PILOT BASED FRAMEWORKS FOR WEATHER RESEARCH FORECASTING

BY DINESH PRASANTH GANAPATHI

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Dr. Shantenu Jha

and approved by

_____

_____

_____

New Brunswick, New Jersey

January, 2015

ABSTRACT OF THE THESIS

Pilot based frameworks for Weather Research Forecasting

by Dinesh Prasanth Ganapathi

Thesis Director: Dr. Shantenu Jha

The Weather Research Forecasting (**WRF**) domain consists of complex workflows that demand the use of Distributed Computing Infrastructure (DCI). Weather forecasting requires that weather researchers use different set of initial conditions and one or a combination of physics models on the same set of input data. For these type of simulations an ensemble based computing approach becomes imperative. Most DCIs have local job-schedulers that have no smart way of dealing with the execution of an ensemble type of computational problem as the job-schedulers are built to cater to the bare essentials of resource allocation. This means the weather scientists have to submit multiple jobs to the job-scheduler. In this dissertation we use **Pilot-Job** based tools to decouple work-load submission and resource allocation therefore streamlining the complex workflows in Weather Research and Forecasting domain and reduce their overall time to completion. We also achieve location independent job execution, data movement, placement and processing. Next, we create the necessary enablers to run an ensemble of tasks bearing the capability to run on multiple heterogeneous distributed computing resources there by creating the opportunity to minimize the overall time consumed in running the models. Our experiments show that the tools developed exhibit very good, strong and weak scaling characteristics. These results bear the potential to change the way weather researchers are submitting traditional **WRF** jobs to the

DCIs by giving them a powerful weapon in their arsenal that can exploit the combined power of various heterogeneous DCIs that could otherwise be difficult to harness owing to interoperability issues.

Acknowledgements

First and foremost, I would like express my heartfelt gratitude to Dr. Shantenu Jha for having faith in me and giving me the opportunity to work on this project. I am very thankful to Dr. Jha for his encouragement and support through all the years of my grad-school, without his guidance it would not have been possible to finish this dissertation. I would also like to thank Dr. Jha for the training, advice and motivation that kept me focused during the course of this project and overcome both professional and personal challenges in the last three years. Next, I would like to thank Dr. Dave Gochis and Dr. Wei Yu at the National Center for Atmospheric Research (NCAR) for providing me with insights into the aspects of Weather Research and patience they showed during the span of this collaborative project. I would also like to extend a special thanks to Dr. Dave Gochis for hosting me at the NCAR research facility and showing great enthusiasm under extenuating circumstances. Further I would like extend a big thanks to the developers in RADICAL group for their support, timely response and patience. I would like to thank Dr. Matteo Turilli for his timely advice and motivation and Mark Santcroos for working closely with me in understanding and resolving the technical issues. Next, I would like to thank my mom and dad, Mrs Sasi Rekha Ganapathi and Dr. Manickam Ganapathi for the Love, support and encouragement. I would also like to take this opportunity to thank my father for being a friend, philosopher, guide and inspiring me to take up engineering as a career. Lastly, I would like to thank my closest friends Prasoon Mishra, Manu Vohra and Late Varun Krishnamurthy for being there whenever I needed them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A plethora of modern day scientific research applications that are data and compute intensive require the use of Distributed Computing Infrastructure (DCI). Some of these applications consist of complex workflows but are confined to the use of a single resource; which owing to many constraints do not reach their peak efficiency in terms of their overall time to completion. Weather Research and Forecasting (**WRF** simulations) is one such domain where it becomes imperative to use High Performance Distributed Computing Systems to support the large scale computationally heavy simulations. The **WRF**(Weather Research Forecasting) requires many jobs to be executed in a chain of events to produce useful output (forecast) data. At most times these sequence of jobs have the same executables that needs to be coupled with same/different input data set and executed. Traditionally the weather scientists submit these sequence of jobs individually using traditional jobs scripts for a specific computing resource. For instance, scientists running **WRF** simulations on **TACC Stampede System**[4] would have login to Stampede and locally submit a SLURM job script and scientists who wish to use Yellowstone Computing Resources will have to login to Yellowstone and locally submit an LSF job script. This scenario changes if the scientists want run multiple simulation in an ensemble mode; Now they will have to submit multiple jobs to the job-scheduler. Once this is achieved and the jobs are launched, they wait in the batch queuing system and may not become active at the same time. Load and scheduling variations may add unnecessary hours to a job's total time to completion due to inefficiencies in scheduling many individual jobs. These problems discussed above serve as the motivation for this dissertation. The weather researchers could benefit largely from tools that encapsulate the following:

- Automate a lot of this cumbersome process and decouple work load submission and resource assignment.
- Support running the simulations in an ensemble mode.
- Able to work on a variety of heterogeneous systems with little or no change is manner in which job-submissions are made.
- Possessing the capability to be launched remotely from a laptop.

Being involved in the field of High Performance and Distributed Computing it is easy to look at the problem with the the bird's eye view and think about **Pilot-Jobs**. Pilot-Jobs can provide the much needed decoupling between job-submission and resource assignment. Also if a Pilot-Job framework that is easy to use has the capability to seamlessly work on heterogeneous distributed computing resources; could be a solution to weather researchers problem. We talk more about Pilot-Jobs in the subsequent sections.

1.1 Objectives

The main objective of this dissertation is to understand the pre-processing and processing stages of the **WRF and WRF-Hydro Coupled** workflows ; Analyze and develop an appropriate production grade software for weather researchers with the following capabilities:

- Enable remote job submission to Yellowstone High Performance System and retaining the look and feel of traditional job scripts that Weather scientists are used to.
- Support the run of an ensemble of tasks using **Pilot-Job** frameworks and enable location-independent job execution.
- Develop tools that are interoperable (with minimal modifications) across Heterogeneous Distributed Computing Infrastructures.

- Analyze the performance and scalability characteristics of the Pilot-based frameworks we develop.

While working towards accomplishing these goals we encountered we performed the following set of tasks:

- Take stock how things are being done traditionally by the weather researchers on the HPDCs.
- Run the **WPS** (WRF-Preprocessing System) and **WRF/ WRF-Hydro (coupled)** on Yellowstone, natively as the weather scientists would run it.
- Understand the workflow and identify what parts of the workflow that need automation, identify execution pattern (if any).
- Install and test **RADICAL-Pilot** on Yellowstone High-performance computing resource.
- Enable remote login capability for **RADICAL-Pilot** scripts to Yellowstone.
- Integrate current pipeline with **RADICAL-Pilot**.

1.2 Structure of the Dissertation

After discussing the motivation and objectives for this dissertation in Chapter 1 we discuss the background and nuances involved in **WRF**(Weather Research Forecasting) workflows in Chapter 2. In Chapter 3, we introduce and discuss the concept of Pilot-Jobs and the internals of **RADICAL-Pilot**- A **SAGA**(Simple API for Grid Application) based Pilot-Job framework that we extensively deploy in streamlining the **WRF** workflows. In Chapter 4, we analyze the **WRF** workflows and execution patterns and also discuss the implementation of the production-grade tools we have developed. In Chapter 5, we focus on experiments and results by extensively testing the developed frameworks. In Chapter 6, we arrive at the conclusion and discuss the foundations this dissertation lays for future work.

Chapter 2

Background

We start by discussing the Weather Forecasting Models in section 2.1 and go on to discuss **WRF**-Only and **WRF-Hydro** workflows in sections 2.2 -2.3.

2.1 Weather Forecasting Models

The Weather Research and Forecasting (**WRF**) Model is a next-generation mesoscale numerical weather prediction system designed to serve atmospheric research and operational forecasting needs. The model is built to service a plethora of meteorological applications across scales (from tens of meters to thousands of kilometers). Serious efforts to develop **WRF** began in the latter part of the 90's decade. It was a collaborative partnership principally among the National Center for Atmospheric Research (NCAR), the National Oceanic and Atmospheric Administration (represented by the National Centers for Environmental Prediction (NCEP) and the (then) Forecast Systems Laboratory (FSL)), the Air Force Weather Agency (AFWA), the Naval Research Laboratory, the University of Oklahoma, and the Federal Aviation Administration (FAA).[5][6]. WRF allows researchers to generate atmospheric simulations based on real data (observations, analyses). **WRF** offers operational forecasting a flexible and computationally-efficient platform, while providing advances in physics, numerics, and data assimilation contributed by developers in the broader research community. **WRF** is currently in operational use at NCEP, AFWA, and other centers[5]. In this dissertation we focus our efforts on building tools that are capable of supporting two kinds of simulation models, **WRF**-Only (hereafter synonymous with **WRF**) and **WRF-Hydro** simulation models. The essential control flow of both **WRF** and **WRF-Hydro** models share a common pre-processing stage, the WPS (WRF Preprocessing System). The

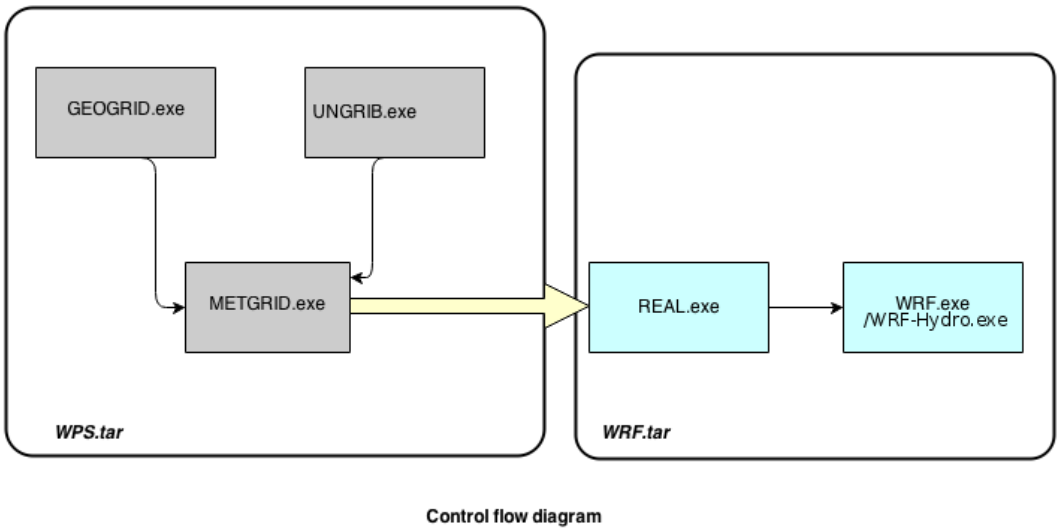

Figure 2.1: **WPS** and **WRF** Control Flow Diagram

control flow diagram for the two modeling systems are depicted in figure 2.1. The two boxes (`WPS.tar` and `WRF.tar`) represent the literal manner in which the software is distributed. The WPS contains `goegrid.exe` and `ungrib.exe` which generate input files needed for the execution of `metgrid.exe`. The execution of `metgrid.exe` generates input files need for the WRF/WRF-Hydro modeling stage which consists of `Real.exe` and `wrf.exe`/`wrf-hydro.exe`. The following sections contain detailed information about the WPS and WRF/WRF-Hydro stages. First we explain the WPS stage and then delve into the aspects of the WRF and WRF-Hydro workflows.

