Weather Research and Forecasting Model 2.2 Documentation: A Step-by-step guide of a Model Run

Technical Report FIU-SCIS-2007-09-02
August 2007

Faculty Advisor:

S. Masoud Sadjadi

Graduate Students:

Javier Munoz, Diego Lopez, and David Villegas

Undergraduate REU Students:

Javier Figueroa, Xabriel J. Collazo-Mojica, Michael McFail, Alex Orta

School of Computing and Information Sciences (SCIS)

Florida International University (FIU)

11200 SW 8th St., Miami, Florida 33199, USA

Contents

Project Motivation / Abstract ... 3
1. Introduction ... 4
  1.1 The wrf.F file ... 5
  1.2 Namelist.input file ... 6
  1.3 WRF Preprocessing ... 7
  1.4 Run-time System Library (RSL) ... 7
2. WRF initialization stage ... 8
  2.1 Module Initialization ... 8
  2.2 Domain Decomposition ... 9
  2.2 I/O Quilting Initialization ... 10
  2.3 Namelist Configuration ... 15
  2.4 The TYPE (domain) Data Structure ... 15
  2.5 Root Domain Allocation and Configuration ... 16
3. WRF run stage ... 19
  3.1 The Integrate Subroutine ... 19
  3.1.1 Time-Keeping Overview ... 20
  3.1.2 Integrate's Flow of Control ... 21
  3.1.3 Integrate's Time Advance Loop In WRF ... 22
  3.1.3.1 Inside the solve interface subroutine ... 24
  3.1.4 Integrate's Recursive Loop For Nesting In WRF ... 25
4. WRF Finalization Stage ... 29
5. Graphs ... 30
  5.1 Integrate Subroutine ... 30
  5.2 Partial WRF Directory Graph ... 31
6. Glossary ... 32
7. References ... 35

Project Motivation / Abstract

In Summer 2007, we, a group of four undergraduate students and three graduate students under the supervision of Dr. Masoud Sadjadi at School of Computing and Information Science (SCIS) of Florida International University (FIU), started an effort on gridifying[1] the Weather Research and Forecast (WRF) code. During this process, it became apparent to us that we needed a better understanding of the code's functionality before we start the gridification process. As the available documentation on WRF was not targeted for developers like us who would need to modify the code operation to adapt it to a grid computing environment (and not just adding a new physics model, for example), we were pushed to search through lines of the WRF's FORTRAN and C code to discover how this code was actually functioning; especially, in parts such as domain decomposition and network interactions among the nodes. Due to the large and complex nature of the WRF code, documentation of the program flow proved necessary. With more time and thought, we decided to start a documentation effort to be useful not only for us, but also for other interested in learning the WRF operation in more dept. This guide should help developers understand basic concepts of WRF, how it executes, and how some of its functions branch into different dimensions. We hope that by the time our audience finishes reading this document they will have gained a strong understanding of how WRF operates.

[1] By "gridifying" here we mean the process of enabling a software program to run on a grid computing environment. We also refer to this process as the "grid enablement process" throughout this document.

1. Introduction

This tutorial is an introduction to the WRF (Weather Research and Forecasting Model) [1] code. Basic knowledge of High Performance Computing [2] and MPI [3] is assumed. This document should be helpful for developers in the community who want to contribute to the WRF project. It will be of particular interest to those new to the code base that will be facing the challenge of working with it for the first time, and so will need as much documentation as possible in order to become familiar with the code. First, a small overview of WRF and its software architecture is required. WRF is composed of three software layers: the model layer, the driver layer, and the mediation layer. The model layer contains the physics modules that contain the logic for the simulation itself. This is the highest level of abstraction and the lowest level in the call tree. The driver layer controls the low level details of how the physics code is efficiently executed. It deals with things such as parallelism, memory allocation, and I/O. The mediation layer acts as a bridge for the model and driver layers. It allows them to communicate through a series of well-defined interfaces [4] which can be seen in Figure 1.

In order to understand how WRF works certain terms need to be defined. A grid is a set of three dimensional points in space that contain weather data such as wind speed, atmospheric pressure, etc. Each grid has a current time and a stop time associated with it. WRF simulates the atmosphere by physics calculations based on the grid data and a specific physics model, and then advances the grid's current time by a unit of time called a time-step. It does this in an iterative manner until the grid's current time equals the stop time. Each grid (or domain) can have a nested domain (or nest) within it. A nested domain is another grid, a subset of the current one. A parent is the grid which contains a nest. This nested domain may also have nested domains within it. WRF currently has a limit of twenty nests. A nested domain may (and almost certainly will) have a different resolution than its parent. Resolution refers to how far apart the three dimensional points are from each other. All this greatly complicates the code, but it does add power and flexibility to the WRF model. A brief introduction to the domain data structure is also in order.

The MOAD (or Mother Of All Domains) is the primary grid, and is the largest domain in the simulation. The MOAD is capable of having any number of children (nested domains), but does not have a parent or any siblings. A sibling is a domain that has the same parent as another domain, for example two children of the MOAD are siblings. For this document we are using

version 2.2 of the ARW WRF code compiled for MPI using RSL LITE. WRF main executable files assume that the configuration and domain setup has been done previously by using WRFSI [5] and Robert A. Rozumalski's [6] Perl scripts.

Finally we want to make reference to our graphs section where two important graph are located. The graph 5.1 is a flow chart of the integrate subroutine and the graph 5.2 contains some of the WRF directories more especially the ones that are mentioned throughout this document. These graphs serve as aid for the understanding of the program's flow. Following the graphs while reading will make understanding WRF easier.

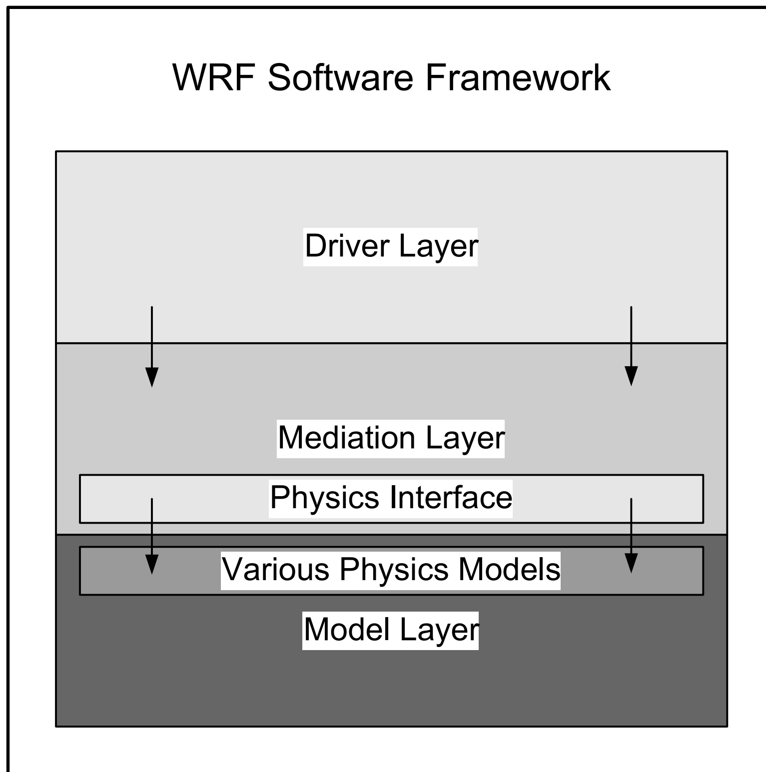


Figure 1 - The WRF layer structure

1.1 The wrf.F file

The main file in WRF is called wrf.F and is located in the main folder of the WRF installation directory (Graph 5.2 block 1). This file is the main program that initializes the WRF

model. It is responsible for starting up the model, reading in configuration data (and broadcasting for distributed memory), initializing the top-level domain (either from initial or restart data), setting up time-keeping, and calling the integrate routine to advance the domain to the ending time of the simulation. After the integration is completed, the model is properly shut down. This documentation will focus largely on the WRF run stage and its functions. Calls to routines that might not relate directly to these topics may be ignored. We also suggest that the reader follow the included graphs. They will provide a better understanding of the flow of control of the program.

1.2 Namelist.input file

The namelist file is divided in four main sections - &time_control, &domains, &physics and &dynamics. The &time_control is the section where all the time configurations are set for the domain to be run. By editing this section many options can be set, including the run time in days, time interval for the run, and/or restart time. The &domains section refers to any domain configuration variable. In this section there are options like time_step to set the time step of the run and max_dom which indicates the maximum number of nests for the run. If set greater than 1,WRF assumes a nested run. One option that stands out in this section is the feedback option. If this option is set to 0 then no feedback is produced in the run. This means a 1-way nesting WRF run will occur. One-way nesting may be used if a lower temporal frequency of coarse-to-fine boundary forcing is acceptable and/or preferable, or if nested forecasts are to be run as separate WRF jobs [7]. Later in this document we will discuss nesting in WRF. When using the word nesting, from now on we are referring to a 2-way nesting process. The last sections of the namelist file (&physics and &dynamics) are targeted to modify different physics options. These options are used by meteorologists and are outside the scope of this document. In the model run explained by this guide, the namelist file is created after using the WRFSI program right before running WRF.

