DESIGN OF A NEXT-GENERATION REGIONAL WEATHER RESEARCH AND FORECAST MODEL

J. Michalakes[2], J. Dudhia[1], D. Gill[1], J. Klemp[1], W. Skamarock[1]

Mesoscale and Microscale Meteorology Division[1]
National Center for Atmospheric Research
Boulder, Colorado 80307 U.S.A.

Mathematics and Computer Science Division[2]
Argonne National Laboratory
Chicago, Illinois 60439 U.S.A.
michalak@ucar.edu
+1 303 497-8199

ABSTRACT

The Weather Research and Forecast (WRF) model is a new model development effort undertaken jointly by the National Center for Atmospheric Research (NCAR), the National Oceanic and Atmospheric Administration (NOAA), and a number of collaborating institutions and university scientists. The model is intended for use by operational NWP and university research communities, providing a common framework for idealized dynamical studies, full physics numerical weather prediction, air-quality simulation, and regional climate. It will eventually supersede large, well-established but aging regional models now maintained by the participating institutions. The WRF effort includes re-engineering the underlying software architecture to produce a modular, flexible code designed from the outset to provide portable performance across diverse computing architectures. This paper outlines key elements of the WRF software design.

1 Introduction

The Weather Research and Forecast (WRF) model is a joint development effort between the National Center for Atmospheric Research (NCAR), the Forecast Systems Laboratory and the National Centers for Environmental Prediction of the National Oceanic and Atmospheric Administration (FSL, NCEP/NOAA), and the Center for Analysis and Prediction of Storms (CAPS) at the University of Oklahoma, with collaboration from scientists at a number of other universities. The model will provide a common framework for both research and operational numerical weather prediction. The WRF will be a completely redesigned code, targeted for the 1-10 km grid-scale and intended for operational weather forecasting, regional climate prediction, air-quality simulation, and idealized dynamical studies. Conditional on its merits, it will be a prime candidate to eventually supercede large, well-established but aging existing

[1] This work was supported under National Science Foundation Cooperative Agreement ATM-9732665.

[2] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

models such as the PSU/NCAR Mesoscale Model (MM5) at NCAR, the ETA model at NCEP, and the RUC system of NOAA/FSL. Collaborating institutions are interacting extensively on dynamical core and grid structure; physical parameterizations, including land-surface processes; initialization from gridded fields; data assimilation; post processing and graphics; real-time operational testing; and software architecture and portability. While specifics of the WRF architecture are subject to continued discussion and debate by scientists and software engineers at the participating institutions, consensus exists that WRF should be a modular, flexible, maintainable and extensible code designed to provide portable performance on diverse computing architectures. With the intent of stimulating further discussion toward a comprehensive WRF design, this paper describes the software architecture and other features of a prototype code now under development at NCAR.

2 Software Design

Many of the inadequacies of existing models stem from constraints imposed by languages and design methods that were in use when the models were first formulated. Typical are statically allocated data structures communicated through COMMON blocks, EQUIVALENCE data, missing or inconsistent indentation conventions, short obscure variable names, implicit typing, haphazard control structures, inconsistent subroutine interfaces, inconsistent definition of physical constants, Fortran record-blocked I/O, use of eight-byte integers to store string values and other Cray specific constructs, and vector specific loop and data structures. This legacy structure impairs readability and limits use of new modules in plug-compatible fashion. It hinders porting and parallelization without extensive modification, leading to multiple versions and unnecessary maintenance effort. Some progress has been demonstrated towards "same source" implementations of existing models on parallel platforms [5]. However, the approach involves source translation and build scripts, and parts of the model such as nesting code and I/O are not handled. WRF presents an opportunity to employ new language features and current software methodologies to construct a well-designed, maintainable, portable code from the outset.

