WRF Software: Code and Parallel Computing

John Michalakes, WRF Software Architect
Dave Gill

Introduction – WRF Software Characteristics

- Developed from scratch beginning around 1998, primarily Fortran and C
- Requirements emphasize flexibility over a range of platforms, applications, users, performance
- WRF develops rapidly. First released Dec 2000
- Supported by flexible efficient architecture and implementation called the WRF Software Framework

Outline

- WRF architecture – driver, mediation, model
- Need and design for parallelism
- Communication patterns to support parallelism
- Directory structure and file location overview
- Model layer interface
  - The “grid” struct
  - Indices
  - Dereferencing
- I/O

Introduction - WRF Software Framework Overview

- Implementation of WRF Architecture
  - Hierarchical organization
  - Multiple dynamical cores
  - Plug compatible physics
  - Abstract interfaces (APIs) to external packages
  - Performance-portable
- Designed from beginning to be adaptable to today’s computing environment for NWP
  http://mmm.ucar.edu/wrf/WG2/bench/
**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.

- **Mediation Layer**
  - **Solve** routine, takes a **domain object** and advances it one time step
  - **Nest** forcing, interpolation, and feedback routines

---

**Driver Layer**

- **Domains**: Allocates, stores, decomposes, represents abstractly as single data objects
- **Time loop**: Top level, algorithms for integration over nest hierarchy
WRF Software Architecture

- **Model Layer**
  - **Physics and Dynamics**: contains the actual WRF model routines are written to perform some computation over an arbitrarily sized/shaped, 3d, rectangular subdomain.

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
  - Cache
  - Memory
  - Secondary storage
  - Peripherals
- The implementation has been continually refined, but the basic idea hasn’t changed much.

Call Structure Superimposed on Architecture

Hardware has not changed much...

**IBM 7090**
- 6-way superscalar
- 36-bit floating point precision
- ~144 Kbytes
- ~50,000 fops/s
- 48hr 12km WRF CONUS in 600 years

**A computer in 2013**
- Dual core, 2.6 GHz chip
- 64-bit floating point precision
- 20 MB L3
- ~5,000,000,000 fops/s
- 48 12km WRF CONUS in 26 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
  - 100,000x increase in memory size
- We make up the difference with parallelism
  - Ganging multiple processors together to achieve $10^{11-12}$ flop/second
  - Aggregate available memories of $10^{11-12}$ bytes

$\sim 1,000,000,000,000$ flop/s \(
\sim 2500\) procs

48-h, 12-km WRF CONUS in under 15 minutes

WRF Domain Decomposition

- The WRF model decomposes domains horizontally
- For \(n\) MPI tasks, the two nearest factors \((n = k \cdot m)\) are selected; the larger is used to decompose the y-direction, the smaller is used to decomposed the x-direction
- Users may choose a preferred decomposition \((nproc_x, nproc_y)\)
- Prime numbers and composites with large prime factors are usually to be avoided
- The behavior of 132 vs 131, and 200 vs 202 are quite different
WRF Domain Decomposition

- As you increase the number of total MPI tasks, you reduce the amount of work inside of each MPI task.
- The amount of time to process communication between MPI tasks tends to be at best constant.
- As more MPI tasks are involved, more contention for hardware resources due to communication is likely.
- As the computation time gets smaller compared to the communications time, parallel efficiency suffers.

January 2000 Benchmark – 64 tasks: 10x8

- 74x61 grid cells
- 1 hour forecast, 3 minute time step, 20 time step average
- IO excluded

Decomposed domain sizes: proc count: I-dim x J-dim
1: 74x61
2: 74x31
4: 37x31
8: 37x16
16: 19x16
32: 19x8
64: 10x8

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>SM – OpenMP % Efficiency</th>
<th>DM – MPI % Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>98</td>
</tr>
<tr>
<td>4</td>
<td>65</td>
<td>91</td>
</tr>
<tr>
<td>8</td>
<td>31</td>
<td>83</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>70</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>40</td>
</tr>
</tbody>
</table>
January 2000 Benchmark

