University Politehnica of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Diploma Thesis

# Simulation and Model Coupling in Meteorology Case Study: WRF optimization

by

**Valentin Marcu**

Supervisor: Associate Prof. Dr. Eng. Emil Slușanschi,

Bucharest, July 2012

# Contents

# List of Figures

# Abstract

Through a collaborative partnership, principally among National Center for Athmospheric Research (U.S.A.), National Oceanic and Athmospheric Administration (U.S.A) and universities, the international modelling community is developing a next-generation mesoscale model, known as Weather Research and Forecasting (WRF) model. The goals of the WRF project are to develop an advanced mesoscale forecast and data assimilation system and to accelerate research advances into operations [JM].

The model is intended to improve forecast accuracy accross scales ranging from cloud to synoptic, with priority emphasis on horizontal grid resolutions of 1-10 kilometers. Numerous real-time forecasting experiments are being conducted to evaluate WRF performance in a variety of forecast applications [YHK08].

This thesis describes the improvements we have made to the WRF model during the past year, using the computational resources provided by the National Center for Information Technology (NCIT) Cluster, located at "Politehnica" University of Bucharest.

After a brief presentation of the WRF architecture and functionality, we will thoroughly explain the modifications brought to one of the most important modules of this model: the MPI interface. Results of these optimizations are also provided, in a comparative manner (initial versus present version performance), along with short comments and conclusions.

Obtaining faster weather forecasts with WRF facilitates the coupling (use of WRF outputs as inputs) with a particle dispersion model called FLEXPART (which is discussed in a related thesis [Con12]), on larger domains and by using approximately the same hardware and software resources.

Chapter 1

# WRF Overview

The Weather Research and Forecasting (WRF) Model is a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. It features multiple dynamical cores, a 3-dimensional variational (3DVAR) data assimilation system, and a software architecture allowing for computational parallelism and system extensibility. WRF is suitable for a broad spectrum of applications across scales ranging from meters to thousands of kilometers [JM].

WRF allows researchers the ability to conduct simulations reflecting either real data or idealized configurations. WRF provides operational forecasting a model that is flexible and efficient computationally, while offering the advances in physics, numerics, and data assimilation contributed by the research community [?].

The WRF model receives contributions of new methods and features from both the operational and research communities.In addition, through contributions to the WRF repository, a wide range of users from various arenas have the ability to share their new developments with the community, creating the potential for the most qualified techniques to advance into operations. The WRF system also provides a common framework that simplifies the implementation and testing of these new methods and capabilities [JM].

The current WRF model has two dynamic cores; the Advanced Research WRF (ARW) and the Nonhydrostatic Mesoscale Model (NMM). In addition to dynamic core, several other elements of the WRF model are user configurable. These elements include the domain (i.e., location, resolution, and nesting), physical parameterizations, analysis nudging, and observation nudging, to name a few. Ultimately, the resulting model forecasts will vary depending on how the model is configured by the user [NCA11].

## 1.1 Software Architecture

The WRF prototype employs a layered software architecture that promotes modularity, portability, and software reuse. Information hiding and abstraction are employed so that parallelism, data management, and other issues are dealt with at specific levels of the hierarchy and transparent to other layers [YHK08].

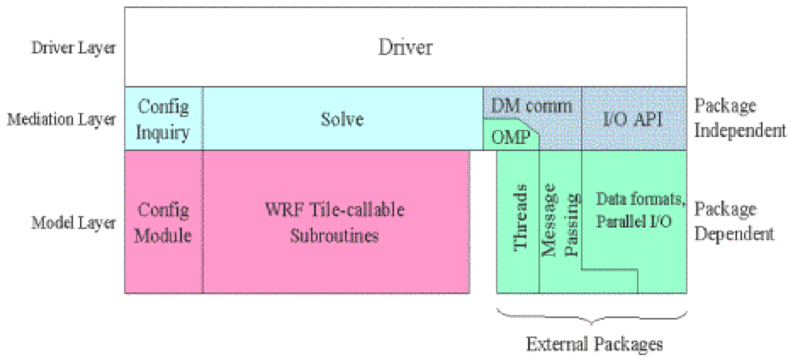


Figure 1.1: Software architecture schematic

There are three distinct model layers:

- The Driver Layer handles run-time allocation and parallel decomposition of model domain data structures. It is responsible with organization, management, interaction and control over nested domains, including the main time loop in the model and high level interfaces to I/O operations on model domains [JM].
- The Mediation Layer encompasses one time-step of a particular dynamical core on a single model domain. The solve routine for the dynamical core contains the complete set of calls to Model Layer routines as well as invocation of interprocessor communication (halo updates, parallel transposes, etc.) and multithreading. The current WRF implementation uses the RSL communication library, which, in turn, uses the Message Passing Interface (MPI) communication package. Shared-memory parallelism over tiles a second level of domain decomposition within distributed memory patches is also specified in the solve routines using OpenMP [YHK08].
- The Model Layer comprises the actual computational routines that make up the model: advection, diffusion, physical parameterizations, and so forth. Model layer routines that have data dependencies rely on the mediation layer to perform the necessary interprocessor communication prior to their being called [JM].

APIs to external packages are also part of the WRF software framework. These allow WRF to use different packages for self-describing data formats, model coupling toolkits and libraries, and libraries for interprocessor-communication by simply adapting the external package to the interface.

## 1.2 Main dataflow

The WRF Preprocessing System (WPS) is used primarily for real-data simulations. Its functions include: degribbing and interpolating meteorological data from another model to this simulation domain (ungrib.exe), defining simulation domains (geogrid.exe) and interpolating terrestrial data (such as terrain, landuse, and soil types) to the simulation domain (metgrid.exe) [?].

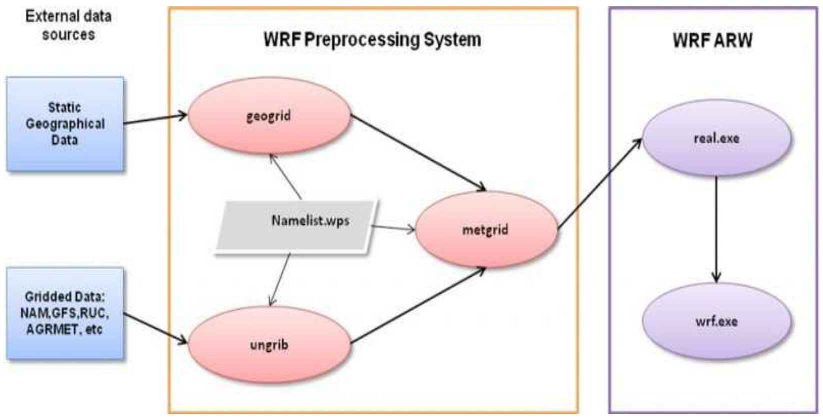


Figure 1.2: WRF flowchart

The Advanced Research WRF (ARW) solver is the key component of the modeling system, which is composed of several initialization programs for idealized, and real-data simulations (real.exe), and the numerical integration program (wrf.exe).

The key features of the WRF model include [NCA11]:

- Regional and global applications
- Fully compressible nonhydrostatic equations with hydrostatic option
- Runge-Kutta 2nd and 3rd order time integration options
- Lateral boundary conditions (for real cases: specified with relaxation zone)
- Two-way nesting with multiple nests and nest levels

The output (generated by wrf.exe) is in NetCDF format, which is widely used to store meteorological data, being nowadays compatible with almost all numerical weather forecast models.

## 1.3 Nesting domains

Nesting is a form of mesh refinement that allows costly higher resolution computation to be focused over a region of interest.

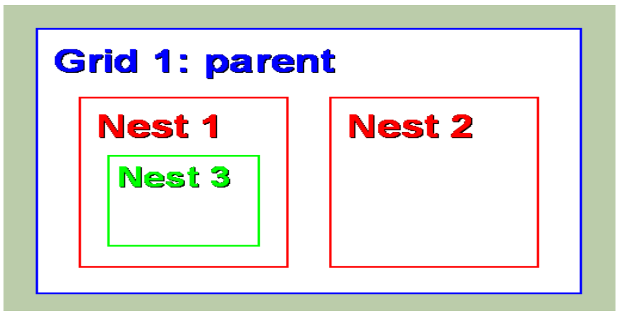


Figure 1.3: Nesting domains principle

Nests in WRF are nonrotated and aligned so that parent mesh points are coincident with a point on the underlying nest, which eliminates the need for more complicated generalized regridding calculations. Nest configurations are specified at run-time through the namelist [YHK08].