2.2 WPS -WRF Preprocessing System

The **WRF** modeling system consists of two main stages the **WRF** Preprocessing System (WPS) and the **WRF** modeling stage. The purpose of the WPS is to prepare the necessary input for **WRF** in order to perform real-data simulations. The pre-processing stage or the WPS stage is run offline as it is not computationally expensive. Though some of the executable/programs in the WPS package are parallelized, in most cases they are rarely submitted to the job-scheduler and are run on the login node or run offline. The WPS is common to the both, the **WRF** and the **WRF-Hydro** workflows. The WPS performs the following functions in order to prepare the initial data for the

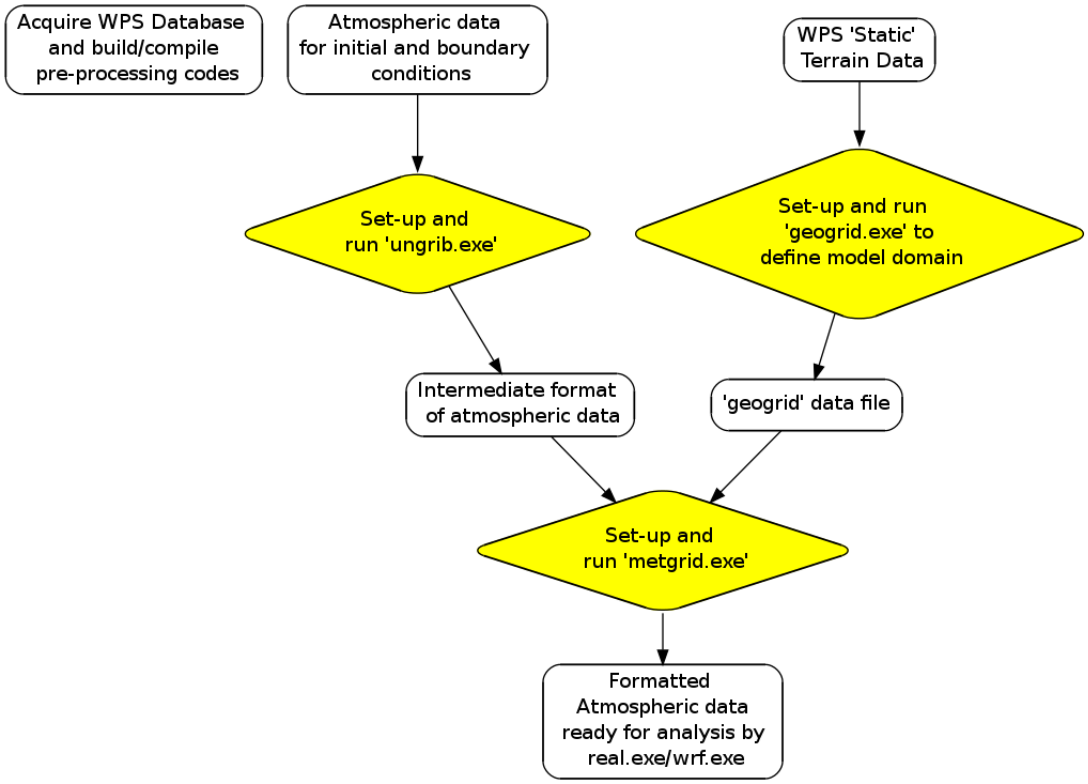

Figure 2.2: Flowchart of the **WRF** Pre-Processing System

WRF/WRF-Hydro modeling systems[7]:

- Defines the simulation coarse domain/nested domains (region of earth).
- Computes the latitude, longitude, map scale factors, and Coriolis parameters at every grid point.
- Performs the interpolation of time-invariant terrestrial data to simulation grids (e.g. terrain height and soil type).
- Performs the interpolation of time-varying meteorological fields onto simulation domains.

The steps involved in the WPS are depicted in figure 2.2. The yellow diamonds in the figure represent executables and white ovals represent data files.

2.2.1 Geogrid

Geogrid defines the following for the **WRF** model:

- The map projection (The real earth is (roughly) an ellipsoid but the **WRF** computational domains are defined by rectangles in the plane).
- Geographic location of domains(region of the earth).
- Dimensions of domains.

Apart from the above geogrid also interprets time-invariant Global data sets called the WPS Static Terrain data as shown in figure 2.2. Several of the data sets are available in only one resolution, but others are made available in resolutions of 30 arc seconds, 2, 5, and 10 arc minutes. This facilitates users who expect to work with domains having grid spacings that cover a large range. Once this is done; it does the following:

- Computes the latitude, longitude, and map scale factors at every grid point.
- Interpolates soil categories, land use category, terrain height, annual mean deep soil temperature, monthly vegetation fraction, monthly albedo, maximum snow albedo, and slope category to the model grids by default.

Output from geogrid is written in the **WRF** I/O API format, and thus, by selecting the NetCDF I/O format, geogrid can be made to write its output in NetCDF for easy visualization using external software packages, including ncview and NCL[8].

2.2.2 Ungrib

`ungrib` extracts meteorological fields from GRIB-formatted files. The `ungrib` program reads GRIB (GRIdded Binary or General Regularly-distributed Information in Binary form) files, "degribs" the data, and writes the data in a simple format called the intermediate format which can be used as input by metgrid.exe. The GRIB files contain time-varying meteorological fields and are typically from another regional or global model, such as NCEP's NAM or GFS models. The `ungrib` program can read both,

GRIB Edition 1 and GRIB Edition 2 files. GRIB files typically contain more fields than are needed to initialize **WRF**. Both versions of the GRIB format use various codes to identify the variables and levels in the GRIB file. `Ungrib` uses tables of these codes called Vtables, for "variable tables" to define which fields to extract from the GRIB file and write to the intermediate format. `Ungrib` can write intermediate data files in any one of three user-selectable formats: WPS a new format containing additional information useful for the downstream programs, SI the previous intermediate format of the **WRF** system; and MM5 format, which is included here so that `ungrib` can be used to provide GRIB2 input to the MM5 modeling system. Any of these formats may be used by WPS to initialize **WRF**, although the WPS format is recommended.

2.2.3 Metgrid

The `metgrid` program performs a horizontal interpolation on the intermediate-format meteorological data that is extracted by the `ungrib` program onto the simulation domains defined by the geogrid program. The interpolated metgrid output can then be used by the Real.exe program. The range of dates that are interpolated by `metgrid` are defined in a file called the WPS namelist file. The work of `metgrid` program is time-dependent so it is run every time a new simulation is initialized. Output from metgrid is written in the **WRF** I/O API format, and thus, by selecting the NetCDF I/O format, `metgrid` can be made to write its output in NetCDF for easy visualization using external software packages, including the new version of RIP4[8].

2.3 WRF-Only Workflow

The **WRF** or **WRF**-Only workflow is designed to provide a suite of physics that can be used for the modeling and forecasting but does not include any hydrological models. **WRF** is fully-parallelized and can be run on cluster and HPDC (High Performance and Distributed Computing) systems. **WRF** also offers a multitude of physics models which can be selected before running the simulations. Figure 2.3 shows the workflow for **WRF**-Only simulation model.

2.3.1 Real.exe

The `Real.exe` program uses the intermediate metgrid output files. `Real.exe` is parallelized and can be submitted to the job-scheduler. `Real.exe` is not computationally expensive compared to `wrf.exe` or `wrf-hydro.exe`. The `Real.exe` is responsible for the following:

- Create initial (WRF-input) and boundary condition (WRF-boundary) files for real-data cases
- Does vertical interpolation to model levels (when using WPS)
- Does vertical dynamic (hydrostatic) balance
- Does soil vertical interpolations and land-use mask checks

The output files generated by the `real.exe` are then used by `wrf.exe` to produce the WRF-out files that can be used to generate figures and graphs.

2.3.2 WRF.exe

The core of the computation problem that we try to solve in this dissertation is the execution of `wrf.exe` in ensemble mode. `wrf.exe` is parallelized an computationally expensive to run depending on what domain has been chosen and for how long / the range of dates for which the model is run. `wrf.exe` uses the (WRF-input) and boundary condition (WRF-boundary) files generated by the Real.exe and runs the model simulation with variety of selected namelist switches (such as physics choices, timestep, length of simulation,etc.). The output of the `WRF.exe` are WRF-out files that can be used to generate figures and graphs.

2.4 WRF-Hydro(Coupled) Workflow

The **WRF-Hydro** system was originally designed as a model coupling framework designed to facilitate easier coupling between the **WRF** (Weather Research and Forecasting) model and components of terrestrial hydrological models. **WRF-Hydro** is

both a stand-alone hydrological modeling architecture as well as a coupling architecture for coupling of hydrological models with atmospheric models. WRF-Hydro is fully-parallelized as a result of which it can be run on clusters and high performance computing systems alike. Like the **WRF** model it does not attempt to prescribe a particular or singular suite of physics but, instead, is designed to be extensible to new hydrological parameterizations. **WRF-Hydro** also possesses a multi-scale functionality to permit modeling of atmospheric, land surface and hydrological processes on different spatial grids[9]. Figure 2.4 shows the workflow for **WRF-Hydro** simulation model. The function Real.exe in this workflow is exactly the same as the **WRF**-Only workflow. The `Real.exe` produces the WRF-input files and the WRF-boundary files. These files and the geogrid file generated during the WPS stage can then be ingested by the `WRF-Hydro.exe` to produce the the WRF-out files that can be used to generate figures and graphs.