- WRF timing estimates may be obtained from the model print-out

**Serial**
Timing for main on domain 1: 32.16074 elapsed seconds

**OpenMP**
Timing for main on domain 1: 8.56216 elapsed seconds

**MPI**
Timing for main on domain 1: 7.36243 elapsed seconds

- Get enough time steps to include “day-time” radiation, and to have the microphysics “active” for better estimates

**Application: WRF**

- WRF can be run serially or as a parallel job
- WRF uses domain decomposition to divide total amount of work over parallel processes

**Parallelism in WRF: Multi-level Decomposition**

- Single version of code for efficient execution on:
  - Distributed-memory
  - Shared-memory (SMP)
  - 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, this is the scope of a mediation layer solver or physics driver.

- **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**

- Communication is required between patches when a horizontal index is incremented or decremented on the right-hand-side of an assignment.

  - On a patch boundary, the index may refer to a value that is on a different patch.

  - Following is an example code fragment that requires communication between patches

    ```plaintext
    Note the tell-tale +1 and –1 expressions in indices for rr, H1, and H2 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.
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

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-1,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-1))-

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-1,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-1))-

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-1,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-1))-

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

Distributed Memory MPI
Communications

• Halo updates

memory on one processor memory on neighboring processor
Distributed Memory (MPI) Communications

- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers
Where are WRF source code files located?

```bash
$ (RM) $@

$ (CPP) -I$(WRF_SRC_ROOT_DIR)/inc \n
$ (CPPFLAGS) $(OMPCPP) $*.F > $*.f90

$ (FC) -o $@ -c $(FCFLAGS) $(MODULE_DIRS) \n
$ (PROMOTION) $(FCSUFFIX) $*.f90

• The most important command is the “find” command. If there is an error in the model output, you can find that location in the source code with the find command.

  cd WRFV3
  find . -name ".F" -exec grep -i "Flerchinger" {} \; -print
```
Where are WRF source code files located?

- All of the differences between the .F and .f90 files are due to the included pieces that are manufactured by the Registry.
- These additional pieces are all located in the WRFV3/inc directory.
- For a serial build, almost 450 files are manufactured.
- Usually, most developers spend their time working with physics schemes.

Where are WRF source code files located?

- The “main” routine that handles the calls to all of the physics and dynamics:
  - WRFV3/dyn_em/solve_em.F
- This “solver” is where the tendencies are initialized to zero, some pre-physics terms are computed, and the time stepping occurs
- The calls to most of the physics schemes are made from a further call down the call tree
  - dyn_em/module_first_rk_step_part1.F

Where are WRF source code files located?

- Inside of solve_em and first_rk_step_part1, all of the data is located in the “grid” structure: grid%ht.
- The dimensions in solve_em and first_rk_step_part1 are “d” (domain), and “m” (memory):
  - ids, ide, jds, jde, kds, kde
  - ims, ime, jms, jme, kms, kme
- The “t” (tile) dimensions are computed in first_rk_step_part1 and passed to all drivers.
- WRF uses global indexing

Where are WRF source code files located?

- If you are interested in looking at physics, the WRF system has organized the files in the WRFV3/phys directory.
- In WRFV3/phys, each type of physics has a driver:
  - module_cumulus_driver.F cu
  - module_microphysics_driver.F mp
  - module_pbl_driver.F bl
  - module_radiation_driver.F ra
  - module_surface_driver.F sf
Where are WRF source code files located?

- The subgrid-scale precipitation (*.cu*.F)
  - module_cu_bmj.F
  - module_cu_camzm.F
  - module_cu_g3.F
  - module_cu_gd.F
  - module_cu_kf.F
  - module_cu_kfeta.F
  - module_cu_nsas.F
  - module_cu_osas.F
  - module_cu_sas.F
  - module_cu_tiedtke.F

Where are WRF source code files located?

- Advection
  - WRFV3/dyn_em/module_advect_em.F

- Lateral boundary conditions
  - WRFV3/dyn_em/module_bc_em.F

Where are WRF source code files located?