1.3 WRF Preprocessing

In order to start the numerical weather calculations, the WRF model needs to first have initial conditions data. In this preprocessing stage we have several steps. First we need to get this needed data from a source. As of today this is accomplished via FTP to a various NOAA web servers. The data is available in the form of tiles covering specific parts of the world. There are scripts available [6] to automate this, like Bob Rozumalski Perl scripts. This data is downloaded in a concise data format named GRIB (for GRIdded Binary) [8]. Since WRF uses another more general scientific format called netCDF [9] internally, the second step involves data conversion. This step includes horizontal and vertical interpolation to create the WRF computational domain. Note that other data is also needed such as terrain elevation, physical characteristics of the area, etc. This data should also be downloaded if it hasn't been downloaded before. After all this data is gathered, the WRF model is ready to start the running stage.

1.4 Run-time System Library (RSL)

RSL is the current parallel library used by the WRF. It acts like a 'wrapper', giving a high-level view to communications based on low-level MPI subroutines. Note that RSL is not part of WRF. It is an independent library that complies with WRF's internal interface for communications. It is responsible for the following [10]:

- Inter Domain Communication - Basically using MPI to send messages from one node to other.
- Domain Decomposition - The first decomposition of load among all available nodes.
- Local / Global index translation - Handles the issue of indexes between a local node and the whole domain context.
- Dynamic load Balancing - RSL detects if performance is not as high as it should. If there are issues, it dynamically redistributes the load.

WRF does not have to use RSL, but at this time it is the only implemented communications package. The same development team created both RSL and WRF.

2. WRF initialization stage

The wrf.F file first calls the routine wrf_init which is located at module_wrf_top.F (Graph 5.2 block 4). Some key distributed-memory initializations happen here if the "DM_PARALLEL" flag is specified during compilation. A distributed memory run is needed running the program in a grid environment. This uses RSL, as discussed in the previous section. External communication packages such as RSL or RSL_LITE are also initialized in the wrf_init routine. I/O initialization, including quilting initialization for clients and servers, occurs here. Namelist configuration information is read and distributed to every node. The root domain is allocated, configured, and decomposed. Figure 2 provides a graphical view of the calls from wrf_init. It will be referred to throughout the section.

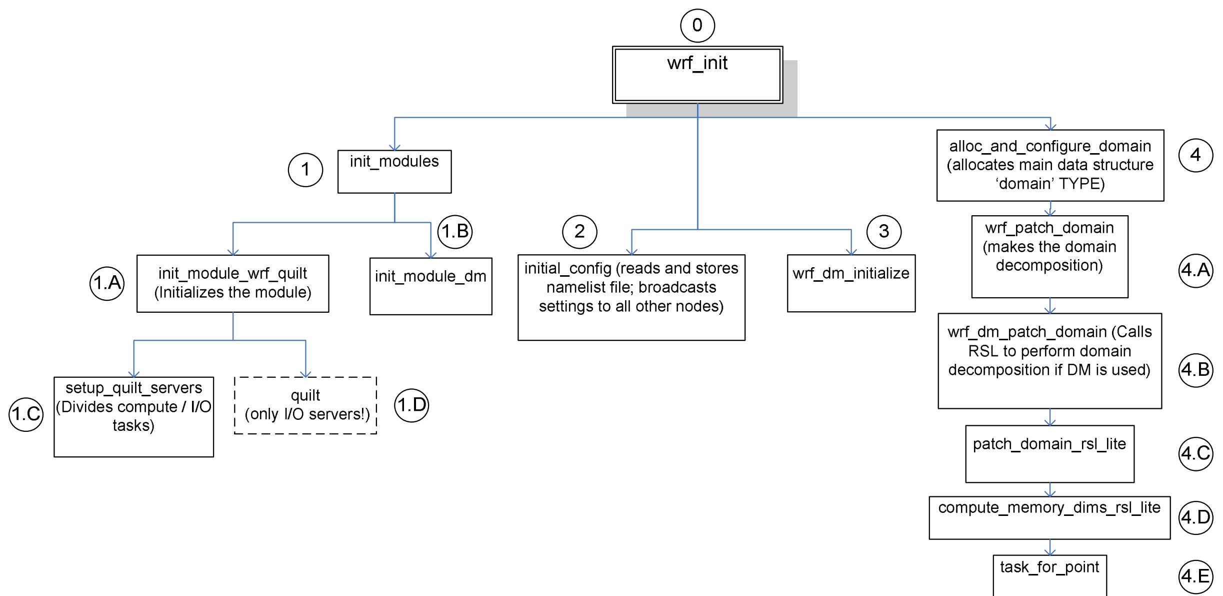


Figure 2 - The wrf_init subroutines calls

2.1 Module Initialization

Wrf_init calls the init_modules first, as shown in Figure 2 Block 1. This call is performed to initialize all the modules prior to the run of the simulation. The init_modules subroutine is

located init_modules.F (Graph 5.2 block 4). The init_modules subroutine takes an integer argument, phase, which specifies which phase of module initialize should occur. If phase is equal to the value 1, then several WRF module specific initializations occur, including setting up model and driver constants. If WRF was compiled with the "DM_PARALLEL" flag, the subroutine init_module_wrf_quilt (Figure 2 block 1.A) is called. After init_module_wrf_quilt returns to init_modules, init_module_dm is called to perform initializations related to distributed memory. MPI is initialized and the distributed memory communicator is set to MPI_COMM_WORLD. At this point Phase 1 ends. Phase 2 performs initialization after the call to MPI_INIT. This includes the initialization of the nesting, tile, I/O, and core specific modules (depending upon the specified physics options). A tile is a piece of a domain which is worked on by a node. The purpose of the phase argument is to allow other superstructures, such as the Earth System Modeling Framework (ESMF) [11], to perform their calls to init_modules after MPI initialization has occurred.

2.2 Domain Decomposition

WRF handles domain decomposition (splitting the domain between nodes) in various distributed memory subroutines. Domain decomposition happens in two stages. The domain is first split into patches, which are then distributed one per node. Assuming multiple processors per node, the patch is then broken down into tiles (which are shared memory), each of which is given to a thread to work on. In this way distributed memory (patch) and shared memory (tile) decomposition are kept separate. So, for example, if a WRF run was being done on a cluster of five nodes, each with two processors, the domain would be decomposed into five patches, each of which would then be split into two tiles. This section is concerned only with patch decomposition; tile decomposition for shared memory is not dealt with here.

The number of nodes to use in the x and y direction (for patch decomposition) is determined via a call from wrf_init to wrf_dm_initialize (Figure 2 block 3). RSL divides the domain into partitions and attempt to ensure that every partition has an equal number of points. Each point of the domain can be allocated independently, allowing irregularly shaped processor nested domains [12]. The subroutine patch_domain_rsl_lite (Figure 2 Block 4.C) calls the compute_memory_dims_rsl_lite (Figure 2 Block 4.D), which computes the patch and memory

indexes for this node. When determining if a grid point belongs on the current node the C function task_for_point (Figure 2 Block 4.E) is used. Grid points are assigned to nodes in a straight forward fashion. If there are n grid points in the x direction and t tasks each task gets n/t grid points, with extra grid points distributed evenly from each boundary towards the middle. A trivial example illustrates:

If there are 19 grid points (labeled 1 to 19) and 5 nodes (labeled A, B, C, D and E) then node A works on grid points 1 to 4, node B works on grid points 5 to 8, node C works on grid points 9 to 11, node D works on grid points 12 to 15 and node E works on grid points 16 to 19.

The extra work (the grid points that don't divide evenly among the nodes) is distributed near the boundaries because those processes have less work to do and should be able to handle the slightly increased load. The same distribution occurs for the y direction, leaving each node with a rectangular section of grid points to work on. The range for each node is then extended to encompass the halo regions. The specified physics core changes the way in which domain decomposition is handled. By default the domain runs from the start index to the end index in each direction. When NMM (Glossary 6.2.2) is used, however, the domain only runs from start index to end index - 1. This minor change is reflected in the domain decomposition so that grid points are still distributed as evenly as possible between the nodes. The MPI communicator, which allows the nodes to communicate via distributed memory calls, is setup in the subroutine wrf_dm_initialize. It is used later by many routines, including integrate.