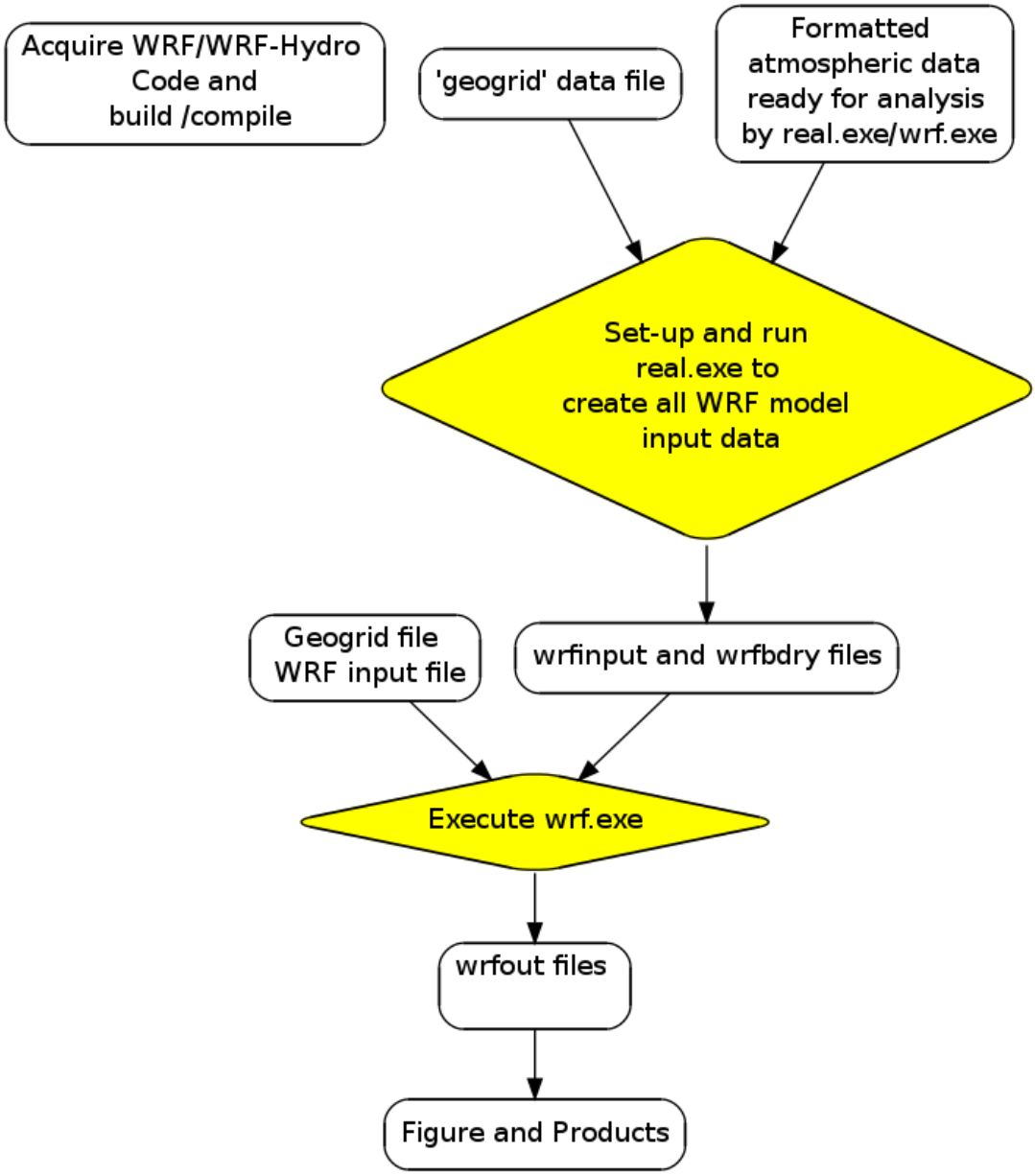

Figure 2.3: Flowchart of **WRF**-Only Modeling System

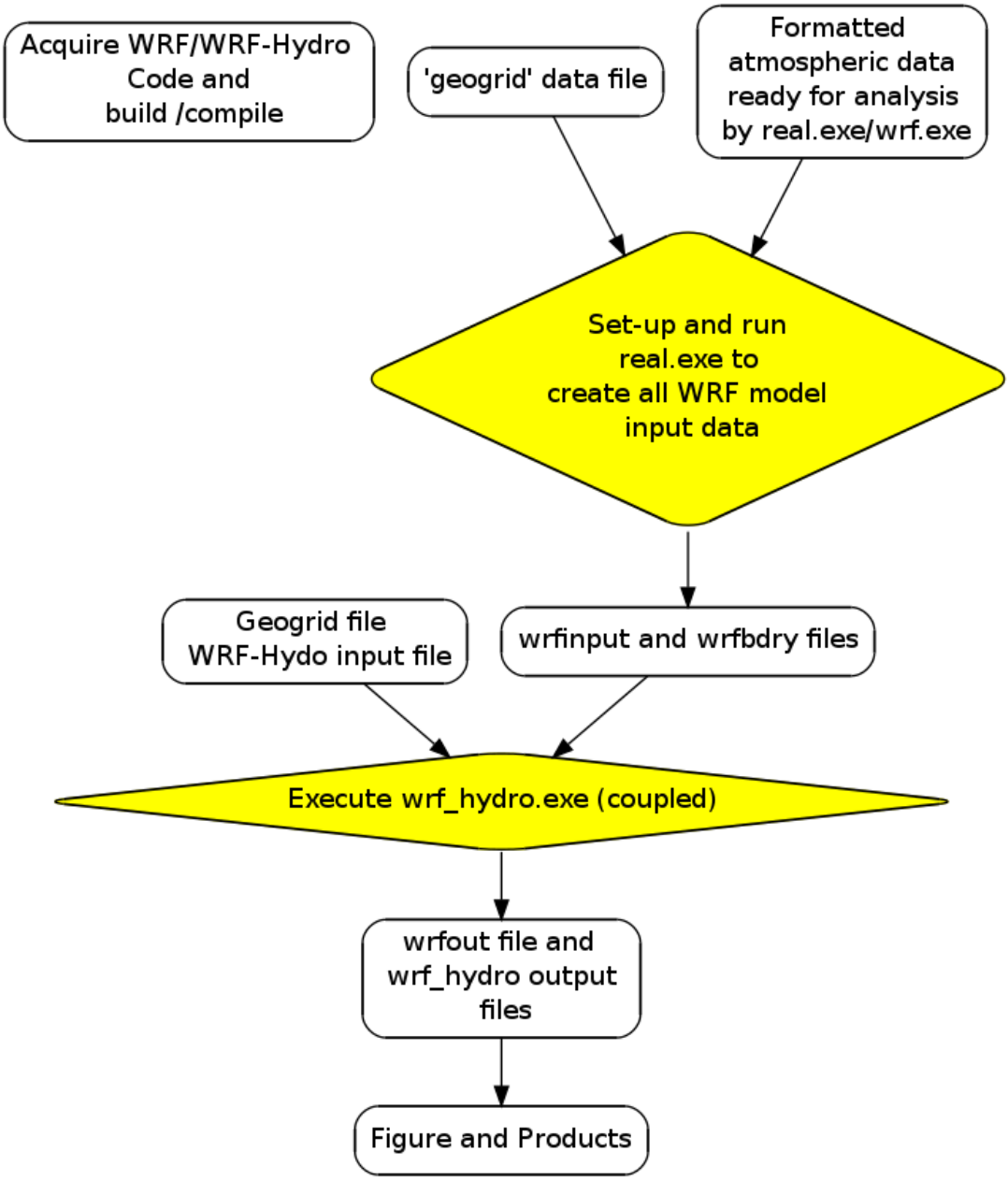

Figure 2.4: Flowchart of **WRF-Hydro** Modeling System

Chapter 3

Software Design Aspects

In this chapter we discuss Pilot-job frame works we use to implement location independent tools to execute **WRF** and **WRF-Hydro** in an ensemble mode in sections. First we introduce the concept of Pilot-Jobs and in the further sections we talk about the technical aspects of **RADICAL-SAGA** and **RADICAL-Pilot**.

3.1 Pilot-Jobs

Pilot-Job is a well known term in the field of High Performance and Distributed Computing. A Pilot-Job is a container job that encloses an array of smaller jobs (possibly idependent or coupled) within itself. The Pilot-jobs possess sophisticated workflow management capabilities to coordinate the launch and interaction of actual computational tasks (of interest to the application users) within the container. A Pilot-Job is treated as a single job by the computing resource's job-scheduler. Essentially by doing this the Pilot-Jobs decouple the resource assignment (reserving the processors/cores for a job) from the actual workload (the computational task). This flexible nature of this execution strategy promotes the distributed scaling-out of applications on multiple and possibly heterogeneous resources[10]. When a specific application needs to be executed in an ensemble mode or has a complex workflow where many jobs need to be executed, the application users have to submit these as individual tasks, there by not only accumulating individual queue wait times per task but also not promoting effective utilization of the available resources. In simple words, Pilot-Job provides an effective alternative approach. It can be pictured of as a container job for many sub-jobs. A Pilot-Job acquires the resources necessary to execute the sub-jobs (it requests for all of the resources required to run the sub-jobs, rather than just one sub-job at a time). For

example, if a system has a LSF-job scheduler, the Pilot-Job is submitted to one of its queues. Once it becomes active, it can run the sub-jobs directly, instead of having to wait for each sub-job to queue. This eliminates the need to submit a different job to the scheduler for every executable and significantly reduces the time-to-completion. In the last few years the concept of Pilot-Job has become increasingly popular, leading to a number of implementations. Some of the most widely known Pilot-Job frame-works are **RADICAL-Pilot** (formerly known as BigJob) [11], **DIANE** [12], **Falkon** [13], **DIRAC** [14] and **Condor-G** [15]. In this dissertation we focus our efforts on providing a Pilot-Job based solution to weather researchers running **WRF** and **WRF-Hydro** in a location independent ensemble mode with support for scaling-out on various distributed computing infrastructures.

3.2 RADICAL-SAGA

Simple API for Grid and Distributed Applications (**SAGA**) defines a high-level interface to the most commonly used distributed computing functionality. **SAGA** provides an access-layer and mechanisms for distributed infrastructure components like job schedulers, file transfer and resource provisioning services[**?**]. Given the heterogeneos nature of distributed infrastructure, SAGA addresses the indispensable need for an interoperability layer that lowers the complexity and brings simplicity to the use of distributed computing infrastructure. **RADICAL-SAGA** provides a Python module (also synonymously referred to as SAGA-Python) that is compliant with the **OGF**(Open Grid Forum) GFD.90 **SAGA** specification. Underneath the API, RADICAL-SAGA implements a flexible adaptor architecture. Adaptors are dynamically loadable modules that interface the API(Application Programming Interface) with various middleware systems and services. Many adaptors are already a part of **RADICAL-SAGA**. This facilitates the ease of using DCI for a diverse group of researchers and scientists. The the flexible architecture of **RADICAL-SAGA** offers scope to easily implement a new adaptor in case a backend system is not supported. Figure 3.1 shows the **RADICAL-SAGA** stack. The following are the list of adaptors that are supported:

