Tutorial Notes: WRF Software 2.0

John Michalakes

WRF Working Group 2

June 29, 2004

Outline

- Introduction
- Computing Overview
- Software Overview
- Data Structures
- Registry
- I/O & Nesting
- Others: Time management, Error Handling
- Example

Introduction

- Intended audience for this tutorial session:
  - Primarily scientific users and others who wish to:
    - Work with the code
    - Extend/modify the code to enable their work/research
    - Address problems as they arise
    - Adapt the code to take advantage of local computing resources
  - Also: developers, computer scientists and software engineers, computer vendors
    - Developing new functionality (e.g. moving nests, coupling)
    - Integration with frameworks and other community infrastructure
    - Porting and benchmarking new platforms

Introduction

- Characteristics of WRF Software
  - Developed from scratch beginning around 1998
  - Requirements emphasize flexibility over a range of platforms, applications, users; performance
  - WRF develops rapidly. First released Dec 2000; Last beta release, 1.3, in May 2003. Official 2.0 release May, 2004
  - Current source code
    - Fortran 90:125,000 lines (+ 40,000 auto-generated)
      960 subroutines (+ 800 auto-generated)
    - C: 6,000 lines (includes auto-generator code)
    - Misc: 1,800 lines (shell, Perl, Makefiles, etc.)

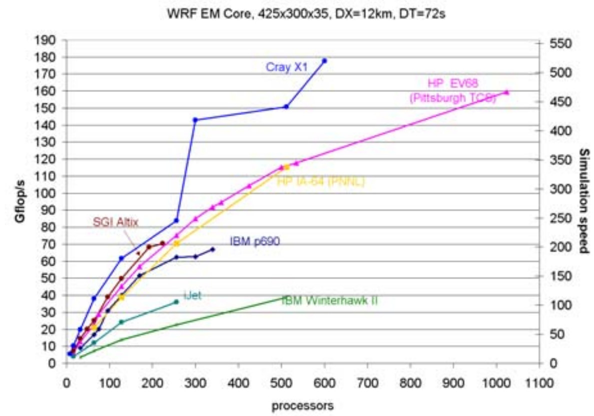
Introduction

- Supported Platforms (alphabetical)

Vendor	Hardware	O.S.	Compiler
Cray Inc.	X1	UNICOS	vendor
HP/Compaq	Alpha	Tru64	vendor
	IA-64 (Intel)	Linux	Intel
		HPUX	vendor
IBM	SP Power-*x*	AIX	vendor
SGI	IA-64 (Intel)	Linux	Intel
	MIPS	Irix	vendor
Sun	UltraSPARC	Solaris	vendor
various	IA-32/AMD 32	Linux	Intel/PGI
	IA-64/Opteron	Linux	Intel/PGI

Performance

http://www.mmm.ucar.edu/wrf/bench

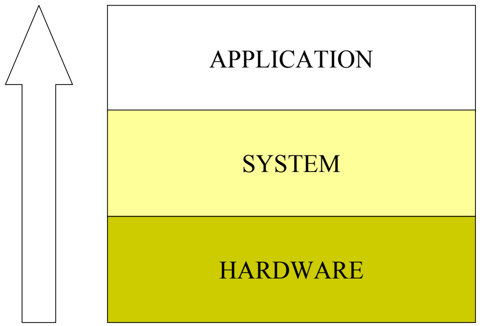


Some terms

- WRF Architecture – scheme of software layers and interface definitions
- WRF Framework – the software infrastructure, also "driver layer" in the WRF architecture
- WRF Model Layer – the computational routines that are specifically WRF
- WRF Model – a realization of the WRF architecture comprising the WRF model layer with some framework
- WRF – a set of WRF architecture-compliant applications, of which the WRF Model is one

Computing Overview

Computing Overview



Patches
Tiles
WRF Comms

Processes
Threads
Messages

Processors
Nodes
Networks

Hardware: The Computer

- The 'N' in NWP
- Components
  - Processor
    - A program counter
    - Arithmetic unit(s)
    - Some scratch space (registers)
    - Circuitry to store/retrieve from memory device
  - Memory
  - Secondary storage
  - Peripherals
- The implementation has been continually refined, but the basic idea hasn't changed much

Hardware has not changed much…

A computer in 1960

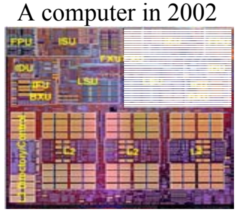
IBM 7090

6-way superscalar
36-bit floating point precision
~144 Kbytes

~50,000 flop/s
48hr 12km WRF Conus in 600 years

A computer in 2002

IBM p690



4-way superscalar
64-bit floating point precision
1.4 Mbytes (shown)
> 500 Mbytes (not shown)

~5,000,000,000 flop/s
48 12km WRF CONUS in 52 Hours

…how we use it has

- Fundamentally, processors haven't changed much since 1960
- Quantitatively, they haven't improved nearly enough
  - 100,000x increase in peak speed
  - > 4,000x increase in memory size
  - These are too slow and too small for even a moderately large NWP run today
- We make up the difference with <u>parallelism</u>
  - Ganging multiple processors together to achieve $10^{11\text{-}12}$ flop/second
  - Aggregate available memories of $10^{11\text{-}12}$ bytes

~100,000,000,000 flop/s
48 12km WRF CONUS in under 15 minutes

Parallel computing terms -- hardware

- **Processor**:
  - A device that reads and executes instructions in sequence to produce perform operations on data that it gets from a memory device producing results that are stored back onto the memory device
- **Node**: One memory device connected to one or more processors.
  - Multiple processors in a node are said to "share-memory" and this is shared memory parallelism
  - They can work together because they can see each other's work
  - The latency and bandwidth to memory affect performance
- **Cluster**: One or more nodes connected by a network
  - The processors attached to the memory in one node can not see the memory for processors on another node
  - For processors on different nodes to work together they must send messages between the nodes. This is "distributed memory parallelism"
- **Network:**
  - Devices and wires for sending messages between nodes
  - Bandwidth – a measure of the number of bytes that can be moved in a second
  - Latency – the amount of time it takes before the first byte of a message arrives at its destination

Parallel Computing Terms -- Software

"The only thing one does directly with hardware is pay for it."

- Process:
  - A set of instructions to be executed on a processor
  - Enough state information to allow process execution to stop on a processor and be picked up again later, possibly by another processor
- Processes may be lightweight or heavyweight
  - Lightweight processes, e.g. shared-memory threads, store very little state; just enough to stop and then start the process
  - Heavyweight processes, e.g. UNIX processes, store a lot more (basically the memory image of the job)

Parallel Computing Terms -- Software

- Every job has at least one heavy-weight *process*.
  - A job with more than one process is a distributed-memory parallel job
  - Even on the same node, heavyweight processes do not share memory[†]
- Within a heavyweight process you may have some number of lightweight processes, called *threads.*
  - Threads are shared-memory parallel; only threads in the same memory space can work together.
  - A thread never exists by itself; it is always inside a heavy-weight process.
- Processes (heavy-weight) are the vehicles for distributed memory parallelism
- Threads are the vehicles for shared-memory parallelism

Jobs, Processes, and Hardware

- MPI is used to start up and pass messages between multiple heavyweight processes
  - The **mpirun** command controls the number of processes and how they are mapped onto nodes of the parallel machine
  - Calls to MPI routines sending and receiving messages and control other interactions between processes
  - http://www.mcs.anl.gov/mpi
- OpenMP is used to start up and control threads within each process
  - Directives specify which parts of the program are multi-threaded
  - **OpenMP** environment variables determine the number of threads in each process
  - http://www.openmp.org
- The number of processes (number of MPI processes times the number of threads in each process) usually corresponds to the number of processors

Examples

- If the machine consists of 4 nodes, each with 4 processors, how many different ways can you run a job to use all 16?
  - 4 MPI processes, each with 4 threads

```
setenv OMP_NUM_THREADS 4
mpirun -np 4 wrf.exe
```

  - 8 MPI processes, each with 2 threads

```
setenv OMP_NUM_THREADS 2
mpirun -np 8 wrf.exe
```

  - 16 MPI processes, each with 1 thread

```
setenv OMP_NUM_THREADS 1
mpirun -np 16 wrf.exe
```

Examples (cont.)