2.2 I/O Quilting Initialization

Quilting was introduced in WRF version 1.2. It allows computation to run with less I/O interruption by specifying a number of additional servers whose sole task is to collect and write model output. This frees client machines to spend more time on model computation [13]. Quilting setup takes place in phase 1 of init_modules with a call to init_module_wrf_quilt. This subroutine is called by both quilt clients and quilt servers in order to setup quilting. It must be called before initializing MPI or quilting will not work and a fatal error will occur. It is located inside of module_io_quilt.F (Graph 5.2 block 3).

Init_module_wrf_quilt (Figure 2 block 1.A) calls setup_quilt_servers (Figure 2 block 1.C) to determine if the current process is a client or a server. This subroutine also sets up MPI

communication groups for the I/O servers. Every I/O server is in an MPI communications group with other I/O servers. By default, the number of I/O server groups is set to 1, meaning that all I/O servers communicate with each other and each compute task only deals with one I/O server. If there is more than one server group each server talks only to the other servers in its group, and each communication task talks to one server in each group. The number of I/O server groups can be changed by altering the namelist configuration option nio_groups.

Figure 3 below illustrates quilt clients connected to quilt I/O servers in one and two groups respectively. The first diagram has nodes 12 to 17 all acting as I/O servers in a single group. All of the I/O servers talk to each other, and each client talks to one of the I/O servers. The second diagram has the servers split into two groups, nodes 12 to 14 are in one server group and nodes 15 to 17 are in another server group. Server to server communication happens only within a group and each client talks to one server in each group.

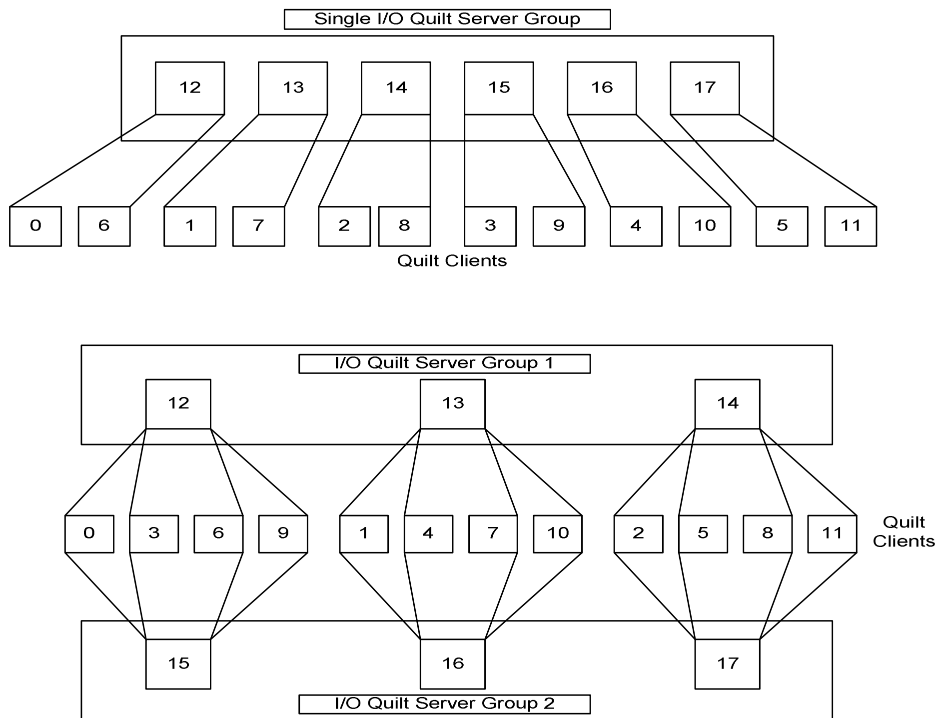


Figure 3 - Quilt I/O servers in one and two groups

The number of servers per I/O group is set by the namelist option nio_tasks_per_group. By default it is 0, indicating no quilting will take place. Values that are greater than 0 indicate how many servers should be assigned to each I/O group. There must be at least as many

computation tasks as I/O tasks in every server group. If this requirement cannot be satisfied because there are less computation tasks than I/O tasks, quilting cannot be done and the run continues without quilting. The I/O clients and servers are assigned in a simple round robin fashion, with no notion of network topology or the differences between nodes.

The communication group MPI_COMM_LOCAL is setup in this subroutine. It is used as the distributed memory communication group by the integrate routine, among others. Its membership is set depending on whether the current task is a compute task or a server task. If it is a compute task the communicator is set to the group of all compute tasks. If it is a server task the communicator is set to all the servers in the current server's group. By default this is all I/O servers, since there is only one server group. In the event quilting is not used it is set to MPI_COMM_WORLD, the group of all nodes. Additionally, the communication group MPI_COMM_IO_GROUPS is setup in setup_quilt_servers (Figure 2 block 1.C). This array contains communication I/O groups of interest to the current process. For a compute task each element in the array is one of the I/O server groups. The group contains the computing tasks in the server group and the server itself as the last element. For a server task only the first element in the array is used. It is a communication group that contains all of the servers in the current server's I/O group (by default all of the I/O servers, since there is only one group). Upon return to init_module_wrf_quilt servers will call the quilt method (Figure 2 block 1.D) and remain there for the duration of the run. Clients will return from init_module_wrf_quilt and perform computational tasks. An example from the code documentation illustrates with 18 tasks, nio_groups = 2, and nio_tasks_per_group = 3 (see also the two group example from Figure 3 above):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Compute Tasks	X	X	X	X	X	X	X	X	X	X	X	X						
1st I/O Server Group													X	X	X			
2nd I/O Server Group																X	X	X

Membership for MPI_COMM_LOCAL communicators:

	Tasks																		
Tasks Included In Group		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	Compute Tasks 0, 3, 6, 9	X			X			X			X			X					
	Compute Tasks 1, 4, 7, 10		X			X			X			X			X				
	Compute Tasks 2, 5, 8, 11			X			X			X			X			X			
	I/O Server Task 12	X			X			X			X			X					
	I/O Server Task 13		X			X			X			X			X				
	I/O Server Task 14			X			X			X			X			X			
	I/O Server Task 15	X			X			X			X						X		
	I/O Server Task 16		X			X			X			X						X	
	I/O Server Task 17			X			X			X			X						X

Membership for MPI_COMM_IO_GROUPS(1):

	Tasks																		
Tasks Included In Group		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	Compute Tasks 0, 3, 6, 9	X			X			X			X						X		
	Compute Tasks 1, 4, 7, 10		X			X			X			X						X	
	Compute Tasks 2, 5, 8, 11			X			X			X			X						X
	I/O Server Task 12	*Not Used*																	
	I/O Server Task 13	*Not Used*																	
	I/O Server Task 14	*Not Used*																	
	I/O Server Task 15	*Not Used*																	
	I/O Server Task 16	*Not Used*																	
	I/O Server Task 17	*Not Used*																	

Membership for MPI_COMM_IO_GROUPS(2):

I/O quilt servers will make a call to the quilt subroutine (Figure 2 block 1.D) and will remain there for the rest of the simulation run. These servers receive I/O requests from compute tasks. They perform package-dependent I/O routines to satisfy the client requests. I/O package initialization for clients occurs later in wrf_init, but for servers it occurs within the quilt method. The first thing that quilt does is initialize specific I/O packages. For example, if the NetCDF libraries [9] are being used then a call is made to ext_ncd_ioinit, an external subroutine located in wrf_io.F (Graph 5.2 block 2). The I/O servers then enter an infinite loop where they handle I/O related requests from compute tasks (clients).

2.3 Namelist Configuration

The call to initial_config (Figure 2 block 2) located at module_configure.F (Graph 5.2 block 3) reads the namelist.input file and stores it in the model_config_rec structure. In a distributed-memory environment the namelist information is broadcasted by the root node upon the return to wrf_init. RSL is a requirement when using nesting; therefore the program must ensure RSL is available if it has been asked to do a multi-domain run. It does this by examining the namelist parameter max_dom, the maximum number of domains for the run. If max_dom is greater than one (indicating nesting) then the program checks to see if the system is configured for a parallel run (by checking the DM_PARALLEL flag) or if the system is a uni-processor that has a non-MPI build which supports nesting (by checking to see if the -DSTUBMPI option was specified). If at least one of these is specified then execution continues normally, otherwise a fatal error occurs.

The initial_config subroutine is responsible for reading in the namelist file into the module_config_rec, which contains all the of the namelist information for this run. The data type model_config_rec_type is stored module_configure.f90 (Registry Generated File) and is auto-generated by the registry. It simply contains fields for all of the values in the namelist file. Values which apply to the entire run are stored as scalars. Values which can vary for every domain are stored as arrays of size max_domains, which is defined in module_driver_constants.F (Graph 5.2 block 3). Subroutines to access the various namelist values begin with nl_get or nl_set, followed by the variable name. To see how the namelist file looks like visit the website referenced [14].