The WRF prototype makes extensive use of modern programming language features that have been standardized in Fortran90: modules, derived data-types, dynamic memory allocation, recursion, long variable names, and free-format. A central object of a nested model is a *domain*, represented as an instance of a Fortran90 derived data type. The memory for fields within a domain are sized and allocated dynamically. The WRF prototype employs INTERFACE blocks and IMPLICIT NONE to allow compile-time checking for misspelled variable names and argument type and number mismatches: errors easily introduced by momentary lapses of attention but only painfully uncovered, if at all. To assure readability and understandability, capitalization, indentation, and commenting are being standardized. The use of Fortran90 array syntax is an open issue subject to efficiency concerns.

The WRF prototype employs a layered software architecture that promotes modularity, portability and software reuse. Information hiding and abstraction are employed so that parallelism, data management, and other issues are dealt with at specific levels of the hierarchy and transparent to other layers; more will be said about this shortly.

Other tools, such as MPI, OpenMP, single-sided message passing, and higher-level libraries for parallelization, data formatting, and I/O, will be employed as required by platform

or application; however, these are considered external to the design, and their interfaces are being encapsulated within specific layers or modules.

2.1 Parallelism: two-level decomposition

The WRF model is required to run efficiently on different types of parallel machine. These include distributed memory, shared memory (or at least shared address space), and distributed memory clusters of shared memory nodes. The software architecture for the WRF prototype addresses this issue by adopting a two-level decomposition for parallelism in which domains may be decomposed into *patches*, for distributed memory; patches, in turn, may be multi-threaded over *tiles*, for shared memory. A patch is a section of a model domain allocated to a single memory address space. It may also include memory for *halo regions*, used to store data from neighboring patches, or *boundary regions*, used to store data associated with certain physical boundary conditions (e.g. periodic boundaries). A tile is the amount of a patch that is allocated to a one thread of execution and every cell of a patch that requires computation must fall into some tile. On a distributed memory machine with single-processor nodes, there would be one patch per processor, each with a single tile. On a distributed memory machine with multiple processors per node, there would still be one patch per node (that is, per memory), but each patch may contain a number tiles equal to or greater (perhaps much greater) than the number of processors on each node. A two-level decomposition is described in [2], and others have also successfully employed two-level approaches to parallelizing geophysical models: e.g., MICOM [9] and MC2 [1,10].

Each domain in a simulation may be decomposed and tiled independently. Particular decomposition, tiling strategies, and communication packages are architecture- and application-specific and not part of the proposed WRF design. However, the initial implementation of the WRF prototype is using two-sided MPI communication to handle exchanges between patches and OpenMP to implement multithreading over tiles.

2.2 Data structures and classifications

The principal data object is the *domain type*. This is a Fortran90 derived data type containing state data fields, defined as deferred shape data arrays (actually, as Fortran90 POINTERS). The size of the memory allocated to the domains is determined by the driver at run time and depends on the logical domain size and the patch decomposition employed. The domain type also contains pointers that can be associated with other instances of the domain type; for example, parent domains, child domains, and siblings.

Model data in the WRF prototype is classified according to how it is used and its persistence. The class of *state* data, designated as *S*, persists over the life of a domain: for example, prognostic variables such as wind velocity, temperature, moisture, and pressure at several time levels. S data is stored as dynamically allocated fields in an instance of a Fortran90 derived data type for a domain. Other data in the model is considered *intermediate*, of which there are two classes. The first class of intermediate data, *I1*, is for data that persists over the duration of a time step on a domain and that must be preserved between calls to different sections of the model (tendency arrays, for example). The second class of intermediate data, *I2*, is for data that is strictly local to one section of the model and is not preserved between successive sections within a time step (for example, local working storage in a physics routine). I1 and I2 classed data are subject to automatic (stack) allocation. I1 data is defined and

allocated at the beginning of the solver for a domain, and it is allocated to be the same memory dimensions as state data. I1 data may be communicated between processors; threads have shared access to it. I2 data is defined and allocated within the local subroutine that uses it, and is defined as having the same dimensions of the tile. I2 data is private to a thread and it may not be communicated between processors. If communication or sharing is required, the data item should be promoted to I1, allocated at the solver level (part of the mediation layer), and passed to the subroutine as an argument.

There is a small amount of "global" data, primarily physical constants. These are stored in a Fortran90 module and are accessed by subroutines that need them by use-association; that is, with a Fortran90 USE statement.