- Note, since there are 4 nodes, we can never have fewer than 4 MPI processes because nodes do not share memory
- What happens on this same machine for the following?

```
setenv OMP_NUM_THREADS 4
mpirun -np 32
```

Other information about Parallel Processes

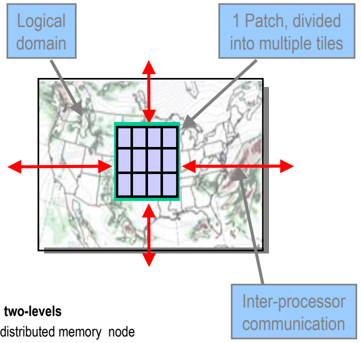
- Memory limits for heavy-weight processes
  - A process doesn't get all the memory and running out is ugly
    - Often appears as a segmentation violation in some otherwise correct-looking part of the program
    - Soft limits on per-process memory controlled by the **limit** and **unlimit** commands
    - Hard limits are set in the operating system; need administrator to change
  - Virtual memory
    - Even when you're not running out of memory, you may be running out of physical memory
    - Program will still run but it will be many times slower
  - Make sure that **mpirun** is distributing processes evenly over the nodes in your partition. You may need to use the **–machinefile** or other options
  - Some versions of MPI have buffer size limits
- Memory limits for light-weight processes
  - Thread-private stack size is usually limited and running out is uglier
  - May be enlarged; for example, the MPSTKZ environment variable with the Portland Group compilers

Application: WRF

- WRF uses *domain decomposition* to divide total amount of work over parallel processes
- Since the process model has two levels, the decomposition has two levels:
  - The domain is first broken up into rectangular pieces that are assigned to heavy-weight processes. These pieces are called *patches*
  - The *patches* may be further subdivided into smaller rectangular pieces that are called *tiles,* and these are assigned to *threads* within the process.

Parallelism in WRF: Multi-level Decomposition

- **Single version of code for efficient execution on:**
  - Distributed-memory
  - Shared-memory
  - Clusters of SMPs
  - Vector and microprocessors



Model domains are decomposed for parallelism on two-levels

Patch: section of model domain allocated to a distributed memory node

Tile: section of a patch allocated to a shared-memory processor within a node; this is also the scope of a model layer subroutine.

Distributed memory parallelism is over patches; shared memory parallelism is over tiles within patches

Distributed Memory Communications

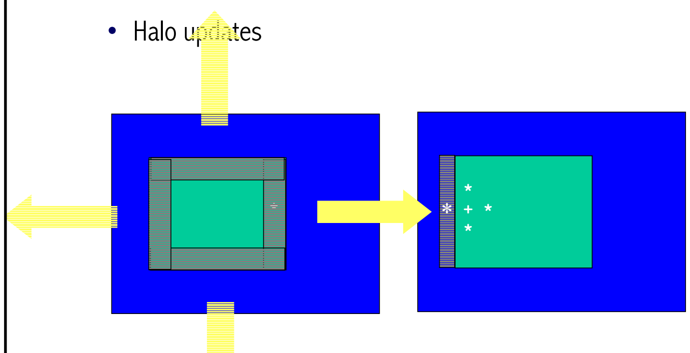
Example code fragment that requires communication between patches

Note the tell-tale +1 and –1 expressions in indices for **rr** and **H1** arrays on right-hand side of assignment. These are *horizontal data dependencies* because the indexed operands may lie in the patch of a neighboring processor. That neighbor's updates to that element of the array won't be seen on this processor. We have to communicate.

```
                       (dyn_eh/module_diffusion.F )

SUBROUTINE horizontal_diffusion_s (tendency, rr, var, . . .
. . .
   DO j = jts,jte
   DO k = kts,ktf
   DO i = its,ite
      mrdx=msft(i,j)*rdx
      mrdy=msft(i,j)*rdy
      tendency(i,k,j)=tendency(i,k,j)-                                  &
           (mrdx*0.5*((rr(i+1,k,j)+rr(i,k,j))*H1(i+1,k,j)-            &
                       (rr(i-1,k,j)+rr(i,k,j))*H1(i   ,k,j))+          &
            mrdy*0.5*((rr(i,k,j+1)+rr(i,k,j))*H2(i,k,j+1)-            &
                       (rr(i,k,j-1)+rr(i,k,j))*H2(i,k,j  ))-           &
            msft(i,j)*(H1avg(i,k+1,j)-H1avg(i,k,j)+                    &
                       H2avg(i,k+1,j)-H2avg(i,k,j)                     &
                                 )/dzetaw(k)                            &
           )
   ENDDO
   ENDDO
   ENDDO
 . . .
```

Distributed Memory Communications

- Halo updates

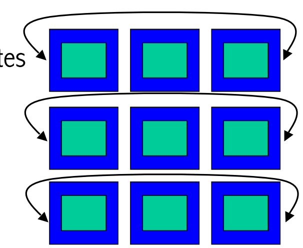


memory on one processor

memory on neighboring processor

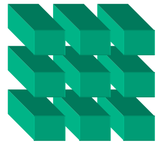
Distributed Memory Communications

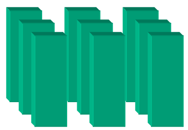
- Halo updates
- Periodic boundary updates



Distributed Memory Communications

- Halo updates
- Periodic boundary updates
- Parallel transposes





Review

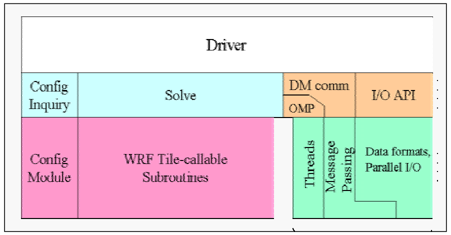
Layer			Distributed Memory Parallel		Shared Memory Parallel
APPLICATION (WRF)	Domain	*contains*	Patches	*contain*	Tiles
SYSTEM (UNIX, MPI, OpenMP)	Job	*contains*	Processes	*contain*	Threads
HARDWARE (Processors, Memories, Wires)	Cluster	*contains*	Nodes	*contain*	Processors

WRF Software Overview

WRF Software

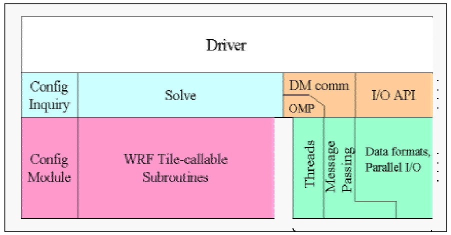
- Architecture
- Directory structure
- Module Conventions and USE Association
- Model Layer Interface

WRF Software Architecture



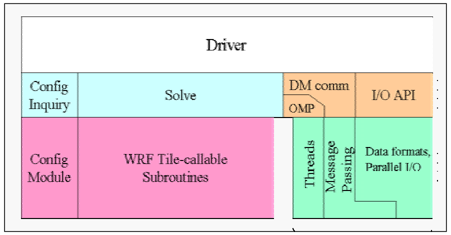
- Hierarchical software architecture
  - Insulate scientists' code from parallelism and other architecture/implementation-specific details
  - Well-defined interfaces between layers, and external packages for communications, I/O, and model coupling facilitates code reuse and exploiting of community infrastructure, e.g. ESMF.