2.4 The TYPE (domain) Data Structure

To obtain a solid understanding of WRF, we consider that further explanation of the TYPE (domain) data structure is needed. Our grid of data, the parent or nested grid, is represented internally as a ‘domain’ data TYPE. TYPEs in FORTRAN are much like structs in C. The domain TYPE is specified in the file module_domain.F (Graph 5.2 block 3). The data structure is essentially a tree representation of the domain area. Each node represents a domain as seen in Figure 4.

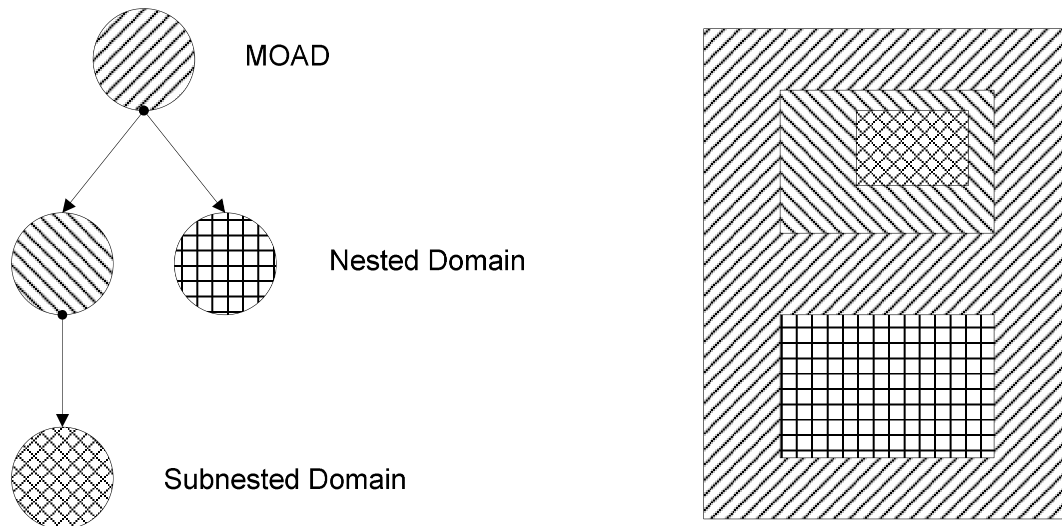


Figure 4 - ‘Domain’ TYPE data structure

It contains the meteorological data arrays needed to do the calculations in its level. The MOAD is on the root level as seen in Figure 4. Below MOAD there are nested domains. Note that siblings can not overlap, but they can have multiple nests below them (Subnested Domains). Overlapping is not permitted because of the way in which the WRF system computes information in the physics modules. It advances the calculations in time steps. If siblings overlapped then calculations will be done twice in same grid points of the simulation. This would waste computation time and even worse, create more rounding errors than those which are implicit in floating point calculations. The domain type itself contains several elements that are important to a domain. These include pointers to any parent, siblings, or nests (children) that may exist. Communicators for distributed memory as well as patch and tile information are also stored here. The clock with the current simulation time for the domain is also kept in this data structure.

2.5 Root Domain Allocation and Configuration

Now that we have a better understanding of the TYPE (domain) structure we can continue analyzing the code. After the max_dom check (see above) the program starts allocating the root domain using the communications package chosen at compile time (e.g. RSL, RSL Lite,

etc.). The top-most domain in the simulation, the MOAD or root domain, is allocated and configured by calling alloc_and_configure_domain (Figure 2 block 4), a function located in module_domain.F (Graph 5.2 block 3). Since this particular call is initializing the root domain (as opposed to nested domains, which will be initialized later) the routine is passed the globally accessible pointer to TYPE (domain), head_grid, defined in module_domain.F (Graph 5.2 block 3). The MOAD never has a parent domain, and at this point it has no children. RSL cannot deal with more than 1023 grid points in either the x or y direction, so a check occurs to make sure the domain is not too large. If there are too many grid points a fatal error occurs. Note that RSL Lite does not have this limitation. This is why RSL Lite is the preferred distributed memory communications package for a WRF run, suggested by NCAR [15].

Inside the call to alloc_ and_configure_domain there is a call to wrf_patch_domain (Figure 2 block 4.A), located in the same module. This routine is in charge of passing on the configuration information of the domain to RSL through a call to wrf_dm_patch_domain (Figure 2 block 4.B) located in the RSL library, module_dm.F (Graph 5.2 block 2). Once inside the RSL subroutines the system calculates the domain decomposition for all of the nodes in the simulation via a call to patch_domain_rsl_lite. RSL is the mechanism that handles domain decomposition (see section 2.2). It provides high-level stencil and inter domain communication that will decompose and allocate the domain by mapping each grid cell of the domain to a processor so that each processor has piece of every domain in the model.

After the call to wrf_patch_domain within alloc_and_configure_domain the subroutine alloc_space_field is called. This subroutine actually allocates memory for the various arrays that compose the domain by using #include files that were generated by the registry for particular dynamical cores (NMM, EM, etc.) [16]. The subroutine wrf_dm_define_comms is then called from alloc_and_configure_domain. This provides a package independent way for the communications package being used to perform initialization for stencil exchanges and other inter-node communication [16]. In RSL Lite this is not used; wrf_dm_define_comms is implemented as a stub method.

When control returns to wrf_init from alloc_and_configure_domain I/O initialization needs to take place. This is done with a call to the subroutine init_wrfio. It in turn calls the wrf_ioinit subroutine, which has two main tasks. It initializes the I/O handles WRF will use and makes a series of I/O package specific initialization calls. For example, if the NetCDF libraries

are being used then a call is made to ext_ncd_ioinit, an external subroutine located in ../external/io_netcdf/wrf_io.f90. The series of I/O initialization calls within WRF are not very complicated because most of the I/O initialization that needs to occur is package specific and is handled by the various external calls made by wrf_ioinit.

3. WRF run stage

Returning to the main wrf.F file one can see that after the top-level domain is initialized and allocated in the wrf_init routine a call to wrf_run is made. This subroutine contains the model of integration for WRF. The start and stop times for the domain are set to the start and stop time of the model run, and the integrate module located in module_integrate.F (Graph 5.2 block 3) is called to advance the domain forward by the specified time interval. This routine is a top level routine that provides domain nesting functionality in WRF. Nesting is used to increase resolution over portions of a domain. It is done by arranging a fine-resolution domain within a coarse resolution domain by and forcing and feeding back data between the two. John Michalakes [17] from NCAR provides a good pseudo code algorithm for the nesting code included in the integrate routine:

```
Parent domain definition and initialization.
Nested domain definition and initialization.
Loop over time.
    Advance parent domain one time step.
    Transfer parent domain state data to force the nest
    Loop over nest time steps.
       Advance nested domain one time step.
    End loop.
    Transfer nested domain state data back to parent domain.
    If it is time, perform model output for both parent and nest.
End Loop.
```

After the simulation is completed, a Mediation Layer-provided subroutine, med_shutdown_io is called to allow the model to do any I/O specific clean-up and shutdown, finally the Driver Layer routine wrf_shutdown is called to end the run, including shutting down the communications, most communication layer will call MPI_FINALIZE if they are using MPI.

3.1 The Integrate Subroutine

This subroutine it's a bit hard to follow due to its recursive nature, but a step by step walk-through of the code should simplify the more difficult parts. We will go into more detail about this routine because of its importance and the fact that it is the originator for nesting in WRF.

```
CALL integrate ( head_grid , head_grid%total_time_step+1 ,
head_grid%time_step_max )
```

The integrate subroutine takes three arguments: the domain to be integrated, a starting time step, and an ending integer time step. The field total_time_steps is the time step counter of the domain. At the beginning of the model's run the counter for the domain is set to zero. Time_step_max is the last time step to execute.

Integrate is a recursive subroutine, which means it can call itself to integrate nested domains. Integrate passes the nested domain to itself and a starting and ending series of nested steps (in this case the small number of steps to bring the nest up to the current time level). Thus, the top level call to integrate from WRF advances the principal domain and all nests from the starting time through to the end of the simulation. A simplified algorithm for integrate is as follows:

```
Procedure INTEGRATE ( grid, start step , end step )
FOR current step <------ start_step to end step
      If current step is time to open a nest, open and initialize a nest
      If current step is time to deactivate nest, deactivate
      If current step is time to do domain output, output domain
      Advance grid one time step( call SOLVE )
      FOREACH active nest associated with grid
            Recursive CALL integrate ( nest, ( current step-1)*nest ratio ,
            current step * nest ratio )
```

This algorithm implements a depth first traversal of the tree of nested domains rooted at the principal domain. Not shown is some additional code for dealing with overlapping (which are handled breadth-first, through sibling pointers in the domain derived data type) [16]. Please use the diagram at section 5.1 to follow graphically the explanation below. Note that this diagram is presented using static UML modeling, but as we know, Fortran is not an object-oriented language. Please understand the compromise made for ease of explanation.