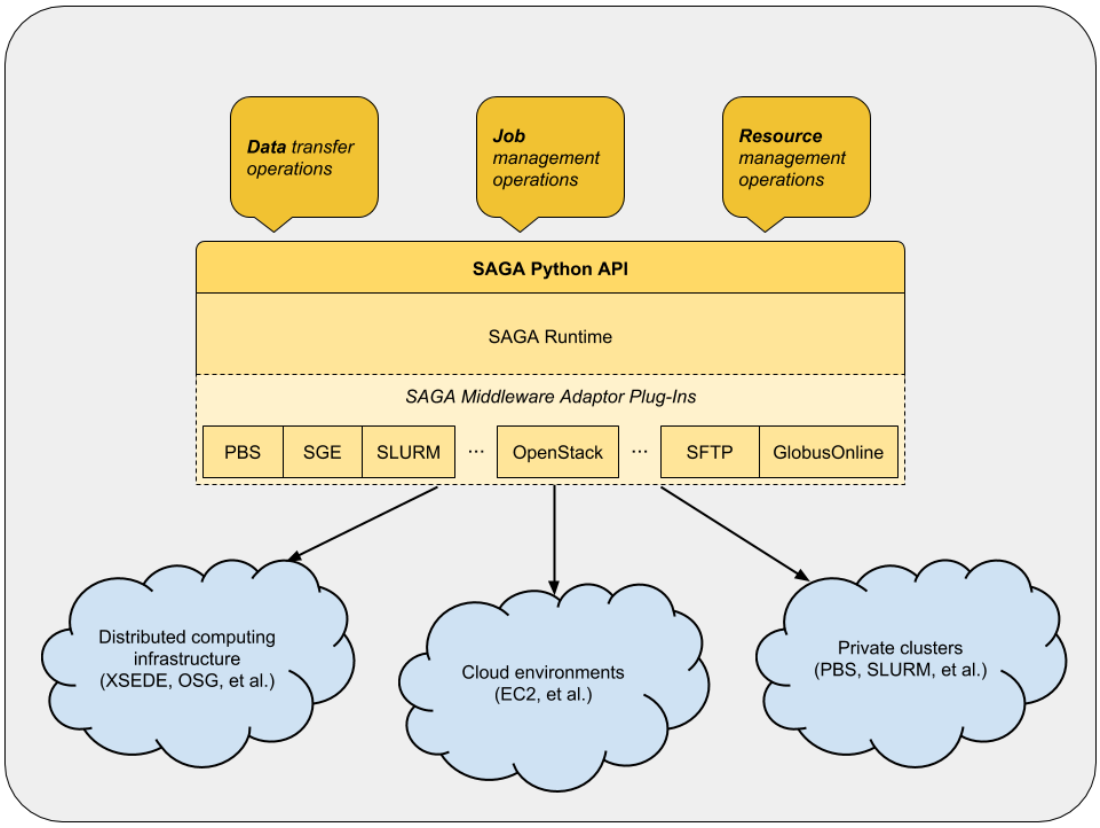

Figure 3.1: RADICAL-SAGA Stack [2]

- SSH and GSISSH
- Condor and Condor-G
- PBS and Torque
- Sun Grid Engine
- SLURM
- LSF(Load Sharing Facility)
- LoadLeveler

3.3 RADICAL-Pilot: RADICAL-SAGA based Pilot framework

RADICAL-Pilot (formerly BigJob) is a flexible Pilot framework that simplifies job and data management for clusters, grids and clouds. **RADICAL-Pilot** is written in

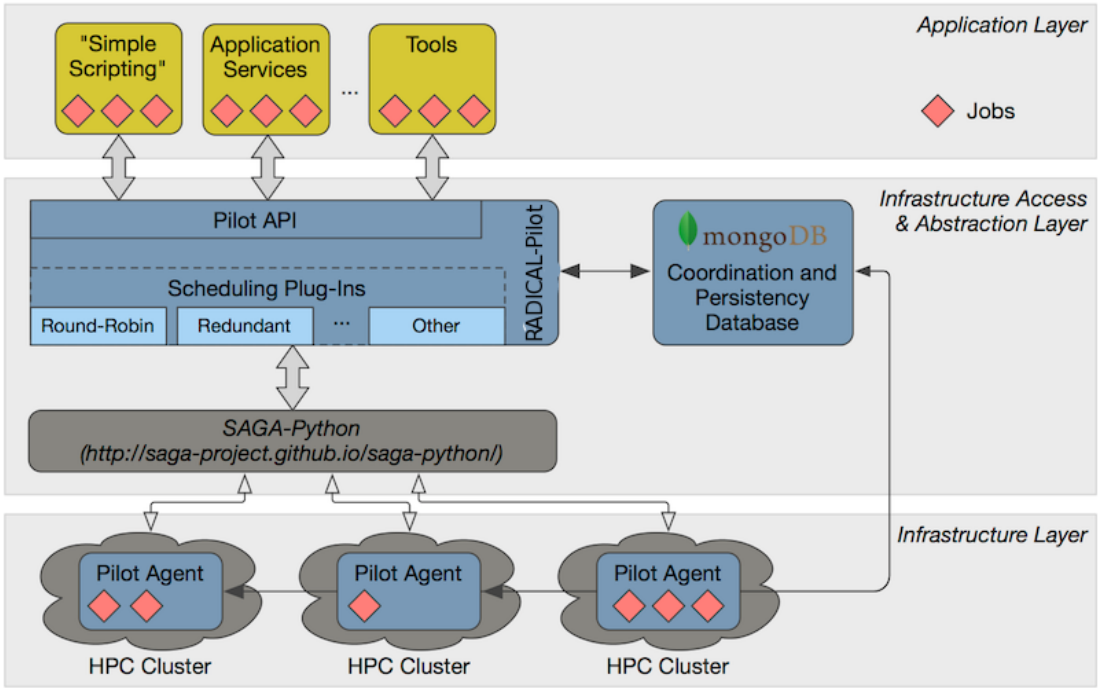

Figure 3.2: RADICAL-Pilot Architecture[3]

Python. It allows user-level control of *Pilots* and supports a wide range of diverse applications. It is built on top of The Simple API for Grid Applications which implies that RADICAL-Pilot works on a variety of backends such as PBS, LSF, SGE, Amazon EC2, etc. Apart from this, **RADICAL-Pilot** natively supports MPI (Message Passing Interface) jobs unlike many other Pilot-Job systems. This means that **RADICAL-Pilot** can be used to run a bag of MPI tasks in an ensemble mode. The architecture of **RADICAL-Pilot** and its integration with the **SAGA** layer are depicted in figure 3.2. In this model, the resource is acquired by a user application and the *'Compute-Units'*(actual computational task) are scheduled into the resource directly, rather than going through the job scheduler(of the system) for each task. In many cases, this can drastically shorten overall execution time as the individual *Compute Units* do not have to wait in the systems scheduler queue. *Compute Units* are often single-core / multi-threaded executables, but RADICAL-Pilot also supports the execution of parallel executables, based on MPI or OpenMP. This forms our basis for using RADICAL-Pilot to support the **WRF** workflows.

RADICAL-Pilot gives the user a programming library (Pilot-API) that provides

abstractions for accessing the resources and managing the task. With this library, the user can develop not just simple submission scripts but very complex applications, higher- level services and tools. We explain the terminologies, APIs and other details that are paramount in developing application specific **RADICAL-Pilot** scripts in next sub-sections, that will help us better understand the implementation section.

3.3.1 Loading the Module and Creating a Session

To use RADICAL-Pilot in a Python application/script, the `radical.pilot` module needs to be imported. This can be done using a simple import statement:

```
import radical.pilot
```

A radical.pilot.Session is the root object for all other objects in RADICAL- Pilot.It can be thought as a tree or a directory structure with a Session as root. Each Session can have zero or more `radical.pilot.Context`, `radical.pilot.PilotManager` and `radical.pilot.UnitManager` attached to it. A Session also encapsulates the connection(s) to a back end MongoDB server which is the brain and central nervous system of RADICAL-Pilot[16].

To create a new Session, the URL of a MongoDB server needs to be provided as follows:

```
session = radical.pilot.Session(MogoDB URL)
```

3.3.2 Creating a Compute Pilot

A radical.pilot.ComputePilot is responsible for `Compute Unit` (task) execution. Compute Pilots can be launched either locally or remotely, on a single machine or on one or more HPC clusters. ComputePilots are grouped in `radical.pilot.PilotManager` containers, so before you can launch a Compute Pilot, you need to add a Pilot Manager to your Session[16].

```
pmgr = radical.pilot.PilotManager(session=session)
```

In order to create a new Compute Pilot, first its requirements and properties need to be described. This is done with the help of a`radical.pilot.ComputePilotDescription` object. The mandatory properties that you need to define are:

- resource - The name (hostname) of the target system or localhost to launch a local ComputePilot.
- runtime - The runtime (in minutes) of the ComputePilot agent.
- cores - The number or cores the ComputePilot agent will try to allocate.

We can define and submit an 8-core local pilot that runs for 5 minutes like this:

```
pdesc = radical.pilot.ComputePilotDescription()
pdesc.resource = "localhost"
pdesc.runtime  = 5 # minutes
pdesc.cores    = 8
```

A Compute Pilot is launched by passing the `ComputePilotDescription` to the `submit_pilots()` method of the PilotManager. This adds the Compute Pilot to the Pilot Manager.

```
pilot = pmgr.submit_pilots(pdesc)
```

3.3.3 Creating Compute Units

Once a Compute Pilot is launched, we can generate a `radical.pilot.ComputeUnit` object for the Compute Pilot to execute. The Compute Unit can be thought of something very similar to an operating system process that consists of an executable, a list of arguments, and an environment along with some runtime requirements. Analogous to Compute Pilots, a Compute Unit is described via a `radical.pilot.ComputeUnitDescription` object. The mandatory properties that need to be defined are:

- executable - The executable to be launched.
- arguments - The arguments to be passed to the executable.
- cores - The number of cores required by the executable.

For example, you can create a workload of 8 /bin/echo Compute Units as follows:

```
compute_units = []
for unit_count in range(0, 8):
    cu = radical.pilot.ComputeUnitDescription()
    cu.executable = "/bin/echo"
    cu.arguments  = ["$Hello"]
    cu.cores      = 1

    compute_units.append(cu)
```

3.3.4 Input / Output File Transfer

In many applications, a computational task needs some input data. For this reason, a `radical.pilot.ComputeUnitDescription` allows the definition of input-data and output-data as follows:

- input-data defines a list of local files that need to be transferred to the execution resource before a ComputeUnit can start running.
- output-data defines a list of remote files that need to be transferred back to the local machine after a ComputeUnit has finished execution.

The following is an example of file1.dat and file2.dat being staged as input files:

```
cu = radical.pilot.ComputeUnitDescription()
cu.executable = "/bin/cat"
cu.arguments = ["file1.dat", "file2.dat"]
cu.cores = 1
cu.input_data = ["./file1.dat", "./file2.dat"]
```

3.3.5 Scheduling Compute Units

In order to execute the ComputeUnits on the ComputePilot, we need to create a `radical.pilot.UnitManager` instance. A Unit Manager combines three things-

- The ComputeUnits, added via `radical.pilot.UnitManager.submit_units()`
- One or more Compute Pilots, added via `radical.pilot.UnitManager.add_pilots()`
- A Unit Scheduler.