WRF Software Architecture



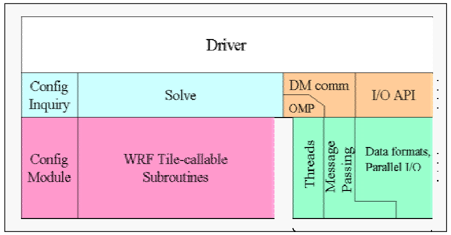
- Driver Layer
  - Allocates, stores, decomposes model domains, represented abstractly as single data objects
  - Contains top-level time loop and algorithms for integration over nest hierarchy
  - Contains the calls to I/O, nest forcing and feedback routines supplied by the Mediation Layer
  - Provides top-level, non package-specific access to communications, I/O, etc.
  - Provides some utilities, for example **module_wrf_error**, which is used for diagnostic prints and error stops

WRF Software Architecture



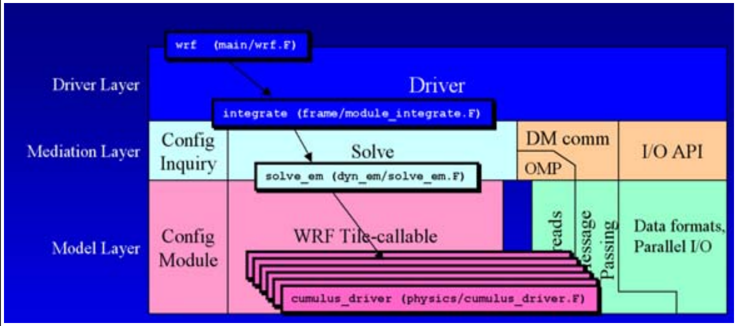
- Mediation Layer
  - Provides to the Driver layer
    - Solve solve routine, which takes a domain object and advances it one time step
    - I/O routines that Driver when it is time to do some input or output operation on a domain
    - Nest forcing and feedback routines
    - The Mediation Layer and not the Driver knows the specifics of what needs to be done
  - The sequence of calls to Model Layer routines for doing a time-step is known in Solve routine
  - Responsible for dereferencing driver layer data objects so that individual fields can be passed to Model layer Subroutines
  - Calls to message-passing are contained here as part of solve routine

WRF Software Architecture

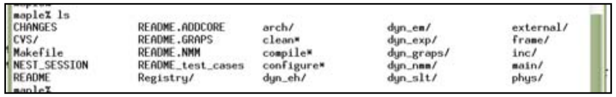


- Model Layer
  - Contains the information about the model itself, with machine architecture and implementation aspects abstracted out and moved into layers above
  - Contains the actual WRF model routines that are written to perform some computation over an arbitrarily sized/shaped subdomain
  - All state data objects are simple types, passed in through argument list
  - Model Layer routines don't know anything about communication or I/O; and they are designed to be executed safely on **one thread** – they never contain a **PRINT**, **WRITE**, or **STOP** statement
  - These are written to conform to the Model Layer Subroutine Interface (more later) which makes them "tile-callable"

Directory Structure



WRF Model Directory Structure



```
maple% ls
CHANGES        README.ADDCORE       arch/        dyn_em/      external/
CVS/           README.GRAPS         clean*       dyn_exp/     frame/
Makefile       README.NMM           compile*     dyn_graps/   inc/
NEST_SESSION   README_test_cases    configure*   dyn_nmm/     main/
README         Registry/            dyn_eh/      dyn_slt/     phys/
maple%
```

2.1. DIRECTORY STRUCTURE

The top-level WRFMODEL directory contains the following:

main -- directory containing Makefile and files containing main programs for the WRF model and initialization programs;

frame -- directory containing Makefile and source files specific to the WRF software framework;

dyn_xx -- directory containing Makefile and source files specific to a particular dynamical core xx;

phys -- directory containing Makefile and source files for physics;

share -- directory containing Makefile and source files for non-physics modules shared between dynamical cores;

external -- directory containing Makefile and subdirectories containing external packages for I/O, communications, etc.;

Registry -- directory containing the registry database;

clean, configure, and compile -- shell scripts (csh) for cleaning, configuring, and compiling the model;

arch -- directory containing settings files and scripts for configuring the model on different platforms; the file containing the settings for all currently supported platforms is configure.defaults;

inc -- directory that holds registry-generated include files (essentially empty on initial distribution);

tools -- directory containing tools used to build the model; the Makefile and source files for the registry mechanism reside here;

run and test -- run directories for the model; run is the default run directory; test contains standardized idealized and real-data test cases for the model; and

Makefile -- the top level (UNIX) make file for building WRF. This is not used directly; WRF is configured and built using the scripts mentioned above.

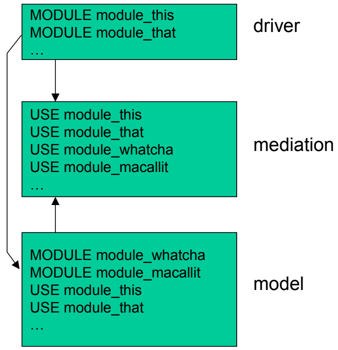
page 5,
WRF D&I Document

WRF File Taxonomy and Nomenclature

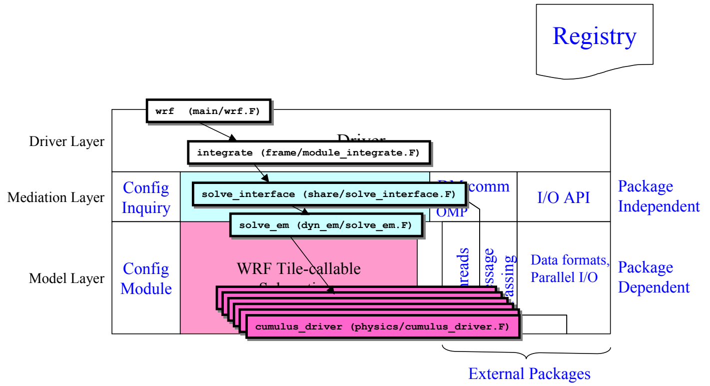
Module files				
	14	Driver	frame/module_*.F	WRF framework (driver layer)
	1	Driver	frame/module_state_description.F	registry generated framework file
	15	Model	dyn_em/module_*.F	em core-specific model layer
	17	Model	dyn_nmm/module_*.F	nmm core-specific model layer
	11	Model	share/module_*. F	non-core specific model layer
	26	Model	phys/module_*pp*_*. F	physics modules, where *pp is kind of physics*
	2	Model	phys/module_*.F	misc physics routines
	5	Mediation	phys/module_*_driver.F	physics drivers
Non-module Fortran Source				
	5	Driver	main/*.F	main programs (1 wrf and 6 preprocs)
	3	Driver	frame/*.c	C-language routines in the WRF framework
	5	Mediation	dyn_em/*. F	em core-specific routines (includes solver)
	12	Mediation	dyn_nmm/*. F	nmm core-specific routines (includes solver)
	5	Mediation	share/mediation_*. F	mediation layer
	38	Mediation	share/*something*. F	mediation layer and miscellaneous
Include files				
			inc/*. inc	registry generated includes
			inc/*. h	io api definitions, autogenerated from build
Others				
	13		Makefile */Makefile	build mechanism
	3		configure, compile, clean scripts	build mechanism
	19		tools/*.c	source for registry program
	1		tools/regtest.csh	a regression tester for WRF model
Externals				
	7	External	external/*	external package directories

Module Conventions and USE Association

- Modules are named module_*something*
- Name of file containing module is module_*something*.F
- If a module includes an initialization routine, that routine should be named init_module_*something()*
- Typically:
  - Driver and model layers are made up of modules,
  - Mediation layer is bare subroutines, except for physics drivers in phys directory
  - Gives benefit of modules while avoiding cycles in the use association graph



Code Structure



WRF Model Layer Interface

- Interface Mediation layer <> Model Layer
  - All state arrays passed through argument list as simple (not derived) data types
  - Domain, memory, and run dimensions passed unambiguously in three physical dimensions
  - Restrictions on model layer subroutines
    - No I/O, communication, no stops or aborts (use wrf_error_fatal in frame/module_wrf_error.F)
    - No common/module storage of decomposed data (exception allowed for set-once/read-only tables)
    - Spatial scope of a Model Layer call is one "tile"
    - Temporal scope of a call is limited by coherency
    - Computation on halos is allowed and considered a model-layer concern

```
SUBROUTINE solve_xxx (
    . . .
!$OMP DO PARALLEL
   DO ij = 1, numtiles
      its = i_start(ij) ; ite = i_end(ij)
      jts = j_start(ij) ; jte = j_end(ij)
      CALL model_subroutine( arg1, arg2, . . .
           ids , ide , jds , jde , kds , kde ,
           ims , ime , jms , jme , kms , kme ,
           its , ite , jts , jte , kts , kte )
   END DO
    . . .

 END SUBROUTINE
```

