |  |
| --- | --- |
| Process Routine | Description |
| MDLFLD | Called first; initializes values used throughout / computes model surface values |
| MDL2P | Interpolates model data to pressure surfaces |
| MDL2SIGMA2 | Additional interpolation of model data |
| MDL2AGL | Interpolates model data to AGL height surfaces |
| SURFCE | Handles surface based fields |
| CLDRAD | Handles sounding, cloud related, and model posted radiation fields |
| MISCLN | Handles TPAUSE level Z,P,T,U,V and vertical shear; max wind level Z,P, U and V; FD level T, Q, U and V; Freezing level Z and RH; constant mass (boundary) fields, LFM look-alike fields; NGM look-alike fields |
| FIXED | Handles time independent “fixed” fields |
| MDL2THANDPV | Interpolates model data to THETA and P surfaces |
| CALRAD\_WCLOUD | Derives model “brightness” T using CRTM |

Table 3. UPP Calculation Routines

Regression tests will be created to test capabilities based on model, input format, and type of parameter calculated. Table 4. Supported Platforms/Compilers/Data shows the types of models, data formats, platforms, and compilers that require testing.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | netcdf | | | binary | | | | | binarympiio | | |
|  |  | ARW | NMM | HWRF | ARW | | NMM | | HWRF | ARW | NMM | HWRF |
| AIX / xlf | serial | x | x | x | deprecated | | | | | NA | NA | NA |
|  | dmpar | x | x | x | deprecated | | | | | x | x | x |
| LINUX/ifort | serial | x | x | x | deprecated | | | | | NA | NA | NA |
|  | dmpar | x | x | x | deprecated | | | | | x | x | x |
| LINUX/PGI | serial | x | x | x | x | x | | x | | NA | NA | NA |
|  | dmpar | x | x | x | deprecated | | | | | x | x | x |
| LINUX/gfortran | serial | x | x | x | x | x | | x | | NA | NA | NA |
|  | dmpar | x | x | x | deprecated | | | | | x | x | x |

Table 4. Supported Platforms/Compilers/Data

1. Refactoring involves improving the design of existing code. It doesn’t change the observable behavior of the software; it improves its internal structure. Refactoring does not fix bugs or add new functionality. See <http://en.wikipedia.org/wiki/Refactoring> or Fowler,M., “Refactoring”, 2000, Addison-Wesley. [↑](#footnote-ref-1)