Once instantiated, a Unit Manager assigns the submitted CUs to one of its Compute Pilots based on the selected scheduling algorithm. This can be done as follows:

```
umgr = radical.pilot.UnitManager(session=session,
    scheduler=radical.pilot.SCHED_DIRECT_SUBMISSION)
umgr.add_pilots(pilot)
umgr.submit_units(compute_units)
umgr.wait_units()
```

The `radical.pilot.UnitManager.wait_units()` call blocks until all Compute Units have been executed by the Unit Manager. Simple control flows / dependencies can be realized with `wait_units()`, however, for more complex control flows it can become inefficient due to its blocking nature. To solve this problem, `RADICAL-Pilot` also provides mechanisms for asynchronous notifications and callbacks[16].

3.3.6 Closing and Cleanup

Before the application terminates, we must always call *radical.pilot.Session.close()* to ensure that the RADICAL-Pilot session terminates properly. `close()` will also delete all traces of the session from the database (control this with the cleanup parameter)[16]. This can be done as follows:

```
session.close(cleanup=True, terminate=True)
```

Chapter 4

Implementation of WRF Workflows Using RADICAL-Pilot

In the previous sections we discussed the **WRF/WRF-Hydro** workflows and learned how Pilot-Job frameworks can be useful in effectively and efficiently utilizing the HPDC resources available. Further, we delved into *RADICAL-Pilot*: A **SAGA** based Pilot-Job frameworks and its nuances. In this section we discuss the implementation of **WRF/WRF-Hydro** workflows using **RADICAL-Pilot**. In chapter 2 we learned that the WPS system is the same for both the **WRF** and **WRF Hydro**(coupled) workflows and it is not a computationally expensive task when compared to the execution of `wrf.exe` and `wrf-hydro.exe`. By changing variables in files called the namelist.wps the `geogrid.exe` and `ungrib.exe` work of different set of external data (different domains and different simulation intervals) and generate the necessary intermediate data as discussed on Chapter 2. this intermediate data is then operated upon by the `metgrid.exe` to produce horizontally interpolated intermediate data needed by the `real.exe`. So as far as the computation problem is concerned we can safely state that the link between the **WPS** and **WRF /WRF-Hydro** stages are the intermediate files (also called met-em files) generated by `metgrid.exe`.

4.1 Workflow Analysis

By analysing the workflow we can observe that `real.exe` can operate on either identical or different intermediate files depending on what the WPS generates. Once the `real.exe` is done with its execution the *wrfbdy* and *wrf-input* files are created and these are essentially used by either WRF or WRF-Hydro application kernel depeneding on what model we would like to run. Armed with this insight if we look at the

present execution strategy that is employed by the weather researchers, we observe the following:

- Remote Login into a resource.
- Run WPS and generate appropriate data needed for `real.exe`.
- Submit a local job script to the job-scheduler of the resource with `real.exe` as the executable.
- Wait for `real.exe` to finish executing and the manually submit another local job script to the job scheduler with `wrf.exe` or `wrf-hydro.exe`.
- If many simulations are needed the above steps need to be repeated as many times.

4.2 Execution Pattern

From our discussion in the previous sections it is evident that if we employ a pilot-solution to the application kernel of **WRF / WRF-Hydro**, we can run many simulations in an ensemble mode. Here each simulation would consist of first running the `real.exe` and generate the required files and then run a `wrf.exe` or `wrf-hydro.exe`. Essentially, the first task is the application kernel `real.exe` and the data generated, drives the next task `wrf.exe` or `wrf-hydro.exe`. This sort of task dependency is called **Chained tasks**, where the execution of the second task depends on the execution of the first task. Now when more than one such dependent tasks are run in an ensemble mode we have what is called a **Chained Ensemble**. The concept becomes more clear with an example, if we take Aa1,a2....aN and Bb1,b2....bN as two sets of tasks where each corresponding task in set B can only begin only once the corresponding task in set A is complete. For example, b1 can begin only when a1 is complete and b2 can only begin when a2 is complete but a1 and b2 do not have any such dependencies. Figure 4.1 illustrates a Chained Ensemble of tasks.

Once we understand the concept of *Chained Ensemble* tasks, it becomes easy to map the **WRF/WRF-Hydro** workflow into this execution pattern. Figure 4.2 illustrates

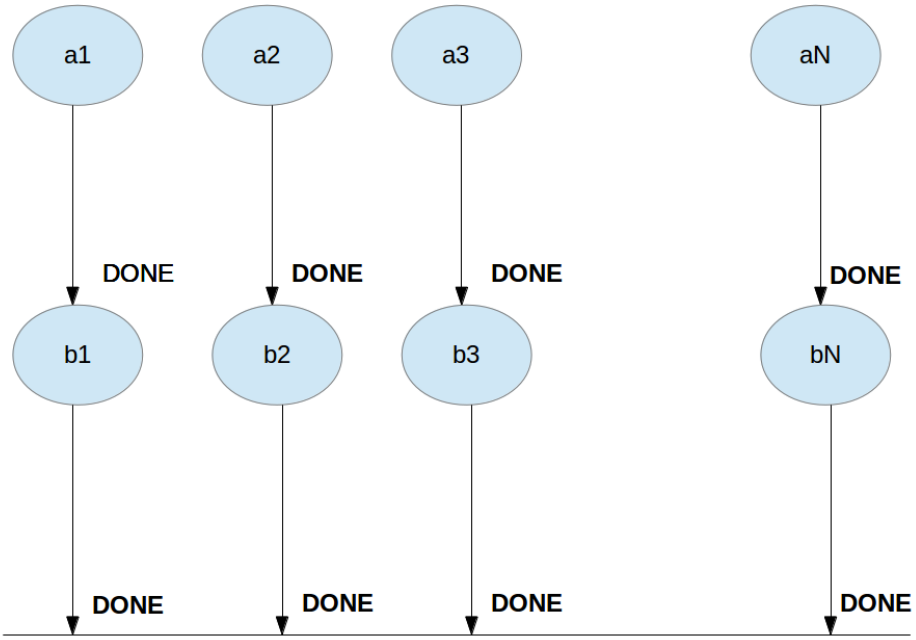

Figure 4.1: Chained Ensemble Execution Pattern

the mapping of the **WRF/WRF-Hydro** workflow into a two stage Chained Ensemble pattern, where the first stage is the execution of `real.exe` and the next stage is the execution of `WRF.exe`/`WRF-Hydro.exe`.

4.3 Implementation

4.3.1 Transition

While developing the *RADICAL-Pilot* scripts we have given thought to the fact that the weather researchers are very used to submitting job scripts on local resources. In oder to ease their transition from traditional job scripts to *RADICAL-pilot* scripts we have tried to maintain the same look and feel of a job script and we mention the variables (that are traditionally a part of the job submission script) at the beginning of the *RADICAL-Pilot* script in the Pilot-setup section. Here is an example of how this is accomplished:

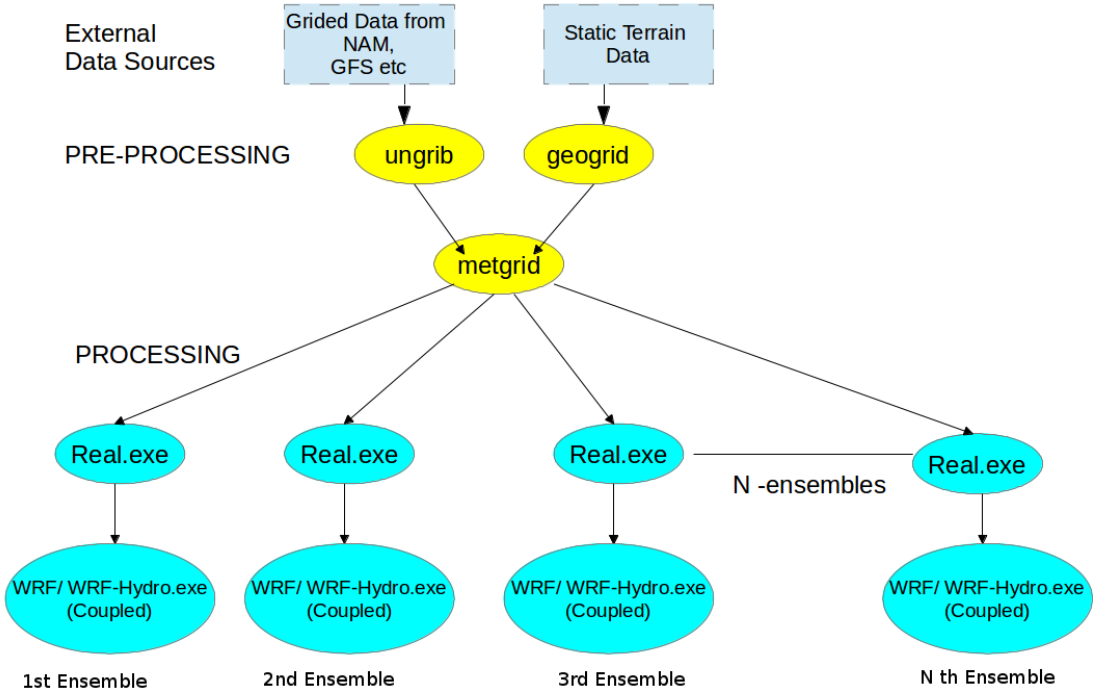

Figure 4.2: Mapping **WRF/WRF-Hydro** Workflows to Chained Ensemble Execution Pattern

---

```
# Sandbox directory on the resource
WORKDIR    = "*****"
#  NUMBER_JOBS is the Number of Ensembles of wrf.exe or wrf_hydro.exe you
    would like to run
NUMBER_JOBS =16
#  TOTAL NUMBER OF CORES to be reserved. (number of cores for 1 ensemble *
    number of ensembles)
TOTAL_NUMBER_CORES_RESERVED =256
#  NUMBER of cores needed for wrf.exe / wrf_hydro.exe in one ensemble
CORES_FOR_WRF =16
#  NUMBER of cores needed for real.exe in one ensemble
CORES_FOR_REAL =16
#  USER_ID for yellowstone
user_id_yellowstone= "****"
```

```
# WALL TIME REQUESTED
walltime_requested=50
# PROJECT ID
project_id_for_yellowstone ="****"
# QUEUE
queue_on_yellowstone ="****"
# Name of the executable, can be ./wrf.exe or ./wrf_hydro.exe
application_kernel_name='./wrf.exe'
```

4.3.2 Structure

In the ensemble mode for WRF/WRF-Hydro simulations, the *RADICAL-Pilot* script creates twice the number of *Compute Units* (discussed in Chapter2) as the number of Ensembles required. So if a user wishes to run N ensembles then the *RADICAL-Pilot* creates 2N number of *Compute Units*. N of these *Compute Units* are reserved to run N instances of `real.exe` and the rest N *Compute Units* are reserved for **WRF/WRF-Hydro** executables. Since we are implementing a *2-Stage Chained Ensemble* execution pattern we need to make sure that the compute units containing the **WRF/WRF-Hydro** executables start their execution only when the *Compute Units* containing their corresponding `real.exe` are done executing. This is accomplished by keeping track of the compute units associated with the instances of `real.exe` in a Python list and constantly polling to check if they are done using the `wait()` function. If a *Compute Unit* associated with one of the instances is complete then it will be removed from the list and the *Compute Unit* attached to its corresponding *WRF/WRF-Hydro* executable will automatically become active. This enforces the execution of `real.exe` before the execution of the `wrf/wrf-Hydro.exe`. Once all the *Compute Units* finish their execution the script gracefully exits.