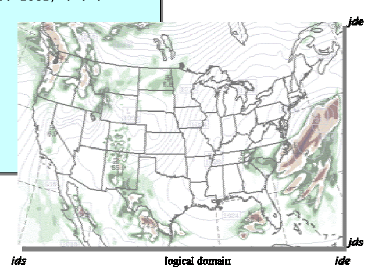
template for model layer subroutine

```
SUBROUTINE model_subroutine ( &
  arg1, arg2, arg3, … , argn,   &
  ids, ide, jds, jde, kds, kde, &  ! Domain dims
  ims, ime, jms, jme, kms, kme, &  ! Memory dims
  its, ite, jts, jte, kts, kte  )  ! Tile dims

IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)         :: arg7, . . .
 . . .
! Define Local Data (I2)
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
 . . .
! Executable code; loops run over tile
! dimensions
DO j = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

template for model layer subroutine

```
SUBROUTINE model ( &
  arg1, arg2, arg3, … , argn,   &
  ids, ide, jds, jde, kds, kde, &  ! Domain dims
  ims, ime, jms, jme, kms, kme, &  ! Memory dims
  its, ite, jts, jte, kts, kte  )  ! Tile dims

IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)         :: arg7, . . .
 . . .
! Define Local Data (I2)
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
 . . .
! Executable code; loops run over tile
! dimensions
DO j = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

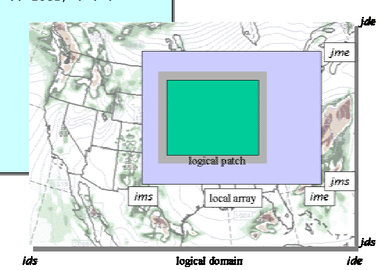
- Domain dimensions
  - Size of logical domain
  - Used for bdy tests, etc.



template for model layer subroutine

```
SUBROUTINE model ( &
  arg1, arg2, arg3, … , argn,   &
  ids, ide, jds, jde, kds, kde, &  ! Domain dims
  ims, ime, jms, jme, kms, kme, &  ! Memory dims
  its, ite, jts, jte, kts, kte  )  ! Tile dims

IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)         :: arg7, . . .
 . . .
! Define Local Data (I2)
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
 . . .
! Executable code; loops run over tile
! dimensions
DO j = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

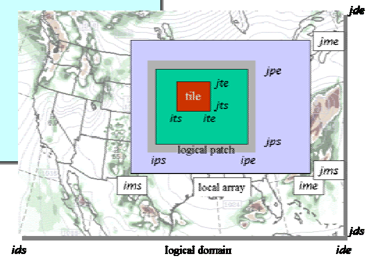
- Domain dimensions
  - Size of logical domain
  - Used for bdy tests, etc.
- Memory dimensions
  - Used to dimension dummy arguments
  - Do not use for local arrays



template for model layer subroutine

```
SUBROUTINE model ( &
  arg1, arg2, arg3, … , argn,   &
  ids, ide, jds, jde, kds, kde, &  ! Domain dims
  ims, ime, jms, jme, kms, kme, &  ! Memory dims
  its, ite, jts, jte, kts, kte  )  ! Tile dims

IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)         :: arg7, . . .
 . . .
! Define Local Data (I2)
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
 . . .
! Executable code; loops run over tile
! dimensions
DO j = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

- Domain dimensions
  - Size of logical domain
  - Used for bdy tests, etc.
- Memory dimensions
  - Used to dimension dummy arguments
  - Do not use for local arrays
- Tile dimensions
  - Local loop ranges
  - Local array dimensions



Data Structures

Data Structures

- Data Taxonomy
- How data appears at different levels of architecture
- Grid representation in WRF arrays
- Lateral Boundary Condition arrays
- Four dimensional tracer arrays

Data Structures

- WRF Data Taxonomy
  - State data
  - Intermediate data type 1 (I1)
  - Intermediate data type 2 (I2)
  - Heap storage (COMMON or Module data)

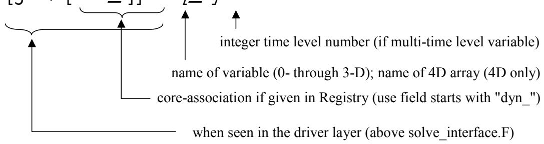
State Data

- Persist for the duration of a domain
- Represented as fields in domain data structure
- Arrays are represented as dynamically allocated pointer arrays in the domain data structure
- Declared in Registry using **state** keyword
- Always **memory** dimensioned; always **thread shared**
- Only state arrays can be subject to I/O and Interprocessor communication

WRF State Variables

- May be 0d, 1d, 2d, 3d, or 4d
- What they look like in the code:

[grid%[*core_*]] *var [_tl]*



Example

The second time level of the u variable in the Eulerian Mass (EM) core can be accessed in the driver layer as:

```
grid%em_u_2
```

in the solve_em routine and below it is simply:

```
u_2
```

I1 Data

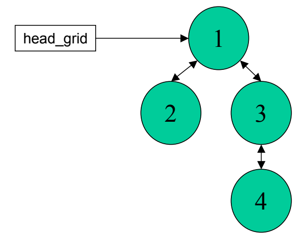
- Data that persists for the duration of 1 time step on a domain and then released
- Declared in Registry using **i1** keyword
- Typically automatic storage (program stack) in solve routine
- Typical usage is for tendency arrays in solver
- Always **memory** dimensioned and **thread shared**
- Typically **not** communicated or I/O

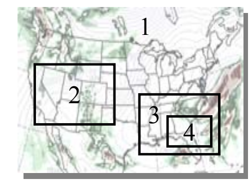
I2 Data

- I2 data are local arrays that exist only in model-layer subroutines and exist only for the duration of the call to the subroutine
- I2 data is not declared in Registry, never communicated and never input or output
- I2 data is **tile** dimensioned and **thread local**; over-dimensioning within the routine for redundant computation is allowed
  - the responsibility of the model layer programmer
  - should always be limited to thread-local data

Data Structures

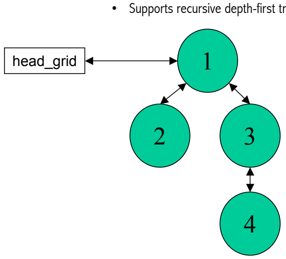
- What you see depends on where you are
  - Driver layer
    - All data for a domain is a single object, a domain derived data type (DDT)
    - The domain DDTs are dynamically allocated/deallocated
    - Linked together in a tree to represent nest hierarchy; root pointer is **head_grid**, defined in frame/module_domain.F
    - Supports recursive depth-first traversal algorithm (frame/module_integrate.F)





Data Structures

- What you see depends on where you are
  - Driver layer
    - All data for a domain is a single object, a domain derived data type (DDT)
    - The domain DDTs are dynamically allocated/deallocated
    - Linked together in a tree to represent nest hierarchy; root pointer is **head_grid**, defined in frame/module_domain.F
    - Supports recursive depth-first traversal algorithm (frame/module_integrate.F)




```
RECURSIVE SUBROUTINE integrate ( domain , stoptime )
  WHILE ( domain.time <= stoptime )
    CALL solve ( domain )   ! Advances domain.time
      FOR EACH CHILD OF DOMAIN
        CALL integrate ( domain.child , domain.time )
```

Data Structures

- What you see depends on where you are (Cont.)
  - Model layer
    - All data objects are scalars and arrays of simple types only
    - Virtually all passed in through subroutine argument lists
  - Mediation layer
    - One task of mediation layer is to dereference fields from DDTs
    - Therefore, sees domain data in both forms, as DDT and as individual fields
  - The name of a data type and how it is referenced may differ depending on the level of the architecture

Heap Storage

- Data stored on the process heap is not thread- safe and is generally forbidden anywhere in WRF
  - COMMON declarations
  - Module data
- Exception: If the data object is:
  - Completely contained and private within a Model Layer module, and
  - Set once and then read-only ever after, and
  - No decomposed dimensions.