A two-way nested run is a run where multiple domains at different grid resolutions are run simultaneously and communicate with each other: The coarser domain provides boundary values for the nest, and the nest feeds its calculation back to the coarser domain. The model can handle multiple domains at the same nest level (no overlapping nest), and multiple nest levels (telescoping) [NCA11].

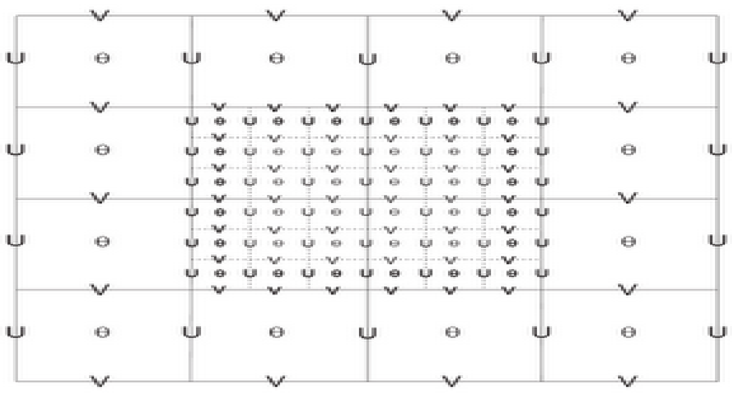


Figure 1.4: Nest versus parent domain resolution

WRF generates separate output files for each domain for which the simulation is set.

## 1.4 Domain decomposition

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

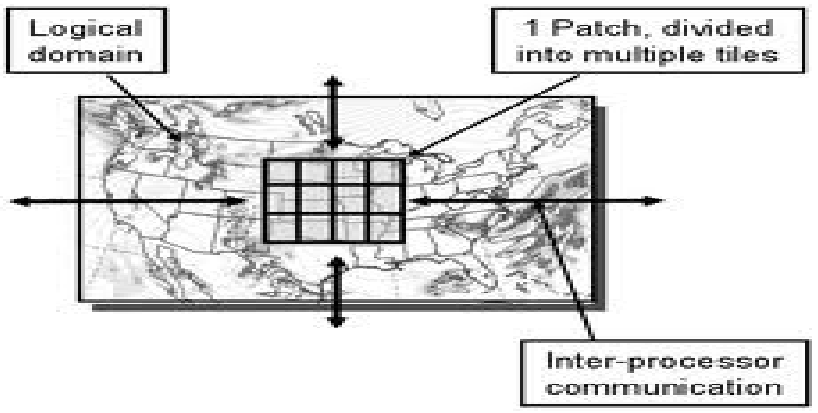

Logical domain
1 Patch, divided into multiple tiles
Inter-processor communication

Figure 1.5: Domain decomposition

A flexible approach for parallelism is achieved through a two-level decomposition in which the model domain may be subdivided into patches that are assigned to distributed-memory nodes and then may be further subdivided into tiles that are allocated to shared-memory processors within a node.

This approach addresses all current models for parallelism (single processor, shared memory, distributed memory, and hybrid) and also provides adaptivity with respect to processor type: tiles may be sized and shaped for cache blocking or to preserve maximum vector length.

Model layer subroutines are required to be tile callable, that is, callable for an arbitrarily sized and shaped subdomain. All data must be passed through the argument list (state data) or defined locally within the subroutine [JM].

Thus, the WRF software architecture and two-level decomposition strategy provides a flexible, modular approach to performance portability across a range of different platforms, as well as promoting software reuse. It will facilitate use of other framework packages at the WRF driver layer as well as the reverse, the integration of other models at the model layer within the WRF framework [YHK08].

## 1.5 WRF implementation in Romania

The National Meteorological Administration uses WRF coupled with a regional numeric forecast model called COSMO (Consortium for Small Scale Modelling), which has smaller domain resolutions.

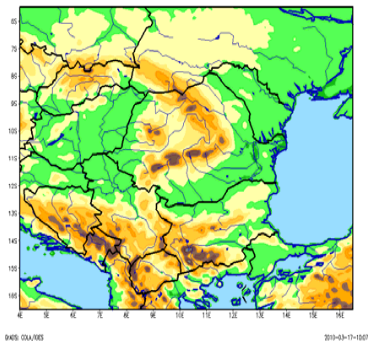


Figure 1.6: COSMO 7km

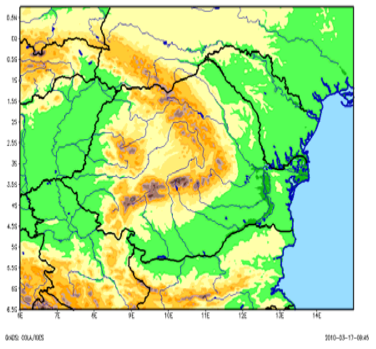


Figure 1.7: COSMO 2.8km

Thus, the output generated by COSMO using a 7km resolution grid is processed by WRF at higher spatial resolutions (3km and 1km), in order to obtain much more accurate forecasts [BM11].

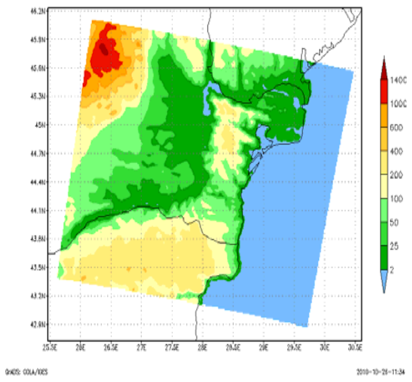


Figure 1.8: WRF 3km

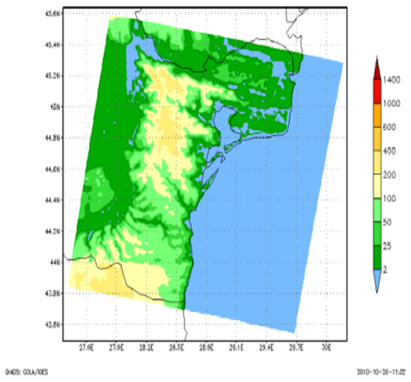


Figure 1.9: WRF 1km

Unfortunately, such high resolution forecasts require massive computational resources and a considerable amount of time in order to complete. Therefore, in the present only small domains are used for long-term simulations [BM11].

The following chapters describe how we managed to obtain faster runs with the same accuracy, permitting future enlargement of the present domains at the same spacial resolutions. We will focus on the optimizations we brought to the MPI transfers between processes and how this improved the overall performance of the model, as well as the computational resources used during the simulations.

Chapter 2

# WRF Optimization

## 2.1 Distributed memory communications

As stated in the previous chapter, hybrid (distributed and shared memory) implementations of WRF require a logical decomposition of the processed domains into rectangular patches, which are assigned to the available MPI processes.

Each MPI process may in turn decompose its patch into smaller tiles, that are processed by a predefined number of OpenMP threads, sharing the local memory of the process. The numerical integration schemes, along with other algorithms

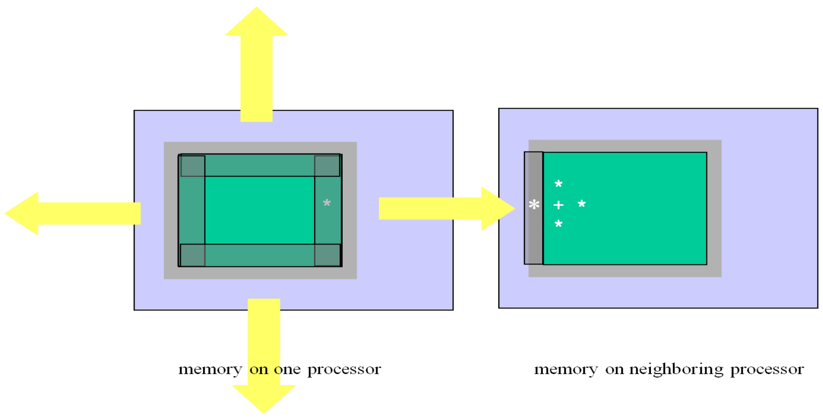


Figure 2.1: Halo update schematic

used by the model, require periodic updates of the meteorological variables stored in neighbouring areas of each domain. While within a patch this update is made directly (all the tiles are visible for all the threads), any two adjacent patches need to exchange messages containing the update information, using various MPI routines, grouped within the RSL (Runtime System Library) interface.