Chapter 5

Experiments and Results

5.1 Hardware Resources

The **RADICAL-Pilot** frameworks for **WRF** are very flexible and can utilize a number of heterogeneous resources. At the time of this dissertation initially three such resources were considered, namely, Yellowstone High Perfomance Computing Resource, TACC Stampede and SuperMUC Petascale Systems. **RADICAL-Pilot** has successfully run and enabled many applications on all of these three systems, but at the time of this dissertation **WRF and WRF-Hydro (Coupled)** were compiled and running only on Yellowstone High Performance Computing Resource. Work is in progress to compile and have **WRF** run on both Stampede and SuperMUC. Once this is done, it will be a matter of changing a few parameters in the existing frameworks to have **WRF and WRF-Hydro (Coupled)** run on all these resources in ensemble mode. Due to this reason we confine our experiments and tests to Yellowstone. Yellowstone is a 1.5-petaflops high-performance IBM iDataPlex cluster, which features 72,576 Intel Sandy Bridge processors and 144.6 TB of memory. Yellowstone's hardware is described in table 5.1

5.2 Test Criteria and Parameters

As a part of our experimental setup we execute a **RADIACL-Pilot** script from a laptop and run **WRF** in an ensemble mode on Yellowstone. For our experiments we use the GFS data (Global Forecasting System) for the floods in Colorado (end of April 2014). At the last stage of the *WPS*, Metgrid.exe finishes its execution to produce 24 intermediate *met-em* files of average size 145MB. As for the input data, we repeat the

Table 5.1: Yellowstone High Performance Computing Resource Specifications [1]

72,576 processor cores	2.6-GHz Intel Xeon E5-2670 (Sandy Bridge) processors with Advanced Vector Extensions (AVX)8 flops per clock
4,536 computation nodes	IBM dx360 M4,dual socket, 8 cores per socket
6 login nodes	IBM x3650 M4, dual socket, 8 cores per socket
144.58 TB total system memory	2 GB/core,32 GB/node (25 GB usable)
FDR Mellanox InfiniBand interconnect	2.6-GHz Intel Xeon E5-2670 (Sandy Bridge) processors with Advanced Vector Extensions (AVX), 8 flops per clock
1.504 petaflops peak	1.26 petaflops HPL, 28.9 Bluefire-equivalents

experiment for two sets of data as mentioned below:

- Prepare and use small input data set; We use a single *met-em* file corresponding to a single GRIB file containing 3 hours worth of weather data (starting from 12:00am to 3:am on 28 April, 2014). The resolution of the source data while interpolating the static terrain data is set to less than 0.9km. We use the single *met-em* files as the input file for all the Real.exe executables.

- Prepare and use a large input data set; We use 24 *met-em* files corresponding to a 24 GRIB files containing 72 hours worth of weather data (starting from 12:00am on 28 April, 2014 to 12:am on 1 May, 2014). The resolution of the source data while interpolating the static terrain data is set to less than 0.9km. We use the 24 *met-em* as the input for all the Real.exe executables.

For the ease of understanding the differences between the large and small data sets are tabulated in table 5.2.

Before we proceed further, let us pause for a moment and discuss the concept of strong and weak scaling as we will be applying them to our use case.

In strong scaling, a program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used

Table 5.2: Comparison Between Small and Large Input Data Sets

Input Data Type	Number of `met-em` files	Avg size of 1 file	Input data description
Small	1	145 MB	Input data includes weather data starting from 12:00am to 3:am on 28 April, 2014
Large	24	145 MB	Input data includes weather data starting from 12:00am to 12:00 am on 28 April to 1 May, 2014

(N). In general, it is harder to achieve good strong-scaling at larger processor counts since the communication overhead for many/most algorithms increases in proportion to the number of processes used. In order to see the strong scaling behavior of an application the problem size is fixed and number of processing units are varied[17]. When the problem size (workload) assigned to each processing element stays constant and additional elements are used to solve a larger total problem; it becomes the case of weak scaling. Here linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors[17]. The next set of steps we execute are common to both the sets of data and are as follows:

- We run `real.exe` at a fixed core count of 16 cores per ensemble. We do this because running Real is not a computationally challenging task but we encapsulate it in our framework for efficient automation of the workflow.
- Next, we run `wrf.exe` by varying the core count from 16 to 256 cores per executable. This demonstrates the strong scaling of the application-kernel as we increase the the processor count for the same executable (`wrf.exe`). In other words strong scaling refers to keeping the ensemble members fixed and varying the resources (number of cores) used per ensemble member.
- Next we test the weak scaling of the application kernels by running them in

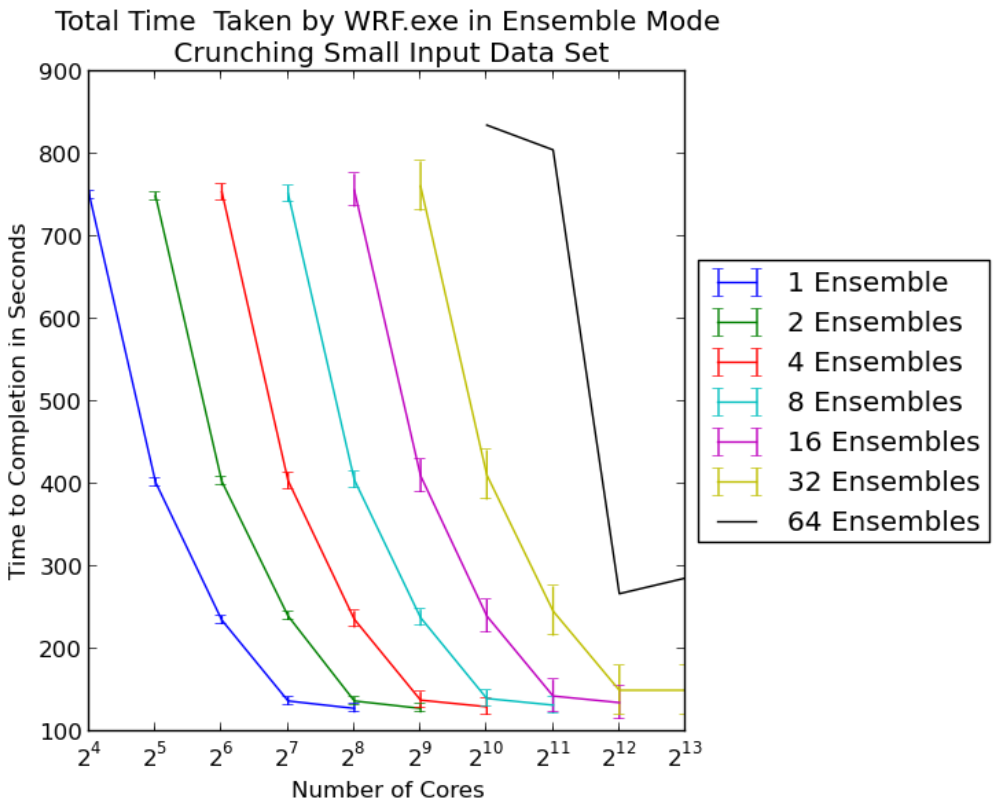

Figure 5.1: The figure depicts the strong scaling characteristics for the small input data set by plotting number of cores used vs Total time to Completion. Here each line represents a configuration where the ensemble size is fixed and ranges over 1 to 64; for each of these configurations the number of cores per ensemble member is varied from 16 through 256. Each configuration shows a linear decrease in the total time consumed when the number of cores per ensemble member is increased from 16 through 256.

ensemble mode. Weak scaling refers to keeping the resources (number of cores) per ensemble member constant and increasing the ensemble size (more ensembles). In order to do this we first, we fix the core count per ensemble (running `wrf.exe` to 16 and vary the number of ensembles from 1 to 64. Next, we increase the core count per executable in steps to 32,64,128 and 256 and observe the behavior by varying the number of ensembles from 1 to 64.

- Finally, we benchmark the performance of **RADICAL-Pilot**.

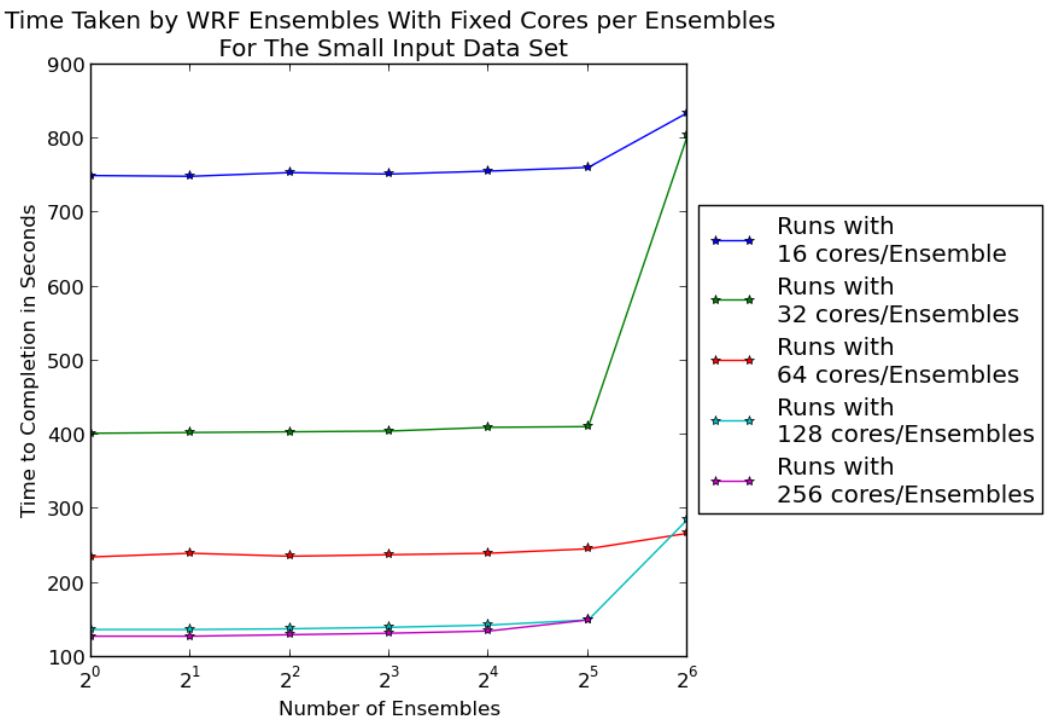

Figure 5.2: The graphs shows the total time taken to run **WRF** ensembles while fixing the number of cores per ensemble member for the small input set. This figure depicts the weak scaling of the **WRF** application kernel. As we increase the ensemble size keeping the number of cores per ensemble member fixed we observe excellent weak scaling. When we increase the ensemble size from 1 through 32 the total time consumed remains constant as long as the number of cores per ensemble member is fixed.