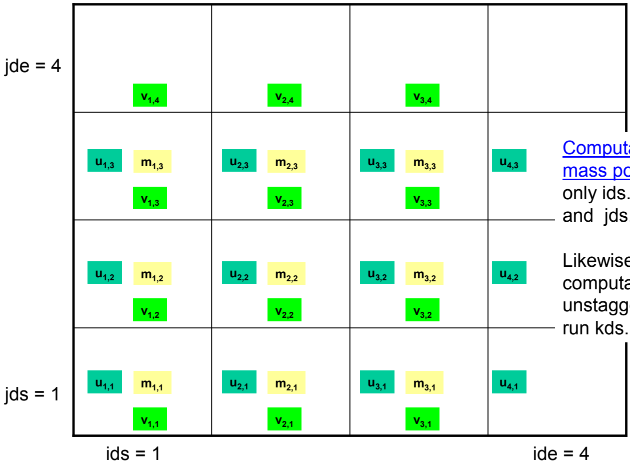
Grid Representation in Arrays

- Increasing indices in WRF arrays run
  - West to East (X, or I-dimension)
  - South to North (Y, or J-dimension)
  - Bottom to Top (Z, or K-dimension)
- Storage order in WRF is IKJ but this is a WRF Model convention, not a restriction of the WRF Software Framework

Grid Representation in Arrays

- The extent of the logical or *domain* dimensions is always the "staggered" grid dimension. That is, from the point of view of a non-staggered dimension, there is always an extra cell on the end of the domain dimension.

Grid Indices Mapped onto Array Indices (C-grid example)



Computation over mass points runs only ids..ide-1 and jds..jde-1

Likewise, vertical computation over unstaggered fields run kds..kde-1

LBC Arrays

- State arrays, declared in Registry using the **b** modifier in the dimension field of the entry
- Store specified forcing data on domain 1, or forcing data from parent on a nest
- All four boundaries are stored in the array; last index is over:
  - P_XSB (western)
  - P_XEB (eastern)
  - P_YSB (southern)
  - P_YEB (northern)

  These are defined in module_state_description.F

LBC Arrays

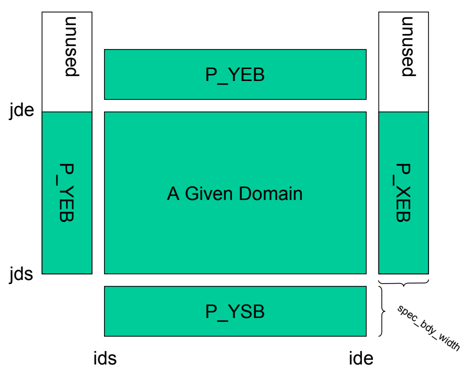
- LBC arrays are declared as follows:

```
em_u_b(max(ide,jde),kde,spec_bdy_width,4)
```

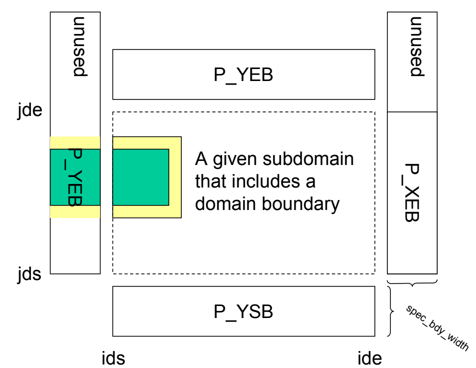
  - Globally dimensioned in first index as the maximum of x and y dimensions
  - Second index is over vertical dimension
  - Third index is the width of the boundary (namelist)
  - Fourth index is which boundary

- Note: LBC arrays are **globally** dimensioned
  - not fully dimensioned so still scalable in memory
  - preserves global address space for dealing with LBCs
  - makes input trivial (just read and broadcast)

LBC Arrays



LBC Arrays



Four Dimensional Tracer Arrays

- State arrays, used to store arrays of 3D fields such as moisture tracers, chemical species, ensemble members, etc.
- First 3 indices are over grid dimensions; last dimension is the tracer index
- Each tracer is declared in the Registry as a separate **state** array but with **f** and optionally also **t** modifiers to the dimension field of the entry
- The field is then added to the 4D array whose name is given by the use field of the Registry entry

Four Dimensional Tracer Arrays

- Fields of a 4D array are input and output separately and appear as any other 3D field in a WRF dataset
- The extent of the last dimension of a tracer array is from PARAM_FIRST_SCALAR to num_*tracername*
  - Both defined in Registry-generated frame/module_state_description.F
  - PARAM_FIRST_SCALAR is a defined constant (2)
  - Num_*tracername* is computed at run-time in set_scalar_indices_from_config (module_configure)
  - Calculation is based on which of the tracer arrays are associated with which specific packages in the Registry and on which of those packages is active at run time (namelist.input)

Four Dimensional Tracer Arrays

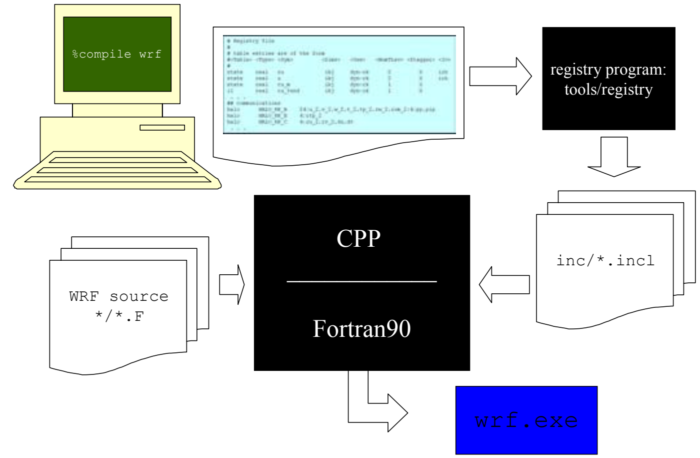
- Each tracer index (e.g. P_QV) into the 4D array is also defined in module_state_description and set in set_scalar_indices_from_config
- Code should always test that a tracer index greater than or equal to PARAM_FIRST_SCALAR before referencing the tracer (inactive tracers have an index of 1)
- Loops over tracer indices should always run from PARAM_FIRST_SCALAR to num_*tracername* -- EXAMPLE

The Registry

WRF Registry

- "Active data-dictionary" for managing WRF data structures
  - Database describing attributes of model state, intermediate, and configuration data
    - Dimensionality, number of time levels, staggering
    - Association with physics
    - I/O classification (history, initial, restart, boundary)
    - Communication points and patterns
    - Configuration lists (e.g. namelists)
  - Program for auto-generating sections of WRF from database:
    - <u>570</u> Registry entries ⇒ <u>30-thousand</u> lines of automatically generated WRF code
    - Allocation statements for state data, I1 data
    - Argument lists for driver layer/mediation layer interfaces
    - Interprocessor communications: Halo and periodic boundary updates, transposes
    - Code for defining and managing run-time configuration information
    - Code for forcing, feedback and interpolation of nest data
- Automates time consuming, repetitive, error-prone programming
- Insulates programmers and code from package dependencies
- Allow rapid development
- Documents the data

Registry Mechanics



Registry Data Base

- Currently implemented as a text file: Registry/Registry
- Types of entry:
  - *State* – Describes state variables and arrays in the domain structure
  - *Dimspec* – Describes dimensions that are used to define arrays in the model
  - *I1* – Describes local variables and arrays in solve
  - *Typedef* – Describes derived types that are subtypes of the domain structure
  - *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 communications for parallel matrix transposes