To maintain consistent data use and to aid in propagating changes to data specifications, we are developing a data registry for use in developing the WRF prototype. This is a table of data elements in the model classified and described in a consistent fashion. Initially this will simply be a human-readable-only table that is maintained with the code and shared as a common reference among developers of the code. Ultimately, scripts and other code maintenance utilities will make use of the table information to automatically propagate changes to model data specifications throughout the software.

2.3 Hierarchical software design

The WRF prototype software architecture shown in Figure 1 consists of three layers: the *driver* layer, which occupies the topmost levels of the call tree; the *model* layer, which occupies the lowest; and a *mediation* layer, that sits between driver and model. All layer interfaces are through subroutine calls and through use association of Fortran90 modules. The layers encapsulate and hide details that are of no concern to other layers. For example, the low-level model layer is protected from architecture-specific details such as message-passing libraries, thread packages, and I/O libraries. The design facilitates interchangeable, reusable software; for example third-party packages for structured adaptive mesh refinement and steering may be employed without affecting the underlying model. Conversely, driver software written for the WRF model can be used for other models that conform to the interface specifications.

Driver layer. The driver is responsible for top-level control of initialization, time-stepping, I/O, instantiating domains, maintaining the nesting hierarchy between domain type instances, and setting up and managing domain decomposition, processor topologies, and other aspects of parallelism. The driver deals only with domains and does not see the data fields they contain. When a new domain is created, the driver calls the mediation layer, passing it an instance of the domain type plus memory size information. On return, memory for fields of the domain type instance is allocated. On a coding level, the definition of the domain derived data type and the routines for allocating and deallocating memory are contained in a Fortran90 module that is known to the driver by use association.

Model layer. The model layer comprises the subroutines that perform actual model computations. Model subroutines are written to be callable over an arbitrarily shaped piece of the three-dimensional model domain. No data is passed through common; a very small amount of global data (physical parameters, model-wide switches, etc.) may enter the subroutine through use association of Fortran90 modules. All state data (S) and one class of intermediate data (I1) are passed through the argument list of the model subroutine. Only primitive Fortran data types and simple arrays are passed through the argument list; state data

stored as fields in the domain type instance is dereferenced by the mediation layer prior to the call to the model subroutine. Local (I2) data is allocated on the program stack when the subroutine is called. Three kinds of dimension information are passed into the model subroutine as the last 18 integer arguments:[3]

- Starting and ending indices of each dimension of the logical (undecomposed) domain. These arguments are used in the routine for testing boundary proximity.
- Starting and ending indices of each dimension of the array in local memory. These arguments are used to dimension arrays that are passed in as arguments to the routine.
- Starting and ending indices of the tile dimensions. Used in loops within the subroutine. These are also used to dimension local (I2) storage.

Routines in the model layer are callable for a single tile, that is, the section of a domain allocated to a single thread. The extent of one call to a model routine is also bounded by the amount of computation that can be completed without concern for coherency before horizontal dependencies arise. When that occurs, the model routine returns to the driver/mediation layer, which communicates, synchronizes, or otherwise ensures coherency, then passes control back to the next routine in the model. It is hoped that this structure will induce model layer developers to identify and make explicit the coherency issues in the code. Computation on halo regions is allowed and is specified by the driver/mediation layer, which increases the span of the tile dimensions by the necessary amount when the model routine is called. An open question at present is whether and how to allow model routines themselves to perform halo computations without introducing too much driver information into the model layer.

*Mediation layer***.** This layer mediates between the model and driver layers. The mediation layer contains information pertaining to both the model layer and the driver layer: model-specific information such as the flow of control to compute a time step on a domain and driver-specific mechanisms such as tiling and communication.

Information the model layer knows that the driver layer needs is provided through query routines that the model makes available to the driver. These include routines for determining recommended halo widths, boundary region memory requirements, communication stencil widths and/or widths of replicated halo computations for a model subroutine. These routines are also considered part of the mediation layer. The Fortran90 module that contains the domain type definition and routines for allocating fields within the domain are also part of the mediation layer.