5.3 Analysis and Discussion

After designing and carrying out the experiment we assimilate the data and analyze it in this section. First we discuss the small input data **WRF** simulations and then we discuss the large input data set. In order to do this, we fix the ensemble size (number of ensemble members) at 1,2,4,8,16,32 and 64. For each of the ensemble size, we plot total time taken when the number of cores per ensemble member is varied from 16 through 256 and capture the strong scaling characteristics. In figure 5.1 the blue line represents the runs made for 1 ensemble; each successive data point marked on this line are the time taken 16-256 cores are used to run 1 ensemble member. We observe that each ensemble run shows a linear decrease in the total time consumed when the number of cores per ensemble member is increased from 16 through 256. This pattern

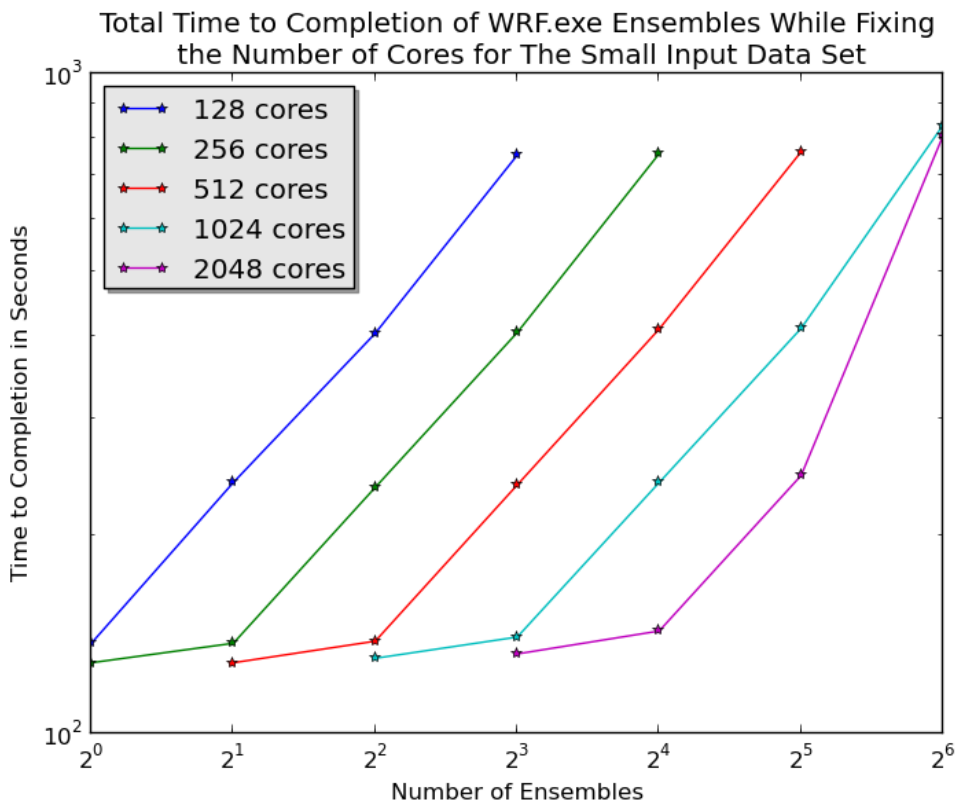

Figure 5.3: Graph depicts the number of ensembles vs total time to completion in log scale for the small input data set. For each core count (varying from 128 cores to 2048 cores) the time to completion of 1 to 64 ensembles is shown. If we fix the resources (cores); lesser the ensemble members (more cores per ensemble member) lesser the time to completion.

is identical for 1,2,4,8,16 and 32 ensembles as the lines have the same slope and shape. The error bars on the runs for 1,2,4,8,16 and 32 ensemble members are 5,5,10,10,20 and 30 seconds respectively. Due to shortage of allocation hours on the HPC resource we did not make multiple runs for 64 ensemble members to determine the error bars. If we look at the error bars they are consistently well within 5 percent.

The patterns observed in figure 5.1 urge us to use the data gathered and plot/study how the total time taken changes when we fix the number of cores used per ensemble member and increase the ensemble size (run more number of ensembles members) from 1 through 64. This data is plotted in figure 5.2. Here it becomes very clear that the time taken to run ensemble size of 32 members is identical to the time taken by 1 ensemble member as long as the number of cores used per ensemble member is same.

Looking at the numbers this is great for task level parallelism. 32 ensembles using 256 cores per ensembles (a total of 8192 cores) takes the same time (within 5 percent error margin) as a single ensemble running on 256 cores. This augurs very well for the weather researchers. They will be able to run 32 tasks (of **WRF** simulations) in the time can run 1 (**WRF** simulation). Figure 5.2 also demonstrates the excellent weak scaling characteristics of **WRF** simulations with **RADICAL-Pilot**.

When we increase the number of ensemble members from 32 to 64 we observe a deviation from the expected flat line nature as seen in figure 5.2. The reason for this highlighted in figure 5.4 and 5.5. In these figures it is evident that **RADICAL-Pilot** takes about a 1 second (sometimes seen to be more sometimes) to assign a compute unit. Here though we see that the average time taken for one **WRF** ensemble to complete is when using 128 cores is around 137 seconds and found to be 402 seconds when using 32 cores per ensemble member it is observed that the net time to completion of all the **WRF** compute units are 306 seconds and 804 seconds respectively. This can be explained by the delayed execution of one or more compute units as shown in figure 5.4 and 5.5. The reason for this anomaly when go from 32 to 64 ensemble members is under investigation.

Next, for the small input data set we fix the amount of resources (cores) and vary the ensemble size (number of ensemble members) and plot the total time taken. For example if we have 128 cores at our disposal we could run 8 ensemble with 16 cores per ensemble member, 4 ensembles with 32 cores per ensemble, 2 ensembles with 64 cores per ensemble member or 1 ensemble member with 128 cores. We know from the discussion above that figure 5.1 that have good strong scaling characteristics for the application kernel and that we can expect a linear increase in total time completion when reduce the number of cores per ensemble member. This is confirmed by figure 5.3.

Now let us discuss the ensemble runs of **WRF** simulations using the large input data set. What would be an interesting question to ask is if **RADICAL-Pilot** does an equally good job when the **WRF** simulations crunch the large input data set we have defined earlier. We follow identical steps as we did with the small input data set. First

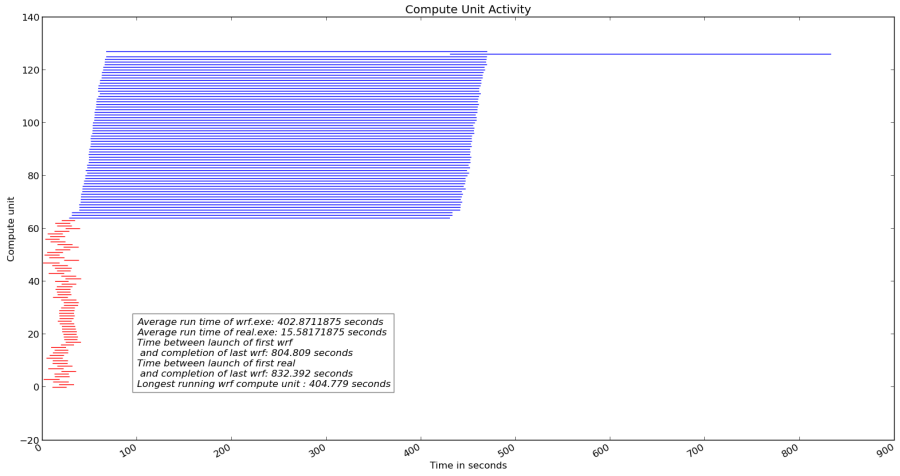

Figure 5.4: Compute unit activity time-line(64 Ensembles with each `wrf.exe` using 32 cores for smaller data set). This graph shows how the compute units become active on a time line and when they finish their execution. The Red lines are the real.exe executing on 16 cores per ensemble and the blue lines are `wrf.exe` compute units executing in succession on 64 cores per ensemble.

we fix the ensemble size at 1,2,8,32 and 64. For each of the these fixed ensemble sizes, we plot total time taken when the number of cores per ensemble member is varied from 16 through 256 and capture the strong scaling characteristics. We have not performed exhaustive runs (like we did for the small input data set) for the larger data sets as we ran out of allocation and these jobs typically run for about 5 hours on each core, but we made enough runs (higher end and lower end consisting of 64 ensemble members and 128 cores each and 1 ensemble with 16 cores respectively) and took enough data points to deduce the behavior of ensemble based approach for running `wrf.exe` for large input data sets. Figure 5.6 captures the strong scaling characteristics for the large input data set and when we compare this with figure 5.1 we see clear symmetry. Armed with this information we next plot the total time taken while we fix the number of cores used per ensembles and run more number of ensembles (from 1 -64). This data is plotted in figure 5.7. Here it becomes very clear that the time taken to run 32 ensembles is identical to the time taken by 1 ensemble member as long as the number of cores used per ensemble member is same; the behavior is identical to what we observe for the

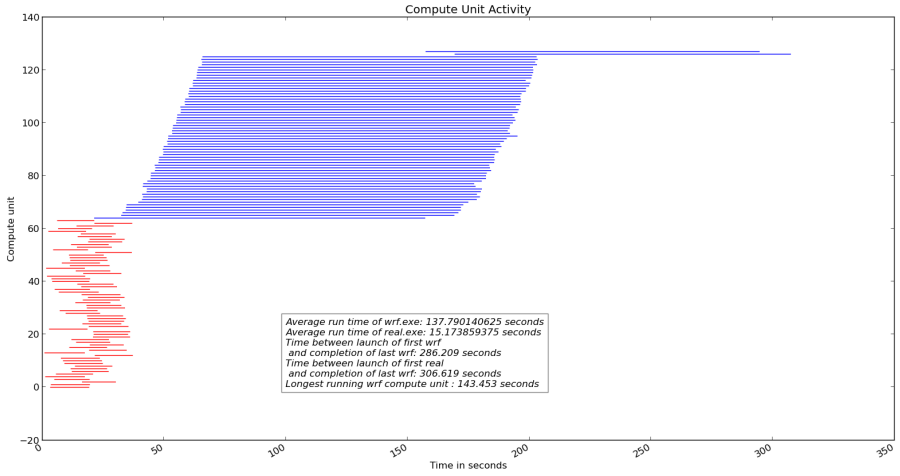

Figure 5.5: Compute unit activity time-line(64 Ensembles with each `wrf.exe` using 128 cores for smaller data set). This graph shows how the compute units become active on a time line and when they finish their execution. The Red lines are the real.exe executing on 16 cores per ensemble and the blue lines are `wrf.exe` compute units executing in succession on 64 cores per ensemble.

small data set. 32 ensemble members using 256 cores per ensemble member (a total of 8192 cores) takes the same time (within 5 percent error margin) as a single ensemble member running on 256 cores. The figures 5.2 and 5.7 are very symmetrical. From all this we can now say that the weather researchers can run at least 32 ensemble members using **RADICAL-Pilot** and expect a total time to completion that they would for a single ensemble member without having to worry about large or small input data sets.