Using MPI_Cart routines (e.g. MPI_Cart_create, MPI_Cart_shift etc), all the MPI processes are logically organized into a two-dimensional array, and each process knows its (maximum) 4 neighbours: (ym,yp) for Y direction and (xm, xp) for X direction. In order to simultaneously send and receive messages to and from all its neighbours, each process allocates up to 8 buffers (4 buffers for receive and 4 buffers for send).

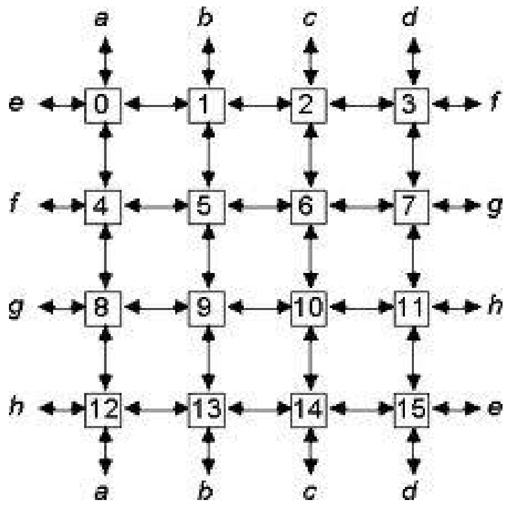


Figure 2.2: The "grid" of processes

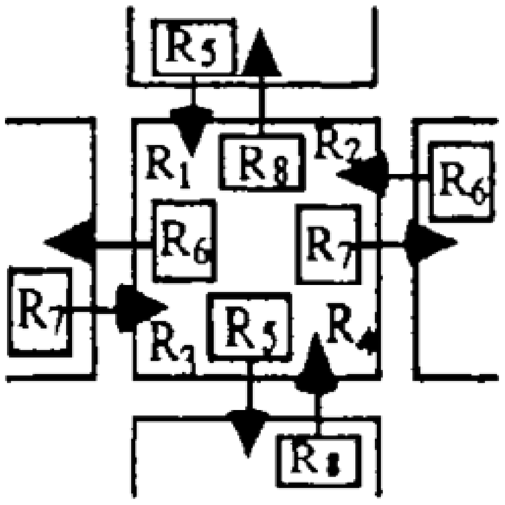
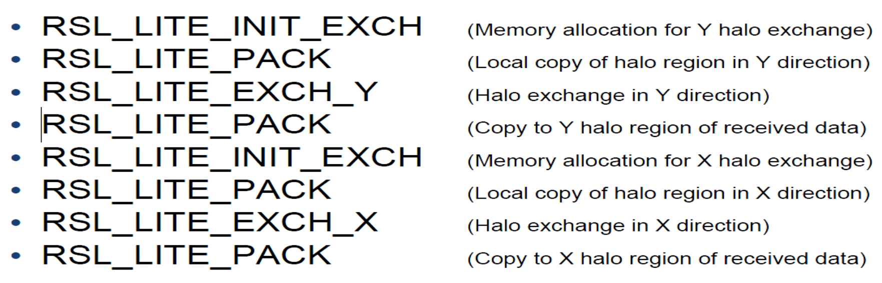


Figure 2.3: Use of send/recv buffers

The data which is exchanged is usually called halo data (because it is located on the borders of each patch) or ghost area data (because the values stored in it change very often). The buffers are temporary and dynamically allocated memory areas, without a correspondent in the domain configuration. Their dimensions usually vary between 100 kilobytes and 1 megabyte.

## 2.2 Initial halo exchange



| | |
|---|---|
| • RSL_LITE_INIT_EXCH | (Memory allocation for Y halo exchange) |
| • RSL_LITE_PACK | (Local copy of halo region in Y direction) |
| • RSL_LITE_EXCH_Y | (Halo exchange in Y direction) |
| • RSL_LITE_PACK | (Copy to Y halo region of received data) |
| • RSL_LITE_INIT_EXCH | (Memory allocation for X halo exchange) |
| • RSL_LITE_PACK | (Local copy of halo region in X direction) |
| • RSL_LITE_EXCH_X | (Halo exchange in X direction) |
| • RSL_LITE_PACK | (Copy to X halo region of received data) |

Figure 2.4: Initial steps of halo exchange

### 2.2.1 Static variables

Each process uses a considerable number of static variables in order to keep information during multiple MPI routines and/or transfers.

Most of them indicate transfer dimensions (e.g. sendw_m, sendw_p, recvw_m, recvw_p), identificators of the memory areas affected by the halo exchange (e.g. ims, ime, jms, jme) and MPI variables (e.g. MPI_Status, MPI_Request).

```
// current transfer dimensions (send/recv , x/y)
static int yp_curs , ym_curs , xp_curs , xm_curs ;
static int yp_curs_recv , ym_curs_recv , xp_curs_recv , xm_curs_recv ;
...
static MPI_Request yp_recv , ym_recv , yp_send , ym_send ;
static MPI_Request xp_recv , xm_recv , xp_send , xm_send ;
```

### 2.2.2 RSL_LITE_INIT_EXCH

This function will allocate memory which is needed for storing ghost area values before halo exchange. The size of memory is dependent on the number of variables and its ghost area size. If there is sufficient memory already allocated, no action will be occurred. If more memory is required, new memory allocation will be occurred [Kim06].

```
    MPI_Cart_shift ( *comm0 , 0 , 1 , &ym , &yp ) ;
    if ( yp != MPI_PROC_NULL ) {
       buffer_for_proc ( yp , nbytes_y_recv , RSL_RECVBUF ) ;
       buffer_for_proc ( yp , nbytes , RSL_SENDBUF ) ;
    }
    if ( ym != MPI_PROC_NULL ) {
       buffer_for_proc ( ym , nbytes_y_recv , RSL_RECVBUF ) ;
       buffer_for_proc ( ym , nbytes , RSL_SENDBUF ) ;
    }
// (same for xp and xm)
  ........
yp_curs = 0 ; ym_curs = 0 ; xp_curs = 0 ; xm_curs = 0 ;
yp_curs_recv = nbytes_y_recv ; ym_curs_recv = nbytes_y_recv ;
xp_curs_recv = nbytes_x_recv ; xm_curs_recv = nbytes_x_recv ;
```

Each send/receive buffer will have ghost area values between domains, which will be determined from the number of processes and X/Y domain decomposition.

### 2.2.3 RSL_LITE_PACK

If there is a data which should be sent, the ghost area data will be copied to local memories which was allocated by RSL_LITE_INIT_EXCH routine. The data will be packed into one contiguous memory region so that one MPI send/receive call could be used for exchanging halo data [Kim06].

If there is a data which was received, the data will be copied to the original ghost area position depending on the domain decomposition [Kim06].

```
    MPI_Cart_shift( *comm0 , 0, 1, &ym, &yp ) ;
    nbytes = buffer_size_for_proc( yp, da_buf ) ;
 p = buffer_for_proc( yp , 0 , da_buf ) ;
 .........
if ( sendwp > 0 )
 F_PACK_INT ( buf, p+yp_curs, imemord, &js, &je,&ks,
 &ke, &is, &ie,&jms,&jme,&kms,&kme,&ims,&ime, &wcount );
 ...........
if ( recvwp > 0 )
 F_UNPACK_INT ( p+yp_curs, buf, imemord, &js, &je, &ks,
 &ke, &is, &ie, &jms,&jme,&kms,&kme,&ims,&ime, &wcount );
 yp_curs += wcount*typesize ;
 ............... (similar code for ym, xp, and xm)
```

### 2.2.4 RSL_LITE_EXCH_Y(X)

The packed ghost area data will be exchanged between MPI process. The number of MPI send/receive calls per MPI process is dependent on the domain decomposition [Kim06].

```
    MPI_Cart_shift( *comm0, 0, 1, &ym, &yp ) ;
if ( yp != MPI_PROC_NULL && *recvw_p > 0 )
 MPI_Irecv (buffer_for_proc(yp,yp_curs_recv,RSL_RECVBUF),
        yp_curs_recv, MPI_CHAR, yp, me, comm, &yp_recv ) ;
if ( ym != MPI_PROC_NULL && *recvw_m > 0 )
 MPI_Irecv ( buffer_for_proc(ym,ym_curs_recv,RSL_RECVBUF),
        ym_curs_recv, MPI_CHAR, ym, me, comm, &ym_recv ) ;

if ( yp != MPI_PROC_NULL && *sendw_p > 0 )
  MPI_Isend ( buffer_for_proc( yp, 0, RSL_SENDBUF ),
         yp_curs, MPI_CHAR, yp, yp, comm, &yp_send ) ;
if ( ym != MPI_PROC_NULL && *sendw_m > 0 )
  MPI_Isend ( buffer_for_proc( ym, 0, RSL_SENDBUF ),
         ym_curs, MPI_CHAR, ym, ym, comm, &ym_send ) ;
```