3.1.1 Time-Keeping Overview

Integrate starts by taking a domain pointed by the grid argument and advancing the domain and all its nested domains from the grids current time stored at grid%current_time until it

reaches the given time in the simulation stored at grid%domain_clock. The routine checks this by calling WRF_UTIL_ClockIsStopTime prior to beginning the loop over time period that is specified by the current_time/stop_subtime interval as seen in Graph 5.1 block 1. Once the time is within limits the integrate routine starts. Integrate knows that the time is within limits because the creation of the tree like structure discussed earlier. The data from the current domain must be contained in the tree before any nesting in that level (ie. tree's lower levels) takes place. In other words: At time T+1, in node N, each child of node N should be at time T. This is achieved by doing calculations depth-first. Each node of the tree represents WRF data; they are pointers to a grid representation of the data itself (see Figure 5). The WRF user determines the NxM size of this grid. The grid is decomposed to grid cells. These cells are represented as horizontal coordinates in space, and each cell has the corresponding physical data in various multi-dimensional arrays and simple data members. All the data members for each grid cell are defined in the WRF Registry. See the Glossary for more information.

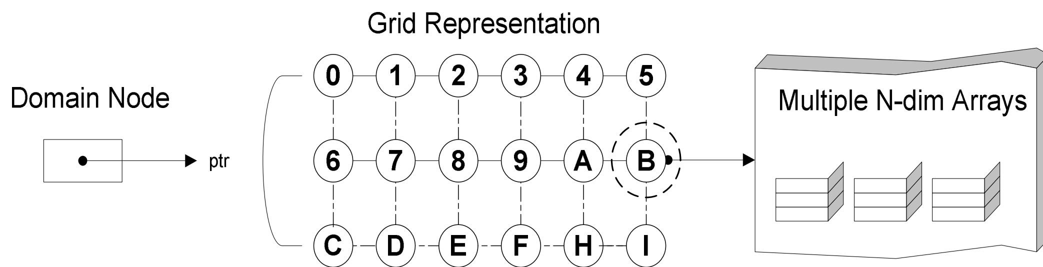


Figure 5 - Grid representation of data in the a domain

3.1.2 Integrate's Flow of Control

Integrate calls the subroutine model_to_grid_config_rec (Graph 5.1 block 2) located in module_configure.F (Graph 5.2 block 3) by passing the current domain ID and parameters. The output of the subroutine is of type grid_config_rec_type, consisting of all the domain characteristics (wind, atmospheric pressure, etc.) for this domain. Once the domain is initialized a subroutine to time the nesting on the simulation is called. This subroutine, start_timing (Graph 5.1 block 5), is located in module_timing.F (Graph 5.2 block 3). Before initializing the time step on the domain specified by the argument grid, the subroutine med_setup_step (Graph 5.1 block

6) located in mediation_layer.F (Graph 5.2 block 4) is called. This is a mediation layer routine that gives the model-layer a chance to do any pre-time-step initializations particular to the specified domain; this routine calls set_scalar_indices_from_config located in ../frame/module_configure.F. The set_scalar_indices_from_config adjusts the integer variables that are defined in module_state_description.F (Graph 5.2 block 3), which are registry generated and are used as indices into 4D tracer arrays for moisture, chemistry, etc. These indices have to be reset each time a different grid is computed. After these indices are set a check for open nests is made. This checks for nests which should start at the current time step and initializes and allocates them. The the new nests are initialized, allocated and timed by calling med_pre_nest_initial, alloc_and_configure_domain and Setup_Timekeeping respectively (see in Graph 5.1 blocks 8 and 9). At this point a check for overlapping domains occurs through set_overlaps located in module_nesting.F (Graph 5.2 block 3). This is a dummy call; WRF 2.X does not allow overlapping.

3.1.3 Integrate's Time Advance Loop In WRF

After a small debug call used for ESMF (Earth Science Modeling Framework) runs, we enter one of the two most important loops of the integrate subroutine (Graph 5.1 block 12). Integrate always enters this loop at least once because there is at least one active domain (grid_ptr is associated with the current grid right before the loop). The comments in the code indicate that this is a dummy loop in WRF 2.0 and executes it only once. There are a few important calls here. The first one is the call to solve_interface (Graph 5.1 block 13), which is located in solve_interface.F (Graph 5.2 block 4). As will be explained in the next section, solve_interface goes to the mediation and eventually to the model layer to do all the necessary physics calculations for the current grid. The call to domain_clockadvance, which is defined in module_domain.F (Graph 5.2 block 3), will advance the current time associated with the current grid by one timestep. Then, grid_ptr is associated with the next sibling of the current domain. If this was not executing as a dummy loop (and was executing more than once), it would solve and advance by one timestep any siblings that the current domain has. Once there were no siblings left to traverse grid_ptr would be null and control would drop out of the loop. Walk through an example, shown in Figure 6, will help provide a more concrete explanation.

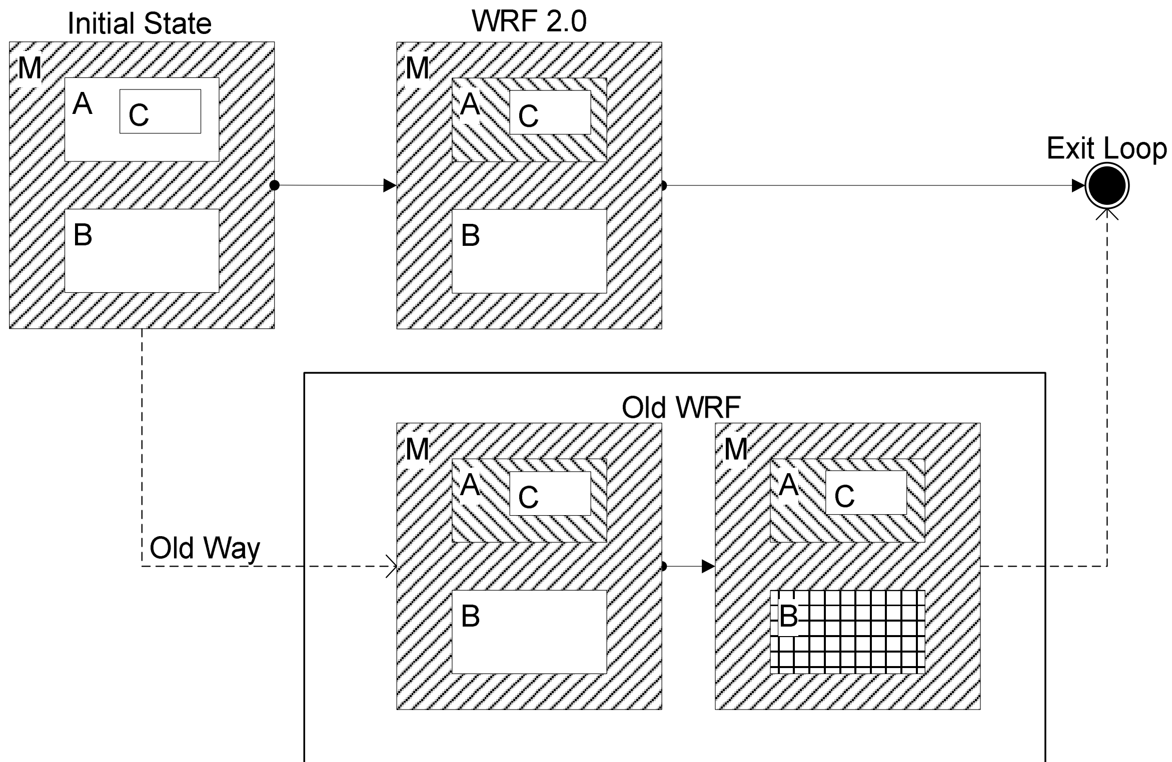


Figure 6 Time Advance Loop Example

As one can see, initially in the example, M the MOAD, is already solved and advanced by one M-TimeStep and the current_grid is A. The current time of M is T + TimeStep[M]. Then initially, A's start_subtime would be T, and A's stop_subtime would be T + TimeStep[M]. B's start_subtime and stop_subtime would also be the same. In WRF 2.0, we would solve A and advance it by one A-TimeStep (which is most likely to be different from M-TimeStep). After that we would exit the loop, and the current time in A would be T + TimeStep[A]. For previous versions of WRF, it seems the loop was actually functional. It would solve A and advance it by one A-TimeStep as explained above. However, then grid_ptr would be set to B, and we would reenter the loop. B would be solved and advanced by one B-TimeStep. Then the current time in B would be T + TimeStep[B]. At this point, grid_ptr would be set to the next sibling, which is null since there are none. The test at the beginning of the loop would fail, and we exit the loop.