3 Outstanding issues

The WRF design is still preliminary – dry prototypes of the dynamical model are being constructed, allowing testing of an actual implementation -- and a number of important issues remain.

External packages. For a machine with a single processor, address space, and thread of control, the WRF model is essentially complete. Otherwise, the model will certainly rely on

[3] In the present implementation, these are passed as separate arguments. Discussion continues on ways to streamline this process, perhaps through a type definition or through the use of TASK-COMMON (which would force an exception to the "no COMMON" rule).

other tools, libraries, and packages depending on the needs of the application and on the architecture. MPI and MPI-2 are likely to be used in distributed memory settings, and it is likely that higher level application libraries such as RSL [6] or NNT/SMS [8] will also be deployed. On symmetric multiprocessors and machines composed of SMP nodes, OpenMP will play an important role in specifying multi-threading. Code frameworks such as Overture [3], LPARX [4], and Dagh [7] are attractive for their emphasis on adaptive mesh refinement and multi-grid solvers, and could be layered into WRF at the driver level; however, compatibility with Fortran90 is an issue that must be explored. Fortran record-blocked I/O will almost certainly need to be abandoned in favor of byte-stream I/O, which is easier to parallelize and compatible with existing parallel I/O packages such as MPI-2 and SMS. Meta-data formats such as NetCDF and HDF are attractive for interoperability, but overheads are a concern. The proposed WRF design neither includes nor endorses any particular package, requiring only that packages be well-encapsulated and interchangeable.

Generality and cost. The proposed WRF design strives for generality with respect to hardware and architecture specific coding. Tile-based expression of subroutines makes the size and shape of what a thread computes a run-time driver-level concern, with potential benefits for cache-locality, load balancing, and vector length. In this vein, one would also wish to control array index order and loop ordering/nesting on a per-architecture basis; however, there are no language facilities for this. Additional help from source translation tools would be required to, say, permute all the definitions of three-dimensional arrays from *i,j,k* to *k,i,j* or *i,k,j* and then to make corresponding changes in the loop structure. Based on early discussions, a *k*-innermost index ordering for three-dimensional arrays has been chosen for the prototype, since there are demonstrated performance benefits on cache-based processors, while at the same time, vector systems have become sufficiently sophisticated to handle without penalty inner loop stride-lengths greater than one. Another performance concern is the cost of dereferencing fields from domain data structures. It is hoped that since dereferencing is done at a high level in the call tree, the amount of work done per dereference will offset the additional overhead. Likewise, calling overhead stemming from the decision to pass all data through argument lists is a concern. Once a prototype code is available, design features with the potential to affect performance will be tested extensively and, if costs are found to be unacceptable, revised.

4 Summary

The design for a WRF prototype code sketched in this paper uses modern programming language features standardized in Fortran90 for constructing modular, flexible, and maintainable software. The proposed software architecture is layered functionally to provide abstraction and encapsulation, facilitating porting to different architectures and compatibility with alternative solvers, physics, and external software packages and parallel libraries. The WRF development effort provides a rare opportunity to reconsider, in collaborative fashion, basic software architecture issues and, free from hindrance of legacy structures and methodologies, to design a next-generation regional model.

5 References

1. M. Desgagne, S. J. Thomas, R. Benoit, M. Valin, and A.V. Malevsky, *Making its Mark*, World Scientific, River Edge, New Jersey (1997), pp. 155—181.
2. I. Foster and J. Michalakes, *Parallel Supercomputing in Atmospheric Science*, World Scientific, River Edge, New Jersey (1993), pp. 354—363.
3. W. D. Henshaw, Los Alamos National Laboratory Report LA-UR-96-3894 (1998).
4. S. R. Kohn, and S. B. Baden, in Proceedings of Supercomputing '95, IEEE Computer Society Press (1996).
5. J. Michalakes, in Proceedings of the Second International Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applications, NASA Preprint GSFC/CP-1998-206860 (1998), pp. 129—139.
6. J. Michalakes, Argonne National Laboratory Tech. Report ANL/MCS-TM-197 (1994).
7. M. Parashar, M. and J. C. Browne, in Proceedings of the International Conference for High Performance Computing (1995) pp. 22--27.
8. Rodriguez, B., L. Hart, and T. Henderson, *Coming of Age*, World Scientific, River Edge, New Jersey (1995) pp. 148--161.
9. A. Sawdey, M. O'Keefe, and W. Jones, *Making its Mark*, World Scientific, River Edge, New Jersey (1997), pp. 209—225.
10. S. J. Thomas, M. Desgagne, R. Benoit, and P. Pellerin, in Proceedings of the Symposium on Regional Weather Prediction on Parallel Computers, University of Athens, Greece (1997), pp. 33—42.