- Compute various RHS terms, pressure gradient, buoyancy, w damping, horizontal and vertical diffusion, Coriolis, curvature, Rayleigh damping
  - WRFV3/dyn_em/module_big_step_utilities_em.F

- All of the sound step utilities to advance u, v, mu, t, w within the small time-step loop
  - WRFV3/dyn_em/module_small_step_em.F

WRF Model Layer Interface – The Contract with Users

All state arrays passed through argument list as simple (not derived) data types

Domain, memory, and run dimensions passed unambiguously in three dimensions

Model layer routines are called from mediation layer (physics drivers) in loops over tiles, which are multi-threaded
Restrictions on Model Layer subroutines:

- No I/O, communication
- No stops or aborts
- Use `wrf_error_fatal`
- No common/module storage of decomposed data
- Spatial scope of a Model Layer call is one “tile”

**WRF Model Layer Interface**

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

**WRF Model Layer Interface**

```fortran
template for model layer subroutine

SUBROUTINE model_subroutine ( &
  arg1, arg2, arg3, ... , argn, &
  ids, ide, jds, jde, kds, kde, & ! Domain dims
  ids, ide, jds, jde, kds, kde, & ! Memory dims
  ims, ime, jms, jme, kms, kme, & ! Memory dims
  its, ite, jts, jte, kts, kte ) ! Tile dims
IMPLICIT NONE
!
! Define Arguments (State 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, ...
...
```

**WRF Model Layer Interface**

```fortran
template for model layer subroutine

SUBROUTINE model_subroutine ( &
  arg1, arg2, arg3, ... , argn, &
  ids, ide, jds, jde, kds, kde, & ! Domain dims
  ids, ide, jds, jde, kds, kde, & ! Memory dims
  ims, ime, jms, jme, kms, kme, & ! Memory dims
  its, ite, jts, jte, kts, kte ) ! Tile dims
IMPLICIT NONE
!
! Define Arguments (State 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, ...
...
```
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 = MAX(jts,jds), MIN(jte,jde-1)
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide-1)
      loc1(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

• Patch dimensions
  • Start and end indices of local distributed memory subdomain
  • Available from mediation layer (solve) and driver layer; not usually needed or used at model layer
WRF I/O

- Streams (similar to Fortran units): pathways into and out of model
- Can be thought of as files, though that is a restriction
  - History + auxiliary output streams (10 and 11 are reserved for nudging)
  - Input + auxiliary input streams (10 and 11 are reserved for nudging)
  - Restart, boundary, and a special DA in-out stream
  - Currently, 24 total streams
  - Use the large values and work down to stay away from “used”
- Non-chemistry: use history streams 13-22, 24
- Chemistry: use history streams 20, 21, 22, 24

WRF I/O

- Attributes of streams
  - Variable set
    - The set of WRF state variables that comprise one read or write on a stream
    - Defined for a stream at compile time in Registry
  - Format
    - The format of the data outside the program (e.g. NetCDF), split
    - Specified for a stream at run time in the namelist

WRF I/O

- Attributes of streams
  - Additional namelist-controlled attributes of streams
    - Dataset name
    - Time interval between I/O operations on stream
    - Starting, ending times for I/O (specified as intervals from start of run)

Example 1: Add output without recompiling

- Edit the namelist/input file, the time_control namelist record
- icfilds_filename = "myoutfields.txt" (MAXDOM)
- ic_form_auxhist24 = 2 (choose an available stream)
- auxhist24_interval = 10 (MAXDOM, every 10 minutes)
- Place the fields that you want in the named text file myoutfields.txt
- (:h:24): RAINC, RAINNC
- Where "+:h:" means ADD this variable to the output stream, "-:" is the history stream, and "24:" is the stream number
Outline

• WRF architecture – driver, mediation, model
• Need and design for parallelism
• Communication patterns to support parallelism
• Directory structure and file location overview
• Model layer interface
  – The “grid” struct
  – Indices
  – Dereferencing
• I/O