3.1.3.1 Inside the solve interface subroutine

This subroutine is passed a domain object called grid and has a set of config_flags as a variable. This routine is still in the driver layer of WRF, but is at the lowest level. It is the last routine that deals with the domain before it is passed to the mediation and model layers. This routine dereferences the state fields from the Domain DDT and passes them as separate arguments to the solve routine.

At the beginning of this routine there are a few if statements which use flags to determine which solver to #include for use in the actual solving. WRF can use a number of different solvers; the current code references four of them. These are the NCAR Advanced Research WRF (or ARW) solver [18], the NCEP Nonhydrostatic Mesoscale Model (or NMM) solver [19], the US Navy's Coupled Ocean Atmosphere Mesoscale Prediction System (or COAMPS) [20], and an experimental solver core referred to as the EXP solver. In the code, the ARW solver is referred to as the EM (Eulerian Mass)(Glossary 6.2.1) solver. Although there is code that refers to all four of these solvers, only ARW and NMM (Glossary 6.2.2) are currently supported [21].

Following this there is a kludge to deal with a problem some compilers have in converting F90 arrays to F77 arrays. This conversion is done for performance reasons. Further details on this kludge can be found at [22]. The solve_interface then makes calls to model_to_grid_config_rec and set_scalar_indices_from_config. The role of these two routines is covered in section 3.1.2 of this document. Following these calls is a series of if statements which will select which solver is to be used. The solver may be chosen at runtime through the dyn_opt namelist option. Depending on what was chosen, the grid, config_flags, and "# include <em_actual_new_args.inc>" will be passed to the solve_xx routine, where xx is a short string that represents the solver to be used (for example, solve_em or solve_nmm). This routine resides in the /dyn_xx/ folder, where again xx is the same short string representing a solver (for example, the ARW solve_em routine is located in ../dyn_em/solve_em.F as seen in Graph 5.2 block 2). "# include <em_actual_new_args.inc>" is a list of actual arguments that are generated automatically by the Registry at the same time that the domain DDT fields are dereferenced from the domain DDT.

The solve_xx routine resides in the the mediation layer. It deals with the highest levels of the model layer and the lowest levels of the driver layer, and so has a mixture of attributes from both of the other layers. It contains things such as the flow of control through a single timestep on a domain, as well as calls to interprocessor communication [16]. Unlike the model layer, however, the solve_xx routine is not tile-callable and does no actual computation – rather, it contains the loops over tiles from which the true model layer subroutines are called [16]. And unlike the driver layer, the solve_xx routine has access to the actual individual state arrays and variables (passed in through their argument list) rather than a single derived data type domain object [16]. As a part of the mediation layer, solve_xx is dependent on both the model and driver layers. Any nontrivial change to either of the other two layers would necessitate a change in the implementation of solve_xx. However, this insulates changes in either the model or driver layer from each other. In other words, one can change the driver layer without worrying about the model layer. All that would need to change is the mediation layer. This also works the other way around. If it was needed to make changes to the model layer, only the mediation layer would be adjusted. This ultimately allows for a more modular development methodology.

3.1.4 Integrate's Recursive Loop For Nesting in WRF

For our model run of WRF we use 2-way nesting. Nesting is used to increase resolution over portions of a domain [10] and is accomplished by positioning a higher-resolution domain within a coarser domain and exchanging forcing and feedback data between the two [10] where the parent domain advances one time step; then data in the region of the nest is transferred from the parent to the nest. The model iterates over the smaller nested domain time steps, bringing it forward to the same time level as the parent. Finally, nested domain data is transferred back onto the region of the parent domain, and the nest time step commences [10]. Having reviewed nesting it is appropriate to continue with the flow of control for the integrate subroutine.

After the time advance loop two subroutines are called: med_calc_model_time and med_after_solve_io (Graph 5.1 block 14). These are stubs left over from a previous version of this program. After these the program goes into a series of nested DO loops. The outer loop iterates over the current siblings, while the inner loop iterates over the children (see in Graph 5.1 Blocks 15 and 16). Both loops are controlled using FORTRAN’s 'Associated' logical function

(see Glossary 1). Two main calls that deal with inter domain communications occur in this loop. The first call is to med_nest_force (Graph 5.1 block 17) inside module_integrate.F (Graph 5.2 block 4), and the second call is to med_nest_feedback (Graph 5.1 block 20) which is also inside mediation_integrate.F. In the first call, med_nest_force, the nest is initialized with interpolated data from the current parent domain. This subroutine calls another subroutine: med_force_domain, located in mediation_force_domain.F (Graph 5.2 block 4). This subroutine uses specific routines supplied by module_dm.F from the external communication package (Graph 5.2 block 2) to aid on the data interpolation; these routines are "interp_domain_X_part1" and "force_domain_X_part2" where X is the physics model that is currently used in the simulation (EM or NMM, see Glossary 2).

There are two parts in the inter-domain data force process, the first part consists of transferring the data to an intermediate domain, this intermediate domain is the same resolution of the parent domain but its size is that of the nest. The first part of the process involves communications between distributed memory processes. The model_to_grid_config_rec subroutine (Figure 7 block 1.B) inside med_force_domain (Figure 7 Block 1.A) passes the model configuration information that is needed for the specific nested domain and then makes a call to interp_domain_X_part1 (Figure 7 block 1.C). This subroutine will broadcast the data between the parent domain and the intermediate domain (see the top part the Figure 7). The second part of this process takes place without any distributed memory communication (i.e. all local). The system runs interpolation on the intermediate domain and computes the values for the nest boundaries. It uses the same model_to_grid_config_rec call to transfer the model's data but now uses the routine force_domain_X_part2. The data is located in HALO_EM_FORCE_DOWN.inc, a registry generated file that will be interpolated to the nested domain. This can be seen in the lower part of Figure 7.

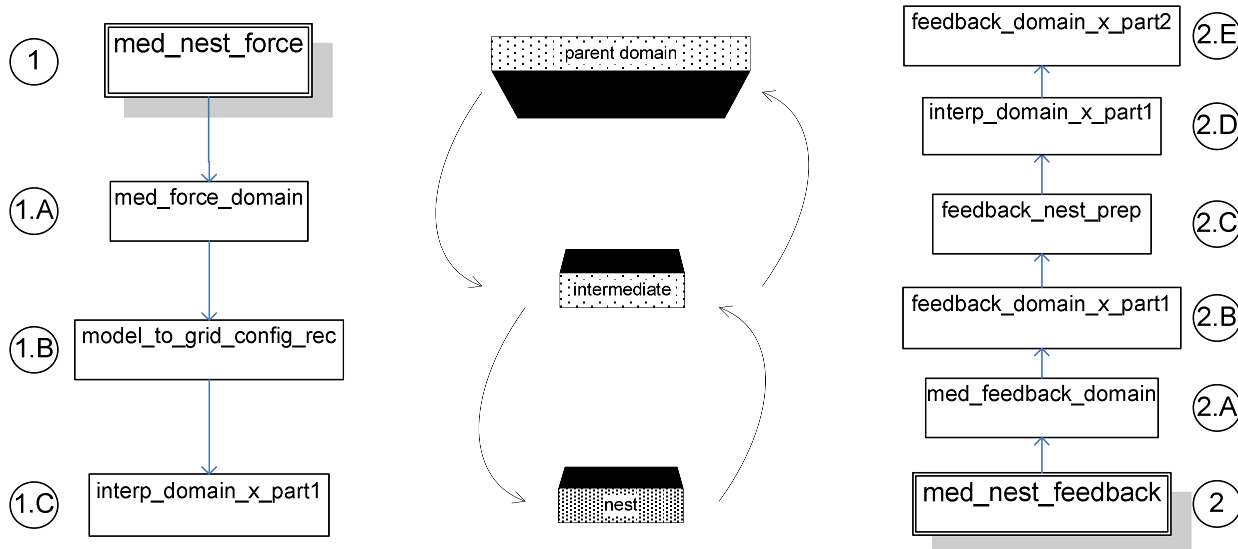


Figure 7 - Domain force and feedback calls.

The program then returns from the med_nest_force to setup the time period over which the nest is to run. Since the current grid has advanced one time step and the nest has not, the start time for the nest is the grid's current time minus one time step, the nests stop_subtime is the current time and this brings up the nest up to the same time level as the grid or parent domain. This step can be seen in Graph 5.1 Block 18.