State entry

- Elements
  - *Entry*: The keyword "state"
  - *Type*: The type of the state variable or array (real, double, integer, logical, character, or derived)
  - *Sym*: The symbolic name of the variable or array
  - *Dims*: A string denoting the dimensionality of the array or a hyphen (-)
  - *Use*: A string denoting association with a solver or 4D scalar array, or a hyphen
  - *NumTLev*: An integer indicating the number of time levels (for arrays) or hypen (for variables)
  - *Stagger*: String indicating staggered dimensions of variable (X, Y, Z, or hyphen)
  - *IO*: String indicating whether and how the variable is subject to I/O and Nesting
  - *DName*: Metadata name for the variable
  - *Units*: Metadata units of the variable
  - *Descrip*: Metadata description of the variable
- Example

```
#      Type Sym  Dims   Use     Tlev Stag IO    Dname      Descrip
# definition of a 3D, two-time level, staggered state array

state  real ru   ikj    dyn_eh   2   X    irh   "RHO_U"  "X WIND COMPONENT"
```

Dimspec entry

- Elements
  - *Entry*: The keyword "dimspec"
  - *DimName*: The name of the dimension (single character)
  - *Order*: The order of the dimension in the WRF framework (1, 2, 3, or '-')
  - *HowDefined*: specification of how the range of the dimension is defined
  - *CoordAxis*: which axis the dimension corresponds to, if any (X, Y, Z, or C)
  - *DatName*: metadata name of dimension
- Example

```
#<Table>  <Dim>  <Order> <How defined>          <Coord-axis>  <DatName>
dimspec    i      1     standard_domain           x          west_east
dimspec    j      3     standard_domain           y          south_north
dimspec    k      2     standard_domain           z          bottom_top
dimspec    l      2     namelist=num_soil_layers  z          soil_layers
```

Rconfig entry

- This defines namelist entries
- Elements
  - *Entry*: the keyword "rconfig"
  - *Type*: the type of the namelist variable (integer, real, logical, string)
  - *Sym*: the name of the namelist variable or array
  - *How set*: indicates how the variable is set: e.g. namelist or derived, and if namelist, which block of the namelist it is set in
  - *Nentries*: specifies the dimensionality of the namelist variable or array. If 1 (one) it is a variable and applies to all domains; otherwise specify max_domains (which is an integer parameter defined in module_driver_constants.F).
  - *Default*: the default value of the variable to be used if none is specified in the namelist; hyphen (-) for no default
- Example

```
#          Type    Sym      How set              Nentries  Default
rconfig   integer dyn_opt  namelist,namelist_01    1          1
```

Package Entry

- Elements
  - *Entry*: the keyword "package",
  - *Package name*: the name of the package: e.g. "kesslerscheme"
  - *Associated rconfig choice*: the name of a rconfig variable and the value of that variable that choses this package
  - *Package state vars*: unused at present; specify hyphen (-)
  - *Associated* 4D *scalars*: the names of 4D scalar arrays and the fields within those arrays this package uses
- Example

```
# specification of microphysics options
package   passiveqv     mp_physics==0    -       moist:qv
package   kesslerscheme mp_physics==1    -       moist:qv,qc,qr
package   linscheme     mp_physics==2    -       moist:qv,qc,qr,qi,qs,qg
package   ncepcloud3    mp_physics==3    -       moist:qv,qc,qr
package   ncepcloud5    mp_physics==4    -       moist:qv,qc,qr,qi,qs

# namelist entry that controls microphysics option
rconfig   integer     mp_physics   namelist,namelist_04     max_domains    0
```

Comm entries: halo and period

- Elements
  - *Entry*: keywords "halo" or "period"
  - *Commname*: name of comm operation
  - *Description*: defines the halo or period operation
    - For halo: *npts:f1,f2,...[;npts:f1,f2,...]**
    - For period: *width:f1,f2,...[;width:f1,f2,...]**
- Example

```
# first exchange in eh solver
halo   HALO_EH_A  dyn_em 24:u_2,v_2,ru_1,ru_2,rv_1,rv_2,w_2,t_2;4:pp,pip

# a periodic boundary update
period PERIOD_EH_A dyn_em 2:u_1,u_2,ru_1,ru_2,v_1,v_2,rv_1,rv_2,rw_1,rw_2
```

I/O

WRF Model IO and Coupling

- WRF I/O and Coupling Streams
  - Streams: the logical data paths into and out of WRF
  - Available streams in WRF
    - Input, plus 5 auxiliary input streams
    - History, plus 5 auxiliary output streams
    - Dedicated output stream for Cycling 3DVAR
    - Boundary
    - Restart
  - Read from and written to in "variable-sets"
  - Variable-sets are defined at ***compile-time*** in the Registry
- Formats
  - The mechanism by which I/O is moved on a stream
  - Implemented using external packages and interfaced to the model through the WRF I/O and Model Coupling API (§7, WRF Design and Implementation Document)
  - Formats are specified at ***run-time*** in namelist.input
    - NetCDF (Format 2)
    - Parallel HDF5 (Format 4), thanks Kent Yang, NCSA
    - Experimental Model-Coupling interfaces through MCT, MCEL (Format 7)

WRF Model I/O and Coupling (Cont.)

- Lower levels of the WRF I/O software stack allow expression of a dataset open as a two-stage operation: OPEN BEGIN and then OPEN COMMIT
  - Between the OPEN BEGIN and OPEN COMMIT the program performs the sequence of writes that will constitute one frame of output to "train" the interface
  - An implementation of the API is free to use this information for optimization/bundling/etc. or ignore it
- Higher levels of the WRF I/O software stack provide a BEGIN/TRAIN/COMMIT form of an OPEN as a single call

I/O Software Stack

- Domain I/O
  - Operations that performs I/O on a stream for an entire domain
  - At this level all opens are single phase
  - The read/write calls to the per-field I/O routines below are Registry-generated
- Package-independent I/O API
  - Lower level opens (each step separate for multi-phase opens)
  - Read or write operation on a single field on a stream
  - Selects particular package-specific API routine to call based on io_form setting in namelist
- Package-specific I/O API
  - Package specific (and thus, external) implementation of each routine in the I/O API

Domain I/O

- Routines in share/module_io_domain.F and share/module_io_wrf.F
  - High level routines that apply to operations on a domain and a stream
    - open and define a stream for writing in a single call that contains the OPEN FOR WRITE BEGIN, the series of "training writes" to a dataset, and the final OPEN FOR WRITE COMMIT
    - read or write all the fields of a domain that make up a complete frame on a stream (as specified in the Registry) with a single call
    - some wrf-model specific file name manipulation routines
- Output_wrf and input_wrf
  - Contain hard coded WRF-specific meta-data puts (for output) and gets (for input)
    - Whether meta-data is output or input is controlled by a flag in the grid data structure
    - Meta data output is turned off when output_wrf is being called as part of a "training write" within a two-stage open
    - It is turned on when it's called as part of an actual write
  - Contain registry generated series of calls the WRF I/O API to write or read individual files

Package-independent I/O API

- frame/module_io.F
- These routines correspond to WRF I/O API specification
- Start with the wrf_ prefix (package-specific routines start with ext_*package*_)
- The package-independent routines here contain logic for:
  - selecting between formats (package-specific) based on the what stream is being written and what format is specified for that stream
  - calling the external package as a parallel package (each process passes subdomain) or collecting and calling on a single WRF process
  - passing the data off the the asynchronous quilt-servers instead of calling the I/O API from this task

Package-specific I/O API

- Format specific implementations of I/O
  - external/io_netcdf/wrf_io.F90
  - external/io_int/io_int.F90
  - external/io_phdf5/wrf-phdf5.F90
  - external/io_mcel/io_mcel.F90
- The NetCDF version contains a small program, diffwrf.F90, that uses the API read and then generate an ascii dump of a field that is readable by HMV (see: www.rotang.com) a small plotting program we use in-house for debugging and quick output.
- Diffwrf is also useful as a small example of how to use the I/O API to read a WRF data set

Defining a variable-set for an I/O stream

- Fields are added to a variable-set on an I/O channel in the Registry

```
#      Type Sym  Dims   Use      Tlev Stag IO    Dname       Descrip
state  real ru   ikj    dyn_eh   2    X    irh   "RHO_U"   "X WIND COMPONENT"
```

IO is a string that specifies if the variable is to be subject to initial, restart, history, or boundary I/O. The string may consist of 'h' (subject to history I/O), 'i' (initial dataset), 'r' (restart dataset), or 'b' (lateral boundary dataset). The 'h', 'r', and 'i' specifiers may appear in any order or combination.