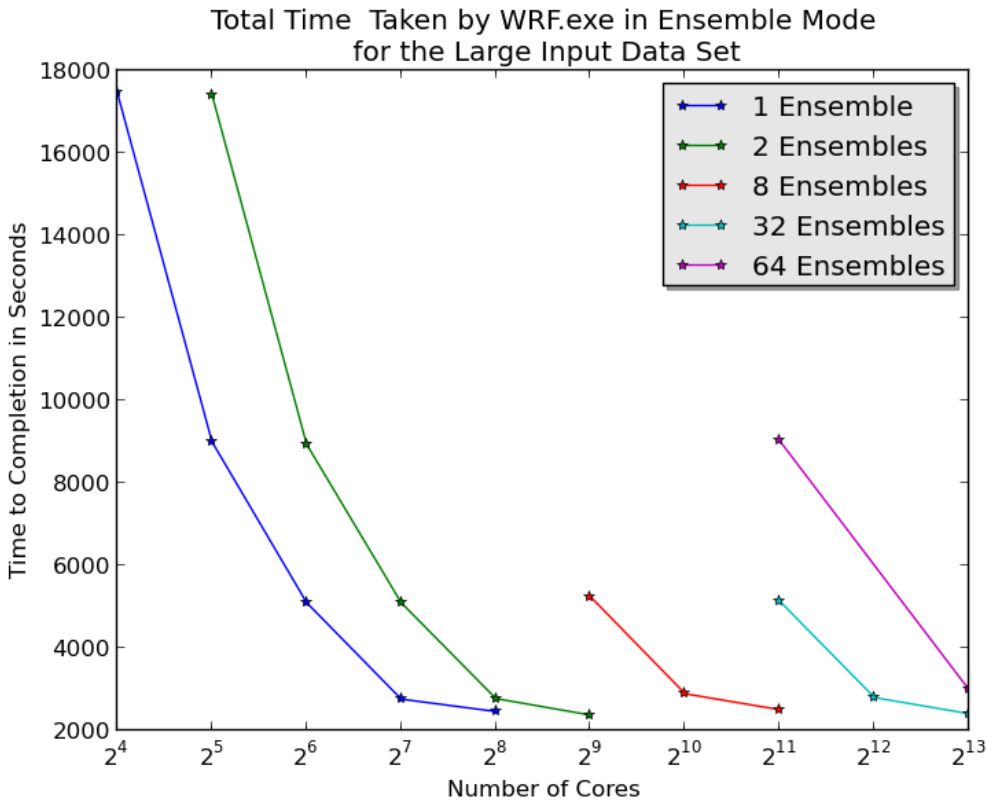

Figure 5.6: The figure depicts the strong scaling characteristics for the large input data set by plotting number of cores used vs total time to completion in linear scale. Here the number of ensemble members are fixed at 1,2,4,8,16,32 and 64; number of cores per ensemble member is varied from 16 through 256. The strong scaling behavior of the **WRF** application kernel for the large input data sets is very good.

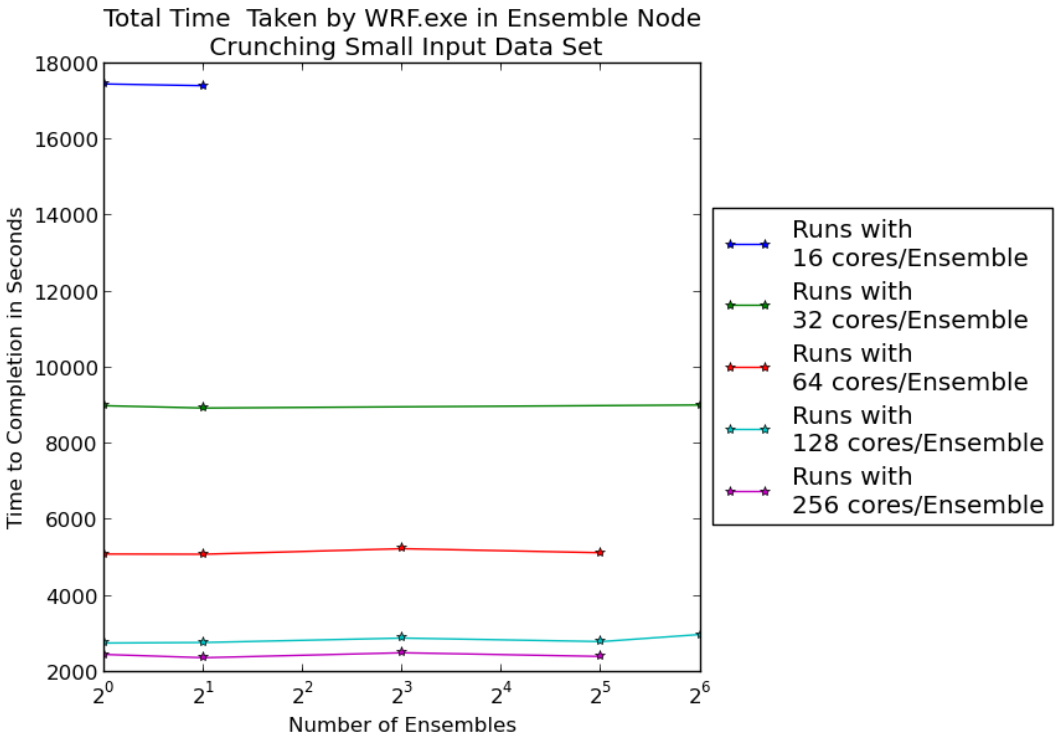

Figure 5.7: The graphs shows the total time taken to run **WRF** ensembles while fixing the number of cores per ensemble member for the large input set. As we increase the number of ensemble members keeping the number of cores per ensemble member fixed we observe excellent weak scaling. When we increase the number of ensemble members from 1 through 32 the total time consumed remains constant as long as the number of cores per ensemble member is fixed. This behavior is identical to that seen for small input data set. By symmetry this implies that **WRF** application kernels using **RADICAL-Pilot** scale well for both small as well as large input data sets.

Chapter 6

Conclusion and Future Work

The weather researchers use complex workflows using Distributed Computing Infrastructure (DCI) to generate forecast data. The heterogeneous nature of these distributed computing resources require the use of a local job-scheduler which becomes impeding to the researchers, as they have to invest significant amount of time on getting to know the underlying hardware resources which could otherwise be invested on their research. During the course of this dissertation we have developed **Pilot-Job** based tools which decouple work-load submission and resource allocation there by giving the weather scientists the flexibility work with a multitude of heterogeneous resources without having to really worry about how to manage them. Another important limitation of most local job-schedulers is that they have no smart way of executing a series of executables and moving data as these steps can be very specific to the application. Hence, job-schedulers are built to cater to the bare essentials of resources allocation as per user request. We have successfully designed, developed and tested tools that encapsulate the (WRF) workflow as an execution pattern that automates much of the **WRF** workflow. While developing these tools we have ensured that their migration from the traditional scripts is seamless and very easy for the weather researchers to adopt. The **Pilot based frameworks for WRF** developed as a part of this dissertation can easily be run on multiple distributed computing resources with minor changes and accomplish location independent job execution, data movement, placement and processing. During the course of our experiments we observe that streamlining the complex workflows by the *Pilot-Job* based tools reduces their overall time to completion. The tools devloped can run an Ensemble of tasks bearing the capability to run on multiple heterogeneous distributed computing resources. Our experiments and analysis show that the tools

developed exhibit very good, strong and weak scaling characteristics. These results not only portend that the weather researchers will save a lot of time that they can invest in their research instead of getting comfortable with the computing resource which could be tedious but also can potentially impact the way weather researchers are submitting traditional **WRF** jobs to the DCIs. The **Pilot based frameworks for WRF** gives the scientists a powerful weapon in their arsenal that can exploit the combined power of various heterogeneous DCIs which would have otherwise been difficult to harness owing to interoperability issues.

RADICAL-Pilot has successfully enabled many applications on all of three these systems, but at the time of this dissertation **WRF and WRF-Hydro (Coupled)** were compiled only on Yellowstone High Performance Computing Resource. Work is in currently in progress to compile and have **WRF** run on both Stampede and SuperMUC[18]. We anticipate the emergence of similar strong and weak scaling characteristics on Stampede and SuperMUC. The foundation for There is immense scope for future work in breaking down the workflow and running different parts of workflow on different Distributed Computing Infrastructures. A glimpse of the future(definitely not very distant) would be a weather researcher initiating a script from his or her personal laptop and different parts of the workflow simultaneously kicked off on different computing resources and data being moved seamlessly between these heterogeneous systems there by completely exploiting their combined capability. The weather researchers will not have to worry about how all this is accomplished while they wait for a *units successfully completed* message they can focus their efforts into their own research instead of nitty-gritty of the computing world.

References

[1] http://www2.cisl.ucar.edu/resources/yellowstone, December 2014.

[2] https://radical-cybertools.github.io/img/radical-saga.png, April 2014.

[3] http://radicalpilot.readthedocs.org/en/latest/intro.html, April 2014.

[4] https://www.tacc.utexas.edu/stampede/, December 2014.

[5] http://www.wrf-model.org/index.php, August 2014.

[6] Wikipedia. Weather research and forecasting model — wikipedia, the free encyclopedia, 2014. [Online; accessed 5-December-2014].

[7] http://www2.mmm.ucar.edu/wrf/users/tutorial/201407/Monday/2_dudhia_overview.pdf, August 2014.

[8] http://www.dtcenter.org/wrf-nmm/users/docs/user_guide/V3/users_guide_nmm_chap3.pdf, August 2014.

[9] http://www.ral.ucar.edu/projects/wrf_hydro/, August 2014.

[10] http://saga-project.github.io/BigJob/, April 2014.

[11] Andre Luckow, Lukas Lacinski, and Shantenu Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 135–144, 2010.

[12] Jakub T Moscicki. Diane-distributed analysis environment for grid-enabled simulation and analysis of physics data. In *Nuclear Science Symposium Conference Record, 2003 IEEE*, volume 3, pages 1617–1620. IEEE, 2003.

[13] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 43. ACM, 2007.

[14] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, et al. Dirac pilot framework and the dirac workload management system. In *Journal of Physics: Conference Series*, volume 219, page 062049. IOP Publishing, 2010.

[15] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[16] http://radicalpilot.readthedocs.org/en/latest/examples/gettingstarted.html#, August 2014.

[17] https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance, December 2014.

[18] http://www.lrz.de/services/compute/supermuc/, December 2014.