Once the time is corrected the program goes into recursion of the subroutine again. It runs up to this point of the integrate routine for every nested domain (Graph 5.1 Block 19), including sibling domains. When the program execution reaches the greatest level of recursion (finest domain with no more nested domains), the recursion finishes. Then the program starts the callbacks until it reaches the MOAD. For a better understanding we will use a simple example. Assume the program has reached the point where the current domain does not have anymore siblings and does not have any children. The recursion will not continue and the integrate subroutine will finish for the current domain. First the program does some debugging calls to let the user know that it is coming back from the recursive call and that the program is about to do data feedback. Then the second most important call of this loop is made, the med_nest_feedback call. As with the forcing routines mentioned previously, this call eventually ends up using routines included in the external communication package. First the program calls med_feedback_domain from the routine med_nest_feedback (Graph 5.1 Block 20). Here the program will decide to use the routines feedback_domain_X_part1 and

feedback_domain_X_part2; depending on the model used replace X with XEM or NMM. Continuing with our previous example we are now in the last nest of the recursion and we need to feedback the data to the parent domain. We assume the use of the EM model. A call to feedback_domain_em_part1 (Figure 7 Block 2.B) is made. In this first part of the feedback the program uses a routine named feedback_nest_prep (Figure 7 Block 2.D) to prepare the data that is going to be sent to the parent domain. This routine invokes a halo (Glossary 6.4) exchange on the nested grid. This is done in a separate routine because the HALOs need data to be dereferenced from the grid data structure, and in this routine the dereferenced fields are related to the intermediate domain, not the nest itself. The program saves the current grid pointer to the intermediate domain, switches grid to point to the nested grid (ngrid), invokes feedback_nest_prep (Figure 7 Block 2.D) and then restores grid to point to the intermediate domain. Essentially the things that are in the current nested domain are copied to the intermediate domain. After we pass the nested domain data to the intermediate domain a call to the second part of this process is made, feedback_domain_em_part2 (Figure 7 Block 2.E). Here the data interpolation between the intermediate grid (which contains the data of the nested domain that was passed previously) and the parent domain occurs. If you would like to find more information about nesting in WRF please refer to John Michalakes publication about Nesting in WRF 2.0 [7].

4. WRF Finalization Stage

After the simulation is complete and the flow of control returns from wrf_run the subroutine wrf_finalize (Figure 8 Block 0) is called. It is in charge of cleanup and gracefully releasing resources now that the run is finished. Within wrf_finalize there is a call to med_shutdown_io (Figure 8 Block 1). This subroutine closes auxiliary data sets and then calls wrf_ioexit (Figure 8 Block 1.A). This performs two major functions. First external I/O packages are shutdown by calling their specific exit methods (e.g. ext_ncd_ioexit for NetCDF) (Figure 8 Block 1.B). Then, if quilting was enabled for the run clients will send their quilt servers a shutdown command via the wrf_quilt_io_exit subroutine (Figure 8 Block 1.C). The quilt servers have been in an infinite loop in the quilt subroutine for the entire run. This shutdown command, in the form of a message with a buffer size of -100, will cause them to close whatever I/O packages are in use and call mpi_finalize, which will cleanup MPI services. The quilt servers then exit with a Fortran STOP statement. A quilt server that receives a negative buffer size that is not equal to -100 treats it as a possible overflow and exits ungracefully with a fatal error.

Upon the return to wrf_finalize a call to wrf_shutdown is made (Figure 8 Block 2). If this was a distributed memory run (DM_PARALLEL is defined) then wrf_dm_shutdown (Figure 8 Block 2.A), a communication package specific subroutine, is called to finalize the distributed memory framework. Otherwise a STOP statement simply terminates program execution. For RSL Lite the wrf_dm_shutdown subroutine makes a call to MPI_FINALIZE, which cleans up all MPI related resources. The flow of control returns to the wrf program, where nothing more is done. At this point program execution ends.

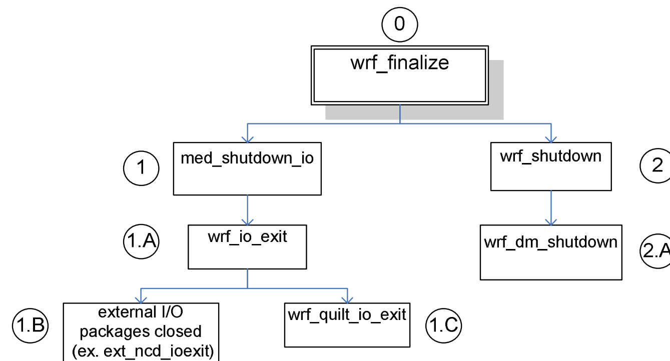
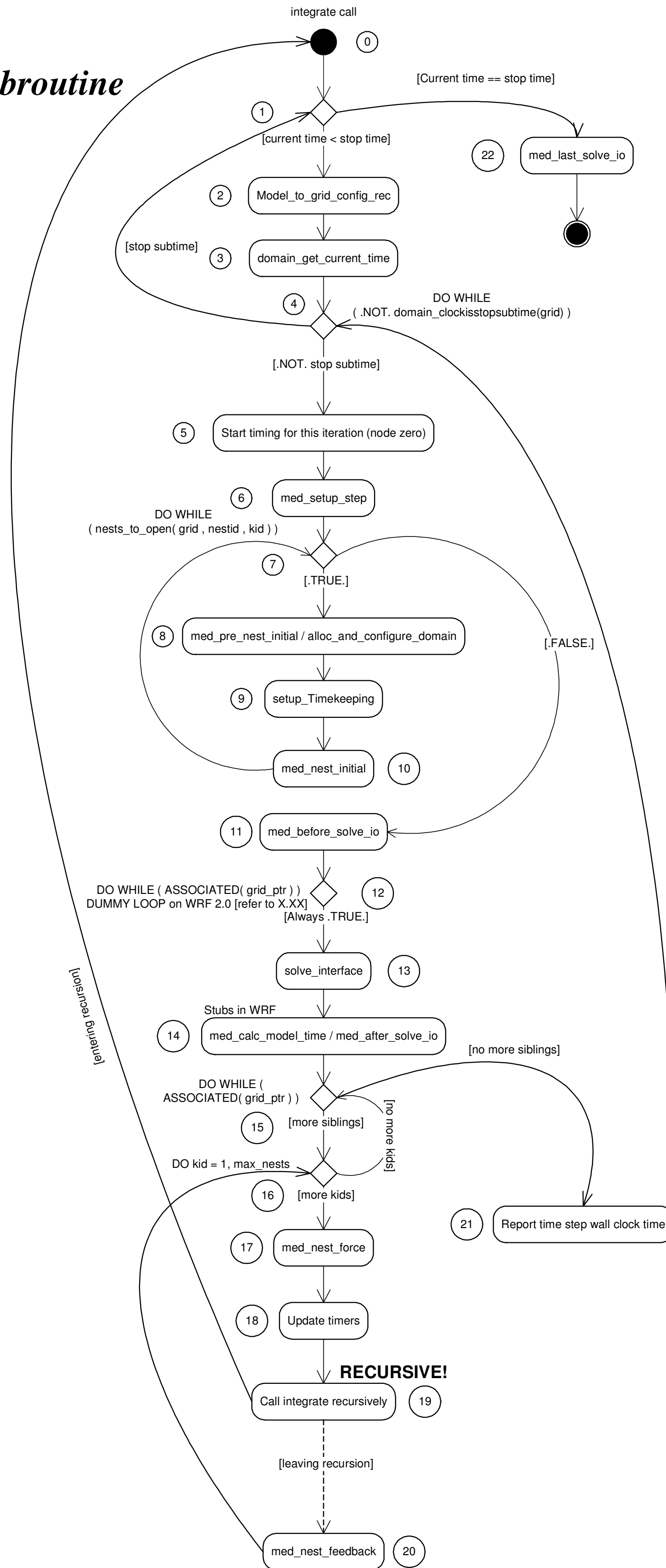