Non-blocking MPI_ISEND/MPI_IRECV call with MPI_WAIT sequence is used for this communication. This will require only weak synchronization between MPI processes which do halo exchange [Kim06].

```
if ( yp != MPI_PROC_NULL && *recvw_p > 0 )
  MPI_Wait( &yp_recv , &stat );
if ( ym != MPI_PROC_NULL && *recvw_m > 0 )
  MPI_Wait( &ym_recv , &stat );
if ( yp != MPI_PROC_NULL && *sendw_p > 0 )
  MPI_Wait( &yp_send , &stat );
if ( ym != MPI_PROC_NULL && *sendw_m > 0 )
  MPI_Wait( &ym_send , &stat );

yp_curs = 0 ; ym_curs = 0 ; xp_curs = 0 ; xm_curs = 0 ;
yp_curs_recv = 0 ; ym_curs_recv = 0 ;
xp_curs_recv = 0 ; xm_curs_recv = 0 ;
```

### 2.2.5 Disadvantages of the initial implementation

- given the large dimensions of transfers, no fragment of data can be accessed until the whole transfer has finished; the amount of time between reveive and access becomes significant
- although non-blocking communication routines are used, MPI_Wait blocks the workflow immediately after the transfers begin; thus, the behavior is similar to an MPI_Barrier (busy-waiting)
- all transfers to/from x-neighbours are called and waited to complete after the completion of all the transfers to/from y-neighbours, despite the sufficient number of local buffers for each process and the total independence of x- and y-transfers
- the use of the interconnection network bandwith is severely unbalanced: periods with no transfers alternate with periods in which all processes transfer between each other large amounts of data, often causing extra delays during MPI_Wait routines

## 2.3 Optimized halo exchange

- RSL_LITE_INIT_EXCH (Y)
- RSL_LITE_PACK (Y)
- RSL_LITE_EXCH_Y (here we only begin y-transfers)
- RSL_LITE_INIT_EXCH (X)
- RSL_LITE_PACK (X)
- RSL_LITE_EXCH_X (here we only begin x-transfers)
- RSL_LITE_UNPACK (Y) (only here we call MPI_Wait for y-transfers)
- RSL_LITE_UNPACK (X) (only here we call MPI_Wait for x-transfers)

### 2.3.1 New static variables

First of all, we defined several global variables (N and NY) used to split the MPI transfers into smaller and simultaneous ones. This change is applied only to those transfers large enough (with a dimension greater than the MIN_LOAD constant).

```
#define N 8  // each y-transfer is replaced by N transfers
#define NX 8 // similar to N, but for x-transfers
//minimum initial y-transfer dimension
#define MIN_LOAD_Y 10000
//minimum initial x-transfer dimension
#define MIN_LOAD_X 10000
```

We then separated completely the variables used during y-transfers from those used during x-transfers.

```
static MPI_Request yp_recv[N],ym_recv[N],yp_send[N],...;
static MPI_Request xp_recv[NX],xm_recv[NX],xp_send[NX],...;
static int last_yp=0, last_ym=0, last_xp=0, last_xm=0 ...;

//w_yp_recv[i]=1 if the last i-th receive from yp finished
static int w_yp_send[N] = {1}, w_ym_send[N] = {1} ...;
static int w_yp_recv[N] = {1}, w_ym_recv[N] = {1} ...;
```

Other variables were used in order to facilitate the new halo exchange suite (which is generated in the gen_comms.c file). Basically, these new variables have the same functionality as the original ones, but their values are used only when calling x-transfer routines.

### 2.3.2 RSL_LITE_PACK

A transfer contains data from multiple sources in the sender's grid, and for each of these sources the packing and unpacking require separate calls of RSL_LITE_PACK function.

The modified implementation of this function permits the access to a portion of a send/receive buffer without waiting the transfer of the whole buffer. This is achieved by calling MPI_Wait only for the sub-transfer that contains the requested data.

For example, if a process wants to unpack the first 100 kbytes of a buffer and the dimension of the current transfer regarding that buffer is 1 Mbyte, assuming that there are N=8 sub-transfers started, MPI_Wait will be called only for the first sub-transfer (which has 125 kbytes, thus including the requested data).

```
int send = last_yp/N, next, first=0, last=0;
if(send > 0) {
  first = 0;
  last = 1+(yp_curs + wcount*typesize)/send;
}
for(next = first; next <= last; next++) {
 if(w_yp_send[next] != 1)
    {
        w_yp_send[next] = 1;
        if(yp_send[next]) MPI_Wait(&yp_send[next], &stat);
    }
  F_PACK_INT ( ....)
}
// ............. (similar code for ym, xp and xm)
```

Something similar happens with the send buffers: MPI_Wait is called (to insure the completion of the previos send transfer) on a portion of the send buffer only when that region is needed for the next send operation.

### 2.3.3 RSL_LITE_EXCH_Y(X)

First of all, the dimensions of each transfer (which is different from the buffer size) are "saved" into 8 static variables, for further use when unpacking the received data (last_yp_recv etc) or when packing data for the next send transfer (last_yp etc).

```
last_yp = yp_curs; last_ym = ym_curs;  //for send routines
last_yp_recv = yp_curs_recv; last_ym_recv = ym_curs_recv;
....
last_xp = xp_curs; last_xm = xm_curs;
last_xp_recv = xp_curs_recv; last_xm_recv = xm_curs_recv;
```

Next, we check whether the transfer is small enough (by comparing its dimension to a predefined constant - MIN_LOAD_Y/X), in which case we use the original transfer calls.

If the transfer is large enough, we split it into N (for y-axis) or NX (for x-axis) "sub-transfers", which address separate portions of the send or receive buffer. Since the MPI_Wait is no longer called here, each transfer suite uses static int vectors (w_yp_recv etc), which are used to "remember" the state of each current sub-transfer (0 if it has finished and 1 otherwise).

```
buf1 = buffer_for_proc( yp, 0, RSL_RECVBUF );
if(yp_curs_recv < MIN_LOAD_Y)
  MPI_Irecv(buf1, yp_curs_recv, MPI_CHAR,
                    yp, me, comm, &yp_recv[0]);
else {
 for(i=0; i < N-1; i++)
   MPI_Irecv ( buf1 + i*yp_curs_recv/N, yp_curs_recv/N,
              MPI_CHAR, yp, me+i*100, comm, &yp_recv[i] ) ;
    MPI_Irecv ( buf1 + (N-1)*yp_curs_recv/N,
        yp_curs_recv - (N-1)*yp_curs_recv/N, MPI_CHAR,
        yp, me+(N-1)*100, comm, &yp_recv[N-1] ) ;
    for(i=0; i < N; i++)
        w_yp_recv[i] = 0; //the i-th sub-transfer has begun
} // ..... (similar code for ym, xp and xm)
```

Finally, if the transfer is small enough (and thus no sub-transfers were started), we wait here its completion.

```
if ( yp != MPI_PROC_NULL && yp_curs_recv < MIN_LOAD_Y)
     MPI_Wait( &yp_recv[0], &stat ) ;
if ( ym != MPI_PROC_NULL && ym_curs_recv < MIN_LOAD_Y)
     MPI_Wait( &ym_recv[0], &stat ) ;
if ( yp != MPI_PROC_NULL && yp_curs < MIN_LOAD_Y)
     MPI_Wait( &yp_send[0], &stat ) ;
if ( ym != MPI_PROC_NULL && ym_curs < MIN_LOAD_Y)
     MPI_Wait( &ym_send[0], &stat ) ;
// similar code in RSL_LITE_EXCH_X
```

### 2.3.4 RSL_LITE_FINISH_TRANSFERS

This function is called in the end of a halo-exchange (for both y- and x-axis), i.e. after several INIT-PACK-EXCH-UNPACK chain calls. Only the last pending sends are waited to complete.

```c
if(last_yp >= MIN_LOAD_Y) for(i=0; i<N; i++)
{
    if(w_yp_send[i] != 1)
    {
        w_yp_send[i] = 1;
        if(yp_send[i]) MPI_Wait(&yp_send[i], &stat);
    }
    if(w_ym_send[i] != 1)
    {
        w_ym_send[i] = 1;
        if(ym_send[i]) MPI_Wait(&ym_send[i], &stat);
    }
}
//...... (similar code for x-axis)
```

### 2.3.5 Advantages of the new halo exchange

- the amount of time between data receive and data access is significantly reduced
- the MPI_Wait routine is called asynchronously by each process and only when the data transfered is requested for access (the "barrier" effect is reduced)
- all data is exchanged on x-axis and y-axis simultaneously and safe (without any interference regarding the send and receive buffers)
- the bandwidth of the interconnection network is better used, especially when its value is comparable with or less than the transfer dimensions

Chapter 3

# Hardware and Software Configuration

## 3.1 The NCIT Cluster

The NCIT (National Center for Information Technology) of Politehnica University of Bucharest is a Beowulf cluster, composed of Intel Xeon and AMD Opteron processors, consisting of 456 cores in total. The cluster is currently used in research and teaching purposes by teachers and students [Pol10].