The 'h' and 'i' specifiers may be followed by an optional integer string consisting of '0', '1', '2', '3', '4', and/or '5'. Zero denotes that the variable is part of the principal input or history I/O stream. The characters '1' through '5' denote one of five auxiliary input or history I/O streams.

Defining Variable-set for an I/O stream

`irh` -- The state variable will be included in the input, restart, and history I/O streams

`irh13` -- The state variable has been added to the first and third auxiliary history output streams; it has been removed from the principal history output stream, because zero is not among the integers in the integer string that follows the character 'h'

`rh01` -- The state variable has been added to the first auxiliary history output stream; it is also retained in the principal history output

`i205hr` -- Now the state variable is included in the principal input stream as well as auxiliary inputs 2 and 5. Note that the order of the integers is unimportant. The variable is also in the principal history output stream

`ir12h` -- No effect; there is only 1 restart data stream and ru added to it.

Assigning I/O Streams to Formats

- Run-time: specified in namelist.input file

```
io_form_history    = 2,
io_form_restart    = 2,
io_form_input      = 2,
io_form_boundary   = 2,
```

Nest Initialization, Forcing, and Feedback

- Three built-in streams for exchange of data between nested domains
  - DOWN: data from a coarse domain state array to a nested domain state array
  - UP: data from a nested domain state array to a coarse-domain state array
  - FORCE: data from a coarse domain array to boundary arrays for a nested domain array
- Format is specialized, parallel and built-in to WRF
- Like I/O streams, variable-sets on nest streams defined in Registry

Nest Initialization, Forcing, and Feedback

There are three streams that a variable may take between a coarse domain and a nested domain: *down,* indicated by a 'd' character in the *IO* string; *up*, indicated by a 'u'; and *force*, a special form of *down*, indicated with an 'f'.

If the stream identifier is specified by itself, a default interpolation subroutine is used. Down uses interp_fcn(), defined in share/interp_fcn.F, which is the semi-Lagrangian interpolator, SINT, from MM5 nesting. Up uses copy_fcn() by default, also defined in that source file.

There is no default for force; however, there is a function bdy_interp() (which also uses SINT) provided in share/interp_fcn.F.

When these are specified, the state variable is passed as an argument to the interpolation routine on both the coarse domain and the nest. If the state variable has multiple time levels, the highest numbered time level is passed.

Nest Initialization, Forcing, and Feedback

Different functions can be specified for nesting in the Registry, and additional fields can be provided to those functions, using the following syntax:

```
f=(my_bdy_fcn:dt,u_b,u_bt)
```

This will cause a different subroutine, named my_bdy_fcn, to be called instead of the default and the additional state variables dt, u_b, and u_bt (boundary and boundary tendency arrays, respectively) will be passed for both the coarse and nested domains.

The down, up, and force descriptions may be included in the same *IO* field for a state-entry: for example:

```
i01rhu=(my_feedback)d=(my_interp:mask)f=(bdy_interp:dt,u_b,u_bt)
```

This would specify that the state variable is input in the main input stream as well as the auxiliary-1 stream, it is part of restart and history data, it is downward forced using the user-supplied routine my_interp() which also takes the state variable mask as an argument; it is upward forced using the my_feedback() routine; and it is forced using the bdy_interp() routine, which takes as extra arguments the dt, u_b, and u_bt state variables.

Nest Initialization, Forcing, and Feedback

Given:

```
f=(my_bdy_fcn:dt,u_b,u_bt)
```

The interface to the subroutine should be as follows. Note the extra arguments defined for dt, u_b, and u_bt on coarse and nested domains. Note also that the registry-generated call to this routine will also provide two logical arguments to the routine indicating whether the variable is x-staggered or y-staggered.

```
SUBROUTINE my_bdy_fcn ( cfld,                                   &  ! CD field
                        cids, cide, ckds, ckde, cjds, cjde,     &  ! CD domain dims
                        cims, cime, ckms, ckme, cjms, cjme,     &  ! CD mem dims
                        cits, cite, ckts, ckte, cjts, cjte,     &  ! CD patch dims
                        nfld,                                   &  ! ND field
                        nids, nide, nkds, nkde, njds, njde,     &  ! ND domain dims
                        nims, nime, nkms, nkme, njms, njme,     &  ! ND mem dims
                        nits, nite, nkts, nkte, njts, njte,     &  ! ND patch dims
                        shw,                                    &  ! stencil half width
                        xstag, ystag,                           &  ! staggering of field
                        ipos, jpos,                             &  ! Nest lower left in CD
                        nri, nrj,                               &  ! nest ratios
                        cdt, ndt,                               &  ! extra vars on CD and ND
                        cbdy, nbdy,                             &  !  "     "    "    "
                        cbdy_t, nbdy_t                          &  !  "     "    "    "
                        )
```

Time Management and Error Handling

WRF Time management

- Implementation of ESMF Time Manager
- Defined in external/esmf_time_f90
- Objects
  - Clocks
  - Alarms
  - Time Instances
  - Time Intervals

WRF Time management

- Operations on ESMF time objects
  - For example: +, -, and other arithmetic is defined for time intervals intervals and instances
  - I/O intervals are specified by setting alarms on clocks that are stored for each domain; see share/set_timekeeping.F
  - The I/O operations are called when these alarms "go off". see MED_BEFORE_SOLVE_IO in share/mediation_integrate.F

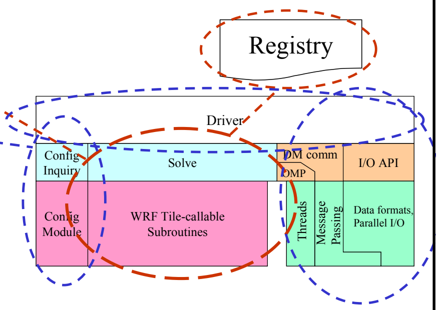
WRF Error Handling

- frame/module_wrf_error.F
- Routines for
  - Incremental debugging output WRF_DEBUG
  - Producing diagnostic messages WRF_MESSAGE
  - Writing an error message and terminating WRF_ERROR_FATAL

Example: Adding a New Core

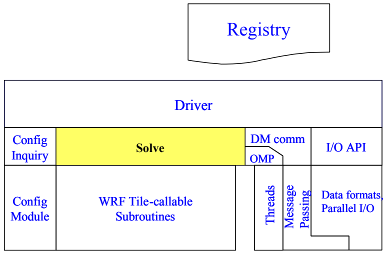
Example: Adding a new core

- Conceptual
  - WRF framework can slot in new dynamics as run-time selectable option
    - Changes to:
      - Mediation layer, model layer
      - Registry
    - Reuse:
      - Top-level driver layer
      - I/O infrastructure
      - Parallel infrastructure



Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing



SOLVE_EXP

PSEUDO CODE FOR NEW WRF SOLVER:

Time loop outside of solver (part of WRF driver: integrate)
DO 1 <- number of iterations

subroutine solve_exp (x_1 , x_2)

for each i,j

$$X_{2,ij} \Leftarrow \left\{ X_{1,i-1,j} \quad X_{1,i,j+1} \quad X_{1,i,j-1} \quad X_{1,i+1,j} \right\}$$

for each i,j

$$X_{1,ij} \Leftarrow X_{2,ij}$$

end subroutine solve_exp

End time loop

SOLVE_EXP

```
!WRF:MEDIATION_LAYER:SOLVER

SUBROUTINE solve_exp ( grid ,                   &
!
#include "exp_dummy_args.inc"
!
                 )

 CALL set_tiles ( grid , . . . )

#ifdef DM_PARALLEL
# include "HALO_EXP_A.inc"
#endif

     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL comp_1_into_2  ( x_1, x_2,                          &
                              ids, ide, jds, jde, kds, kde,      &
                              ims, ime, jms, jme, kms, kme,      &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                     )
     END DO


     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL copy_2_into_1  ( x_2, x_1,                          &
                              ids, ide, jds, jde, kds, kde,      &
                              ims, ime, jms, jme, kms, kme,      &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                     )
     END DO
```