<table>
<tr><td>Layer</td><td colspan="2">DRIVER</td><td>MEDIATION</td><td>MODEL</td></tr>
<tr><td>Sub-layer</td><td>Top Level</td><td>Domain Hierarchy Layer</td><td>Solver, Coherency, Dereferencing</td><td>Model Subroutines</td></tr>
<tr><td>Control</td><td>One model run</td><td>One time step</td><td>One Domain</td><td>One tile</td></tr>
<tr><td rowspan="3">Data
State</td><td rowspan="3"></td><td colspan="2">Domain data type; heap-allocated and passed between Driver and Mediation layers as a structure</td><td>Individual fields passed as arguments</td></tr>
<tr><td>I1</td><td colspan="2">Automatic (stack) data in SOLVE</td><td>Individual fields passed as arguments</td></tr>
<tr><td>I2</td><td colspan="2"></td><td>Allocated automatically in model subroutine</td></tr>
<tr><td rowspan="2">Functions</td><td>Initialization</td><td>Creates, destroys instances of Domain type
Calls Mediation layer to allocate Domain data fields</td><td>SUBROUTINE ALLOC_DOMAIN (domain, dimensions)
ALLOCATE(domain%field1(dimensions))
ALLOCATE(domain%field2(dimensions)) . .
.</td><td>SUBROUTINE QUERY FCN (…)</td></tr>
<tr><td>Time Loop</td><td>Steps all domains in nest hierarchy:
STEP(LEVEL)
Foreach domain at LEVEL
Call SOLVE(domain)
Recurse: STEP(LEVEL-1)</td><td>SUBROUTINE SOLVE (domain, dimensions)
Communication or synchronization
Dereference and call BIG STEP
Communication or synchronization
Minor time loop
Communication or synchronization
Dereference and call SMALL STEP
End minor time loop
Communication or synchronization
Dereference and call PHYSICS</td><td>SUBROUTINE PHYSICS (u, v, …, &
ids, ide, ims, ime, its, ite, &
jds, jde, jms, jme, jts, jte, &
kds, kde, kms, kme, kts, kte)
REAL, DIMENSION(kms:kme, ims:ime, jms:jme) :: u, v
REAL, DIMENSION(kts:kts, its:ite, jts:jte) :: local
...
CALL SUBS (…)</td></tr>
</table>

Figure 1. Hierarchical software design being implemented in the WRF prototype. The three main layers, Driver, Mediation, and Model and their functions are outlined from left to right. The three classes of data, State, Intermediate-1, and Intermediate-2, are shown at the levels in which they reside. State data exists as structures (Fortran90 derived data types) at higher levels of the hierarchy and as individual fields in the model layer. Functional descriptions and representative pseudo-code fragments are shown on the last rows of the table. Intermediate-1 data, which persists for the duration of a time step, is stack-allocated in the model "solve" routine, part of the Mediation layer. Intermediate-2 data, which exists only for the duration of a particular call to a model subtree, is stack-allocated in the called subroutine. The Mediation layer, which contains the top-level flow of control over a single time step on a domain, also contains the calls to communication libraries and/or multi-threading directives, indicated above as "communication or synchronization." The "physics" subroutine interface shown in the Model layer is part of a uniform template that specifies domain, memory, and tile dimensions separately and explicitly, allowing model subroutines to be called for arbitrarily shaped tiles. This gives the Driver and Mediation layers great flexibility in organizing the two-level decomposition for parallellism without changing the underlying model code.