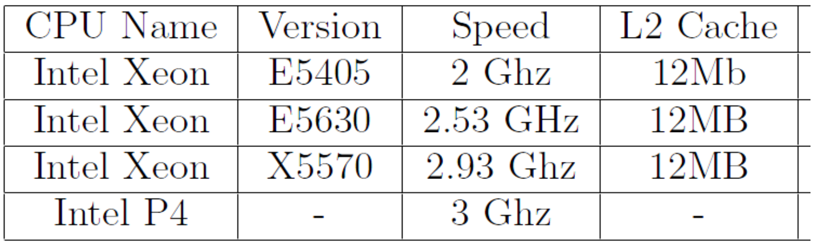
The Linux release used at the NCIT Cluster is a RHEL (Red Hat Enterprise Linux) clone called Scientific Linux (kernel version 2.6), co-developed by Fermi National Accelerator Laboratory and the European Organization for Nuclear Research (CERN) [Pol10].

| Model | Processor Type | Sockets/ Cores | Slots | Memory | Queue |
|---|---|---|---|---|---|
| IBM HS21, 28 nodes | Intel Xeon E5405, 2 GHz | 2/8 | 224 | 16 GByte | ibm-quad.q |
| IBM HS22, 4 nodes | Intel Xeon E5630, 2.53 GHz | 2/16 | 64 | 32 GByte | ibm-nehalem.q |
| IBM LS22, 14 nodes | Amd Opteron 2435, 2.6 GHz | 2/12 | 168 | 16 GByte | ibm-opteron.q |

Figure 3.1: NCIT cluster queues used in this project

The Intel Xeon Processor refers to many families of Intel's x86 multiprocessing CPUs for dualprocessor (DP) and multi-processor (MP) configuration on a single motherboard targeted at nonconsumer markets of server and workstation computers, and also at blade servers and embedded systems. The Xeon CPUs generally have more cache than their desktop counterparts in addition to multiprocessing capabilities [Pol10].

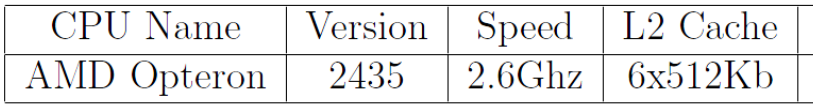
Our cluster is currently equipped with the Intel Xeon 5000 Processor sequence.



| CPU Name | Version | Speed | L2 Cache |
|---|---|---|---|
| Intel Xeon | E5405 | 2 Ghz | 12Mb |
| Intel Xeon | E5630 | 2.53 GHz | 12MB |
| Intel Xeon | X5570 | 2.93 Ghz | 12MB |
| Intel P4 | - | 3 Ghz | - |

Figure 3.2: Intel Xeon Processors

The Six-Core AMD Opteron processor-based servers deliver performance efficiency to handle real world workloads with a good energy efficiency. There is one IBM Chassis with 14 infiniband connected Opteron blades available and the rest are used in our Hyper-V virtualization platform [Pol10].



| CPU Name | Version | Speed | L2 Cache |
|---|---|---|---|
| AMD Opteron | 2435 | 2.6Ghz | 6x512Kb |

Figure 3.3: AMD Opteron Processor

## 3.2 NetCDF Library

All input, intermediate and output files that WRF uses are stored in NetCDF format.

NetCDF (Network Common Data Form) is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages. The netCDF libraries support a machine-independent format for representing scientific data. Together, the interfaces, libraries, and format support the creation, access, and sharing of scientific data [RRK07].

NetCDF data is:

- Self-Describing - A netCDF file includes information about the data it contains;

- Portable - A netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers;
- Scalable - A small subset of a large dataset may be accessed efficiently;
- Appendable - Data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure;
- Sharable - One writer and multiple readers may simultaneously access the same netCDF file;
- Archivable - Access to all earlier forms of netCDF data will be supported by current and future versions of the software [RRK07].

The current installed version of NetCDF is 4.1.3, compiled with openmpi-1.6 and gnu-4.6.3 compilers, which are used on the nodes that have Infiniband support: the ibm-opteron queue.

For the rest of the nodes (ibm-quad.q and ibm-nehalem.q), we installed a slightly older version of NetCDF (4.1.2), compiled with openmpi-1.5.3 and gcc-4.6.0, but only for testing purpose.

## 3.3 OpenMPI support

As stated above, we used two versions of OpenMPI: one with Infiniband support (v1.6, compiled with gnu-4.6.3 on ibm-opteron.q) and one without (v1.5.3, compiled with gcc-4.6.0 on ibm-quad.q and ibm-nehalem.q).

One of the scripts we used to run the WRF model using distributed memory (with or without shared memory enabled) is listed below.

```
#module load compilers/gcc-4.6.0
#module load mpi/openmpi-1.5.3_gcc-4.6.0
module load libraries/netcdf-4.1.3-openmpi-1.6-gcc-4.6.3
module load libraries/openmpi-1.6-gcc-4.6.3
module load compilers/gnu-4.6.3

export OMP_NUM_THREADS=1
export LD_LIBRARY_PATH=/..../netcdf_links/lib:$LD_LIBRARY_PATH
export INCLUDE=/..../netcdf_links/include:$INCLUDE

rm rsl.*
time mpirun -np $1 -npernode 12 ./wrf.exe
#time mpirun --mca btl ^openib --mca btl_tcp_if_include eth0
                    -np $1 -npernode 12 ./wrf.exe
```

## 3.4 SunStudio profiling

The SunStudio suite consists of two main tools: the Collector and the Performance Analyzer. These tools help to answer the following kinds of questions [Man07]:

- How much of the available resources does the program consume?
- Which functions or load objects are consuming the most resources?
- Which source lines and instructions are responsible for resource consumption?
- How did the program arrive at this point in the execution?
- Which resources are being consumed by a function or load object?

One of the scripts we used to collect information from a WRF run is listed below.

```
module load compilers/gcc-4.6.0
module load mpi/openmpi-1.5.3_gcc-4.6.0
module load compilers/sunstudio12.1

export OMP_NUM_THREADS=8
export LD_LIBRARY_PATH=.../netcdf_links/lib:$LD_LIBRARY_PATH
export INCLUDE=.../netcdf_links/include:$INCLUDE

export VT_UNIFY=0
export OUTPUT_ALL_RANKS=yes
export TRACE_ALL_TASKS=yes
export VT_MAX_FLUSHES=0
export VT_BUFFER_SIZE=2G
/opt/sun/sunstudio12.1/prod/bin/collect -p low -M CT8.2 -m off
         -S on  -A on -L none mpirun -mca btl ^openib
        -mca btl_tcp_if_include eth0 -np $1 --  wrf.exe
```

The current installed version of SunStudio is 12.1. It is available only on ibm-quad.q and ibm-nehalem.q.

# Chapter 4

# Case studies

This chapter describes the case studies in this project and the corresponding results. We ran simulations on all the 4 queues available (quad, quad-new, nehalem and opteron), including profilings with Sun Studio (on quad and nehalem).