SOLVE_EXP

```
!WRF:MEDIATION_LAYER:SOLVER

SUBROUTINE solve_exp ( grid ,                   &
!
#include "exp_dummy_args.inc"
!
                 )

 CALL set_tiles ( grid , . . . )

#ifdef DM_PARALLEL
# include "HALO_EXP_A.inc"
#endif

     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL comp_1_into_2  ( x_1, x_2,                          &
                              ids, ide, jds, jde, kds, kde,      &
                              ims, ime, jms, jme, kms, kme,      &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                     )
     END DO


     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL copy_2_into_1  ( x_2, x_1,                          &
                              ids, ide, jds, jde, kds, kde,      &
                              ims, ime, jms, jme, kms, kme,      &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                     )
     END DO
```

SOLVE_EXP

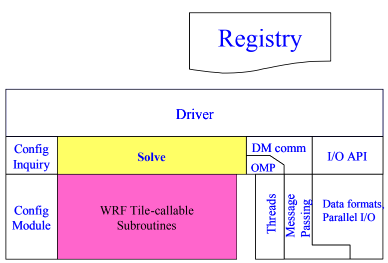
```
!WRF:MEDIATION_LAYER:SOLVER

SUBROUTINE solve_exp ( grid ,                &
!
#include "exp_dummy_args.inc"
!
                 )

 CALL set_tiles ( grid , . . . )

#ifdef DM_PARALLEL
# include "HALO_EXP_A.inc"
#endif

     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL comp_1_into_2  ( x_1, x_2,                            &
                              ids, ide, jds, jde, kds, kde,        &
                              ims, ime, jms, jme, kms, kme,        &
                              grid%i_start(ij), grid%i_end(ij),    &
                              grid%j_start(ij), grid%j_end(ij),    &
                              k_start, k_end                       )
     END DO


     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL copy_2_into_1  ( x_2, x_1,                            &
                              ids, ide, jds, jde, kds, kde,        &
                              ims, ime, jms, jme, kms, kme,        &
                              grid%i_start(ij), grid%i_end(ij),    &
                              grid%j_start(ij), grid%j_end(ij),    &
                              k_start, k_end                       )
     END DO
```


SOLVE_EXP

```
!WRF:MEDIATION_LAYER:SOLVER

SUBROUTINE solve_exp ( grid ,                &
!
#include "exp_dummy_args.inc"
!
                 )

 CALL set_tiles ( grid , . . . )

#ifdef DM_PARALLEL
# include "HALO_EXP_A.inc"
#endif

     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL comp_1_into_2  ( x_1, x_2,                            &
                              ids, ide, jds, jde, kds, kde,        &
                              ims, ime, jms, jme, kms, kme,        &
                              grid%i_start(ij), grid%i_end(ij),    &
                              grid%j_start(ij), grid%j_end(ij),    &
                              k_start, k_end                       )
     END DO


     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL copy_2_into_1  ( x_2, x_1,                            &
                              ids, ide, jds, jde, kds, kde,        &
                              ims, ime, jms, jme, kms, kme,        &
                              grid%i_start(ij), grid%i_end(ij),    &
                              grid%j_start(ij), grid%j_end(ij),    &
                              k_start, k_end                       )
     END DO
```


Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - <u>Model layer subroutines called by solver</u>
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing

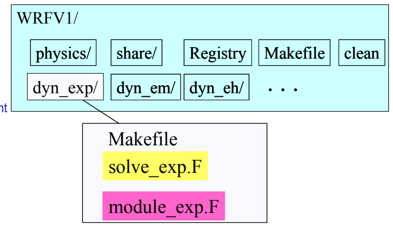


module_exp

```
!WRF:MODEL_LAYER:DYNAMICS
!
MODULE module_exp

   USE module_state_description

CONTAINS

!-------------------------------------------------------------------

SUBROUTINE comp_1_into_2   ( x1, x2,                         &
                             ids, ide, jds, jde, kds, kde,   &
                             ims, ime, jms, jme, kms, kme,   &
                             its, ite, jts, jte, kts, kte    )

   DO j = jts, jte
     IF ( j > jds .AND. j < jde-1 ) THEN
       DO k = kts, kte
         DO i = its, ite
           IF ( i > ids .AND. i < ide-1 ) THEN
             x2(i,k,j) = 0.25*(x1(i+1,k,j)+x1(i-1,k,j)+   &
                               x1(i,k,j+1)+x1(i,k,j-1))
           ENDIF
         ENDDO
       ENDDO
     ENDIF
   ENDDO

END SUBROUTINE comp_1_into_2

SUBROUTINE copy_2_into_1   ( x2, x1,                        &
                             ids, ide, jds, jde, kds, kde,   &
                             ims, ime, jms, jme, kms, kme,   &
                             its, ite, jts, jte, kts, kte    )
. . .
```

Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing



Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing

- Create dyn_exp/Makefile
- Edit top-level WRFV1/Makefile
- Additions to clean script

Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing

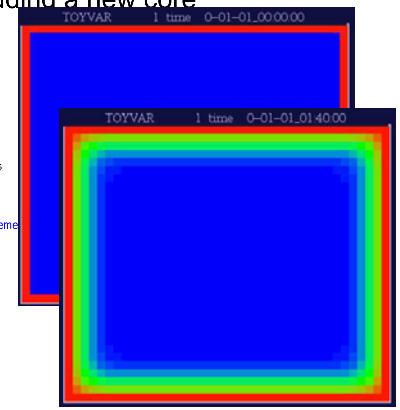
```
Additions to Registry/Registry file:

# define the state variable for new core
state real x ikj dyn_exp 2 - ih "TOYVAR"
# value of namelist variable dyn_opt for this core
package   dyn_exp       dyn_opt==4
# four-point halo-exchange on first time level of x
halo      HALO_EXP_A   dyn_exp    4:x_1
```

Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing

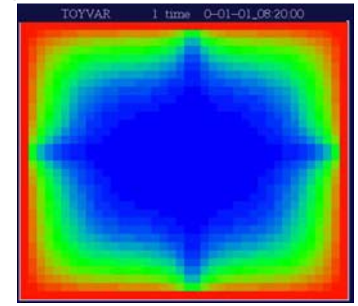
Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and impleme
    communication for parallelism
  - Multi-processor testing



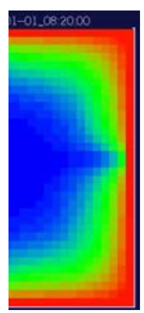
Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing



four processor run

```
!WRF:MEDIATION_LAYER:SOLVER

SUBROUTINE solve_exp ( grid ,              &
!
#include "exp_dummy_args.inc"
!
               )

 CALL set_tiles ( grid , . . . . )

#ifdef DM_PARALLEL
# include "HALO_EXP_A.inc"
#endif

     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL comp_1_into_2  ( x_1, x_2,                         &
                              ids, ide, jds, jde, kds, kde,     &
                              ims, ime, jms, jme, kms, kme,     &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                    )
     END DO


     !$OMP PARALLEL DO   &
     !$OMP PRIVATE ( ij )
     DO ij = 1 , grid%num_tiles
        CALL copy_2_into_1  ( x_2, x_1,                         &
                              ids, ide, jds, jde, kds, kde,     &
                              ims, ime, jms, jme, kms, kme,     &
                              grid%i_start(ij), grid%i_end(ij),  &
                              grid%j_start(ij), grid%j_end(ij),  &
                              k_start, k_end                    )
     END DO
```



Example: Adding a new core

- Steps
  - Develop new or convert existing code:
    - Mediation layer routine: solve
    - Model layer subroutines called by solver
  - Add to WRF
    - Add code to source tree
    - Incorporate into build mechanism
    - Registry entries: data, solver options, comms
    - Some additional splicing
  - Single processor testing
  - Analyze data-dependencies, define and implement communication for parallelism
  - Multi-processor testing