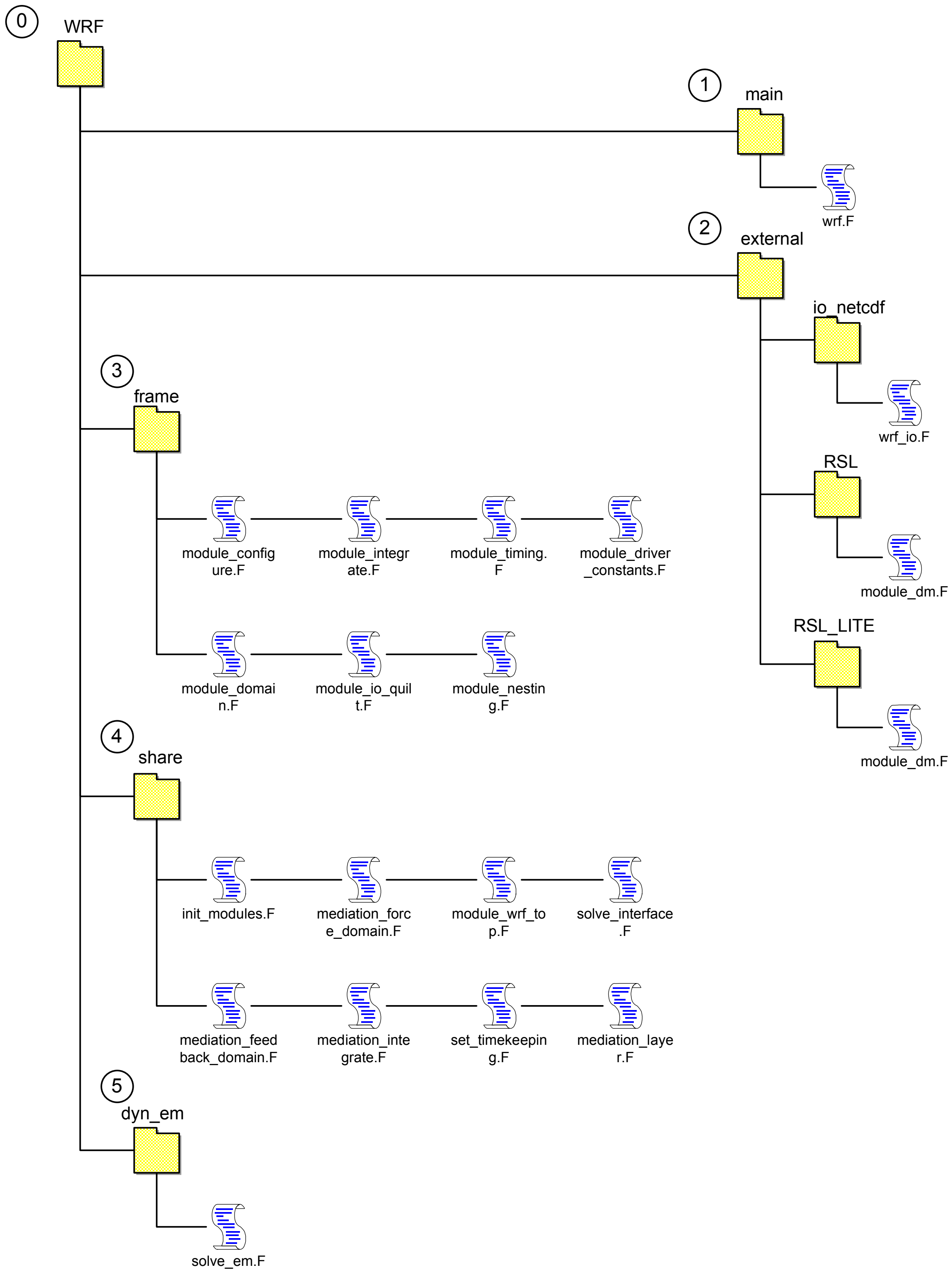
Figure 8 - wrf_finalize subroutine calls

5. Graphs

5.1 Integrate Subroutine



5.2 Partial WRF Directory Graph



6. Glossary

6.1 ASSOCIATED

In Fortran 90, ASSOCIATED (PTR [, TGT]) determines the status of the pointer PTR or if PTR is associated with the target TGT. PTR shall have the POINTER attribute and it can be of any type and (Optional) TGT shall be a POINTER or a TARGET. It must have the same type, kind type parameter, and array rank as PTR. Refer to the following website: http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gfortran/ASSOCIATED.html, for more information.

6.2.1 EM

An Eulerian solver based on a flux formulation of the fully compressible nonhydrostatic equations with a mass (hydrostatic pressure) vertical coordinate has been constructed and is being tested within the WRF coding framework. Prognostic variables for this solver are the mass in the column (hydrostatic surface pressure), and coupled with the column mass - potential temperature, horizontal velocities u and v the vertical velocity w, and the geopotential. [23]

6.2.2 NMM

The Nonhydrostatic Mesoscale Model (NMM) core of the Weather Research and Forecasting (WRF) system was developed by the National Oceanic and Atmospheric Adminstration (NOAA) National Centers for Environmental Prediction (NCEP). The current release is Version 2.2. The WRF-NMM is designed to be a flexible, state-of-theart atmospheric simulation system that is portable and efficient on available parallel computing platforms. The WRF-NMM is suitable for use in a broad range of applications across scales ranging from meters to thousands of kilometers, including: Real-time NWP, Forecast research, Parameterization research, Coupled-model applications and Teaching. Refer to the introduction in the following document: www.dtcenter.org/wrf-mm/users/docs/user_guide/SI/complete_users_guide_nmm_SI.pdf for more information.

6.3 WRF REGISTRY

The WRF 2.0 code-base is approximately 160,000 lines long of which 40,000 are automatically generated in compile time by the Registry. The Registry is a type of Computer Aided Software Engineering (CASE) tool. A tool like this one is helpful when one has to have architecture dependent code and large sections of declaration and initialization, as in the case of grid points in WRF. These are the most error-prone parts of the code and can be easily written via automation. The Registry is divided in two parts: the database with all the required data and the code generator.

As of right now the Registry 'database' is a simple text file. Each line in the text file is a tuple. The notion of ‘tables’ in this text file is identified by the first entry in each tuple. There are twelve tables [24]:

Dimspec – Describes dimensions that are used to define arrays in the model
State – Describes state variables and arrays in the domain DDT
I1 – Describes local variables and arrays in solve
Typedef – Describes derived types that are subtypes of the domain DDT
Rconfig – Describes a configuration (e.g. namelist) variable or array
Package – Describes attributes of a package (e.g. physics)
Halo – Describes halo update interprocessor communications
Period – Describes communications for periodic boundary updates
Xpose – Describes communication for transposition of a variable between decompositions
Initialization – Describes communications and data for nest initialization from the coarse domain
Force – Describes communications and data for forcing of nest boundary arrays
Feedback – Describes communications and data for nest feedback onto the coarse domain

Of special importance are the State, Halo and Xpose tables. The State table defines every data member that each grid cell has in the domain TYPE data structure. This includes physic information as well as temporal data needed for nesting. Halo entries define properties for the information exchange between adjacent grid points. The Xpose table describes the

communication between nested domains. A much more detailed description of the Registry is available in [24].

6.4 HALO

Halo region is part of local memory but is not considered to be part of the patch and needs only exist if the domain is decomposed over multiple patches. The patch size plus the halo region constitute the minimum memory size for the local processor. [25]

7. References

[1] The Weather Research & Forecasting Model, http://www.wrf-model.org/index.php

[2] High Performance Computing - High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions , by Jack Dongarra, Thomas Sterling, Horst Simon, Erich Strohmaier

[3] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/

[4] J. Michalakes, M McAtee, J. Wegiel, “Software Infrastructure for the Weather Research and Forecast Model”

[5] WRF Standard Initialization, http://wrfsi.noaa.gov/

[6] Robert Rozumalski’s WRF version. For more information contact: robert.rozumalski@noaa.gov.

[7] Nesting in WRF 2.0: http://box.mmm.ucar.edu/mm5/workshop/ws04/Session6/Gill.Dave.pdf

[8] Documentation and Software for WMO GRIB. http://www.nco.ncep.noaa.gov/pmb/docs/on388/

[9] NetCDF (network Common Data Form). http://www.unidata.ucar.edu/software/netcdf/

[10] Michalakes, John "RSL: A Parallel Runtime System Library for Regional Atmospheric Models with Nesting."

[11] ESMF

[12] Michalakes, John, "A Runtime System Library for Parallel Finite Difference Models with Nesting."

[13] “Weather Research & Forecast (WRF) Model Development": http://www.mmm.ucar.edu/ppws/ppws_weather.html

[14] WRF Namelist File, http://www.mmm.ucar.edu/wrf/users/wrfv2/wrf-namelist.html

[15] National Center for Atmospheric Research, http://www.ncar.ucar.edu/

[16] "Weather Research and Forecast Model 1.2: Software Design and Implementation. Draft" NCAR Tiger Team.

[17] John G. Michalakes, http://www-unix.mcs.anl.gov/~michalak/CV.htm

[18] WRF ARW online tutorial,
http://www.mmm.ucar.edu/wrf/OnLineTutorial/index.htm
[19] WRF NMM Users Website, http://www.dtcenter.org/wrf-nmm/users/
[20] Coupled Ocean/Atmospheric Mesoscale Prediction System:
http://www.nrlmry.navy.mil/coamps-web/web/home
[21] WRF Users and Websites, http://wrf-model.org/users/users.php
[22] Deref_Kludge, http://www.mmm.ucar.edu/wrf/WG2/topics/deref_kludge.htm
[23] WRF Working Group 1: Numerical and Model Dynamics:
http://www.mmm.ucar.edu/wrf/WG1/
[24] J. Michalakes, D. Schaffer, "WRF Tiger Team Documentation: The Registry"
[25] WRF coding conventions –draft:
http://www.mmm.ucar.edu/wrf/WG2/WRF_conventions.html