The most important performance metrics that were measured were:

- total run time
- speedup
- efficiency
- impact of MPI_Wait

## 4.1 Results of Dobrogea Benchmark

The listing below contains some of the most important settings of the benchmark. Detailed measurements can be consulted in Appendix 1.

```
simulation_length   = 1 hour
max_dom             = 2, // number of domains
e_we                = 111,    181, // grid dimensions
e_sn                = 101,    193, // for each domain
//number of vertical layers
e_vert              = 35,    35,
num_metgrid_levels  = 21,
dx                  = 3000, 1000, // grid resolutions
dy                  = 3000, 1000, // for each domain
```

### 4.1.1 Results on ibm-quad.q

- OMP_NUM_THREADS = 8 (each blade has 1 MPI x 8 OMP)

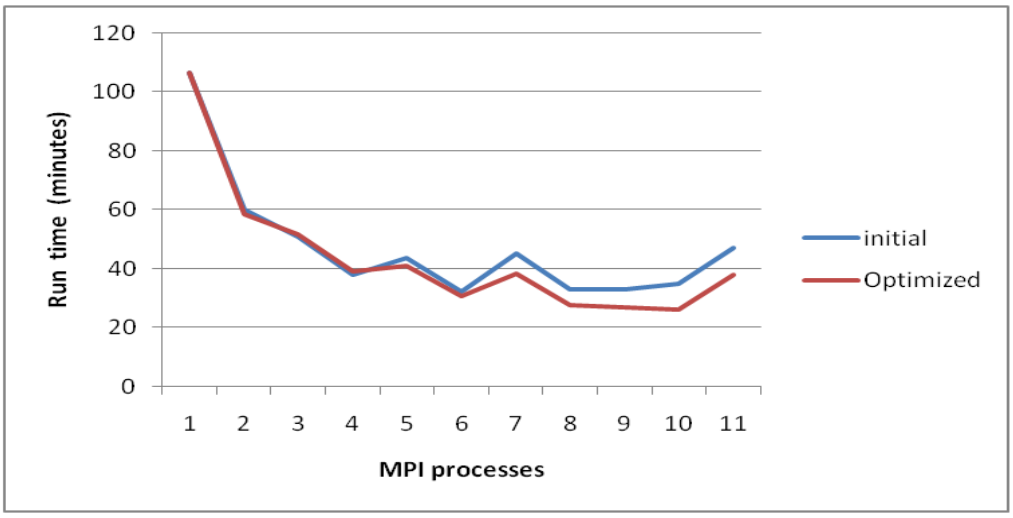


Figure 4.1: Comparative run times on ibm-quad.q

The difference between implementations becomes visible and even significant as the number of MPI processes increases. The maximum improvement is of approximately 20%.

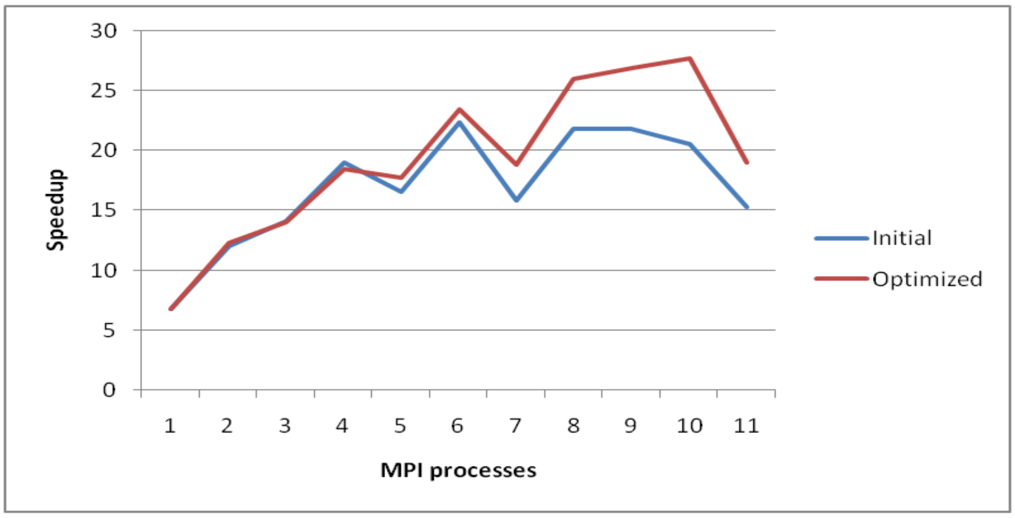


Figure 4.2: Comparative speedup on ibm-quad.q

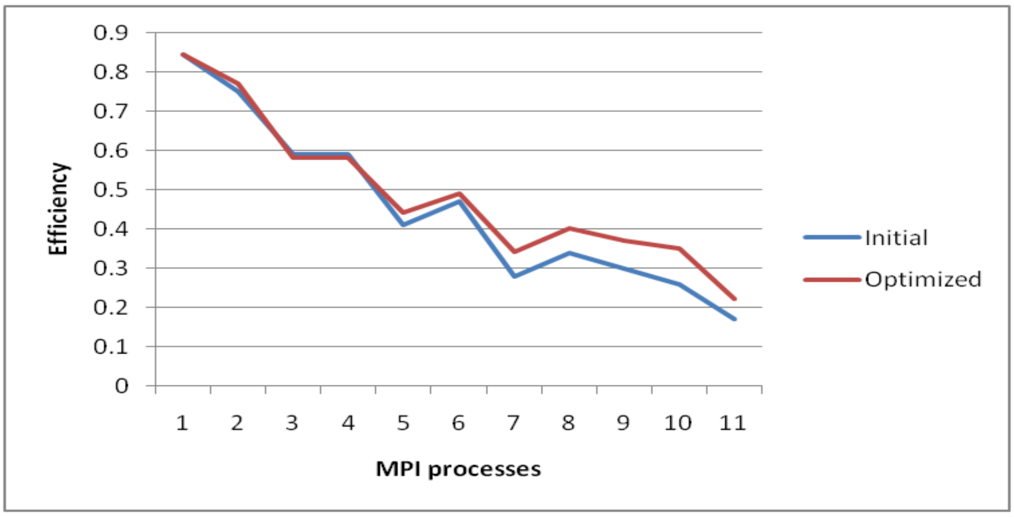


Figure 4.3: Comparative efficiency on ibm-quad.q

An important parameter of a destributed memory application is the point where a process spends more time waiting (calling MPI_Wait) than working (calling application-related routines). The next figure illustrates the significant difference between the two implementations from this point of view.

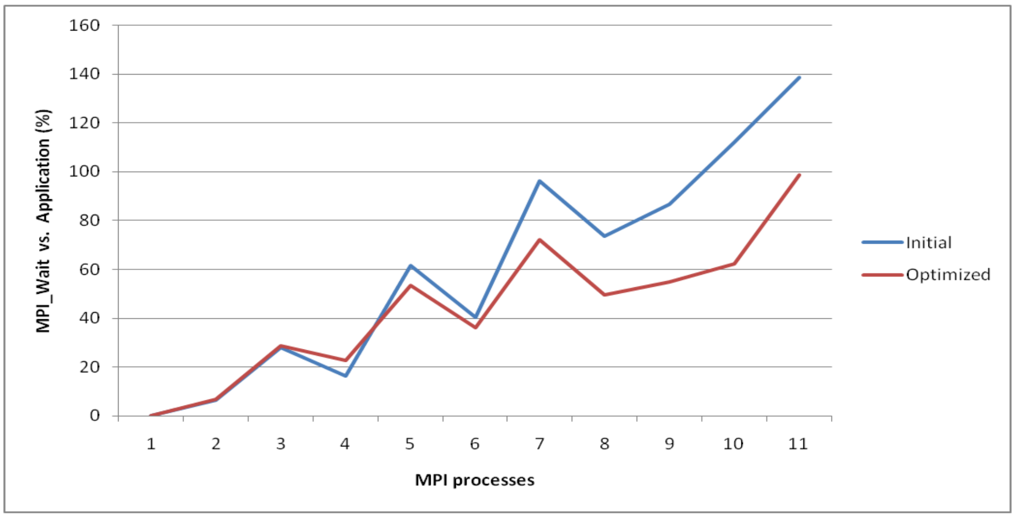


Figure 4.4: Comparative MPI Wait impact on ibm-quad.q

### 4.1.2 Results on ibm-nehalem.q

- OMP_NUM_THREADS = 1 (each blade has only MPI processes)

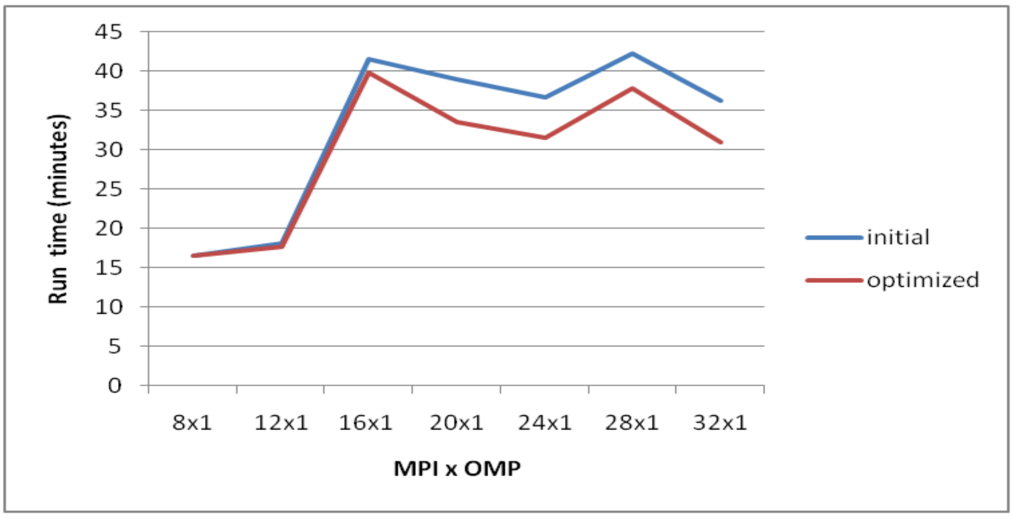


Figure 4.5: Comparative run times on ibm-nehalem.q

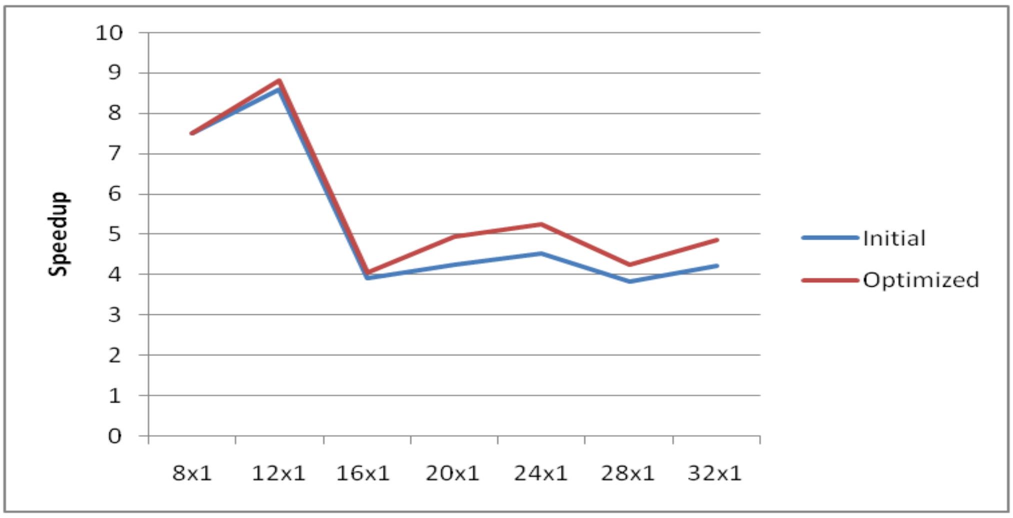


Figure 4.6: Comparative speedup on ibm-nehalem.q

The results are not so good as when using hybrid runs (MPI and OMP), but the improvement is still visible and in some cases it reaches the same maximum as on the ibm-quad nodes (15-20%).

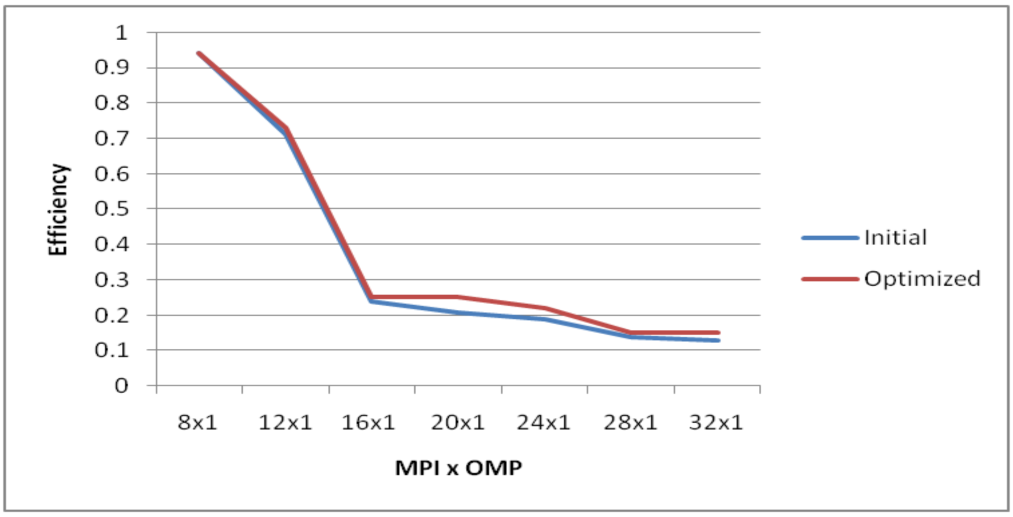


Figure 4.7: Comparative efficiency on ibm-nehalem.q

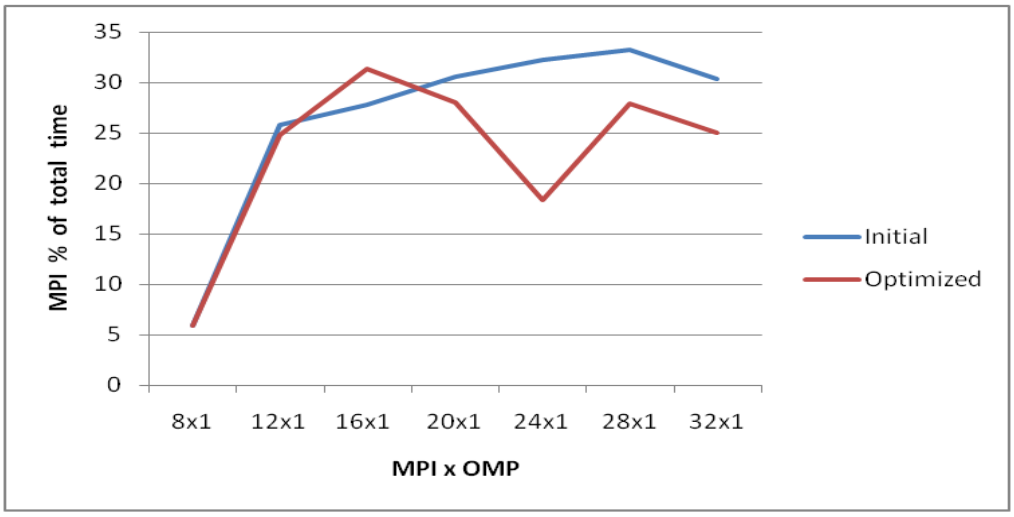


Figure 4.8: Comparative MPI_Wait on ibm-nehalem.q

### 4.1.3 Results on ibm-nehalem.q

- OMP_NUM_THREADS = 12 (each blade has 1x12 MPI x OMP)
- 1 hour of simulation

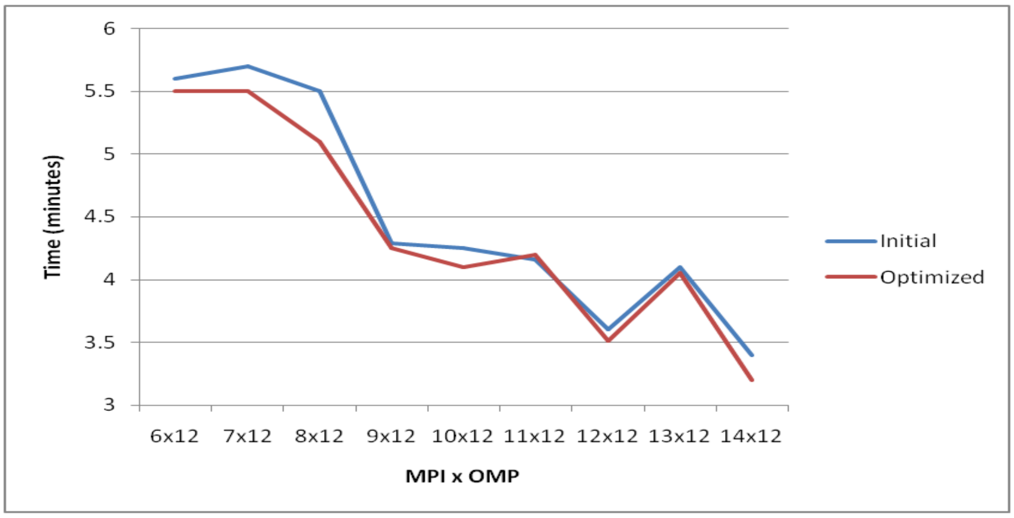


Figure 4.9: Comparative run times on ibm-opteron.q

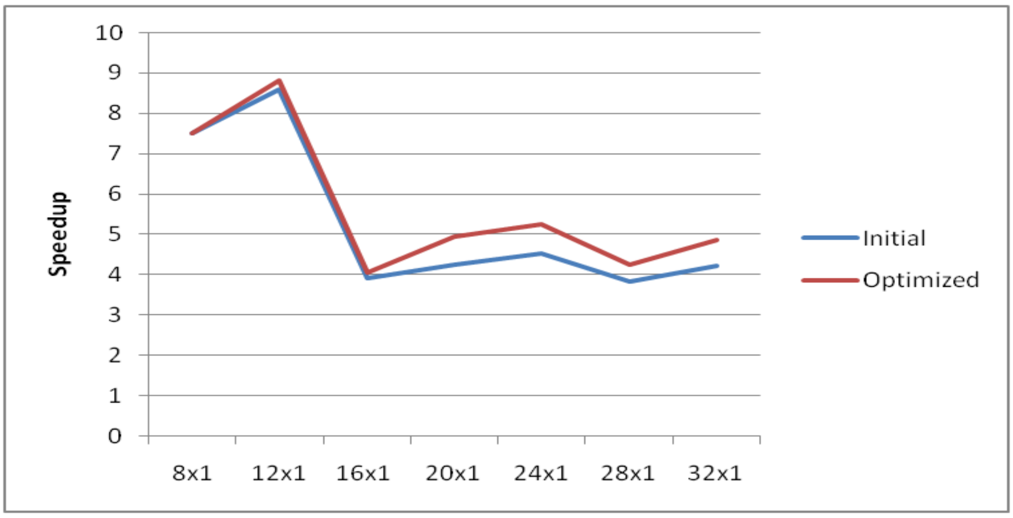


Figure 4.10: Comparative speedup on ibm-nehalem.q

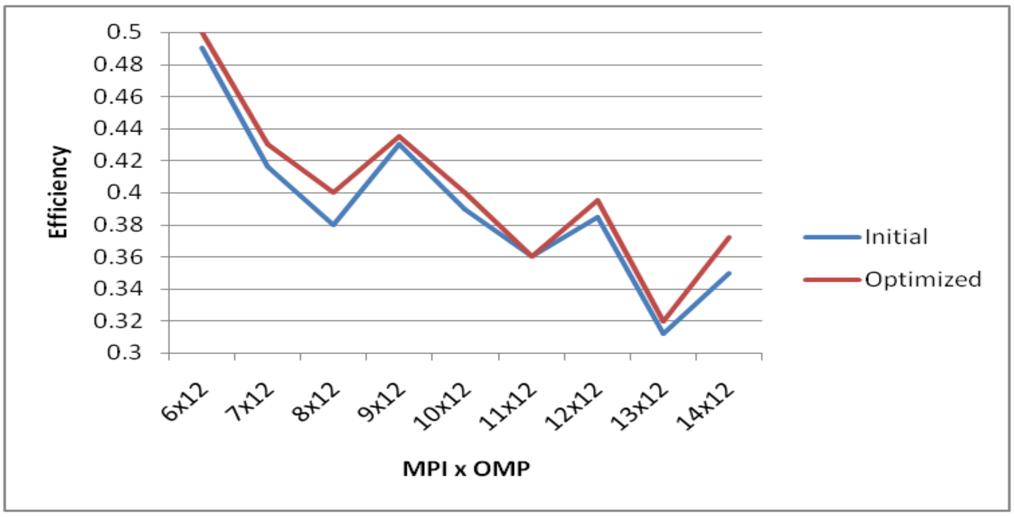


Figure 4.11: Comparative efficiency on ibm-opteron.q

The differences between implementations are smaller than on the other nodes (quad and nehalem), mainly due to a higher interconnection bandwidth (Infiniband) between the processors. The data to be exchanged is rapidly transfered, so there isn't so much need for multiple transfers instead of one for each buffer.

This is the main reason for which we searched for another WRF benchmark, with bigger inputs and therefore more significant use of resources.

## 4.2 User-Oriented WRF Benchmarking Collection - Results on Europe Benchmark

Recognizing the need for a diverse but unified set of benchmark cases in the WRF modeling user community, the User-Oriented WRF Benchmarking Collection represents a joint effort of the University of Alaska's Arctic Region Supercomputing Center and Vienna's University of Natural Resources and Life Sciences Institute of Meteorology [oA].

The vision behind this effort is that users are often interested in determining how much their particular model set up will cost, from a computational perspective. This information will influence decisions in setting up model domains for operational and research WRF simulations, along with aiding in decisions related to necessary hardware and software resources for various model configuration [oA]s.

The primary function of this initiative is to provide the WRF user community with a centralised repository of easy-to-run benchmark cases for a variety of needs, and a reporting facility for users to submit their results and to query results of other users. Each set of benchmarks is intended to address a particular niche of user needs, and is presented in a relatively consistent manner so that users may use common procedures and tools to utilise the collection [oA].

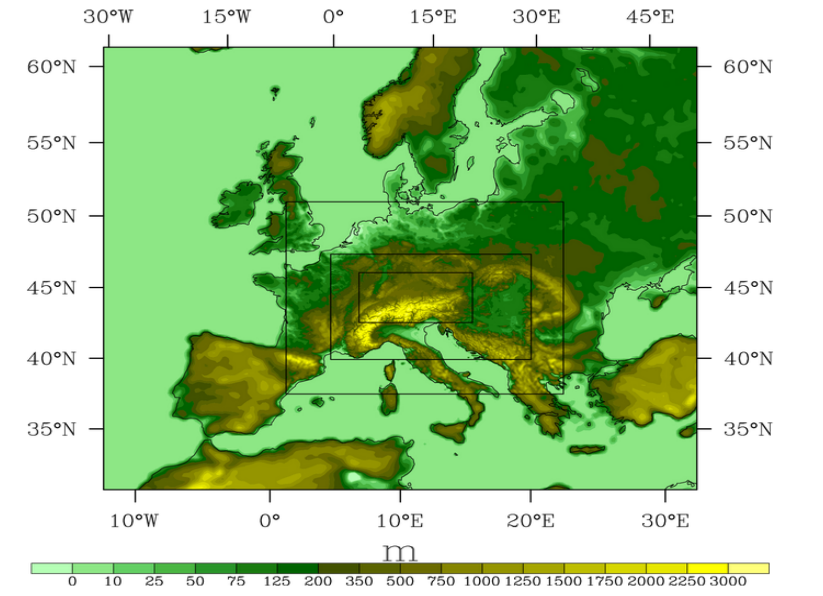
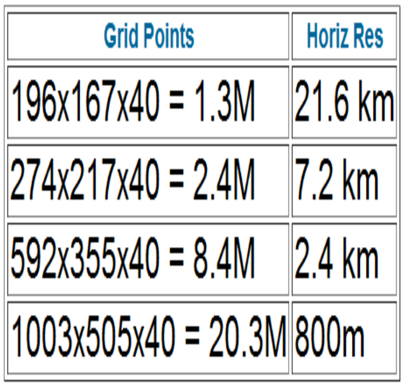


Figure 4.12: The Europe Benchmark domains

European WRF Benchmark Suite is intended to represent typical user needs in complicated and demanding multi-nest and multi-shaped configurations used in regional modeling applications. This case is set up with an emphasis on the complete but more complex evaluation of real-world model runs that feature not only nesting structures, but numerous model parameterisations, input/output issues, etc [oA].



| Grid Points | Horiz Res |
|---|---|
| 196x167x40 = 1.3M | 21.6 km |
| 274x217x40 = 2.4M | 7.2 km |
| 592x355x40 = 8.4M | 2.4 km |
| 1003x505x40 = 20.3M | 800m |

Figure 4.13: The Europe Benchmark grids

Given the dimensions of the input files, we were able to run the scenario only for 3 domains instead of 4 and only on the opteron nodes (which have Infiniband support). Furthermore, the performance is affected for both the original and the modified version of the code. The difference between the two implementations is less visible than the one obtained when running without Infiniband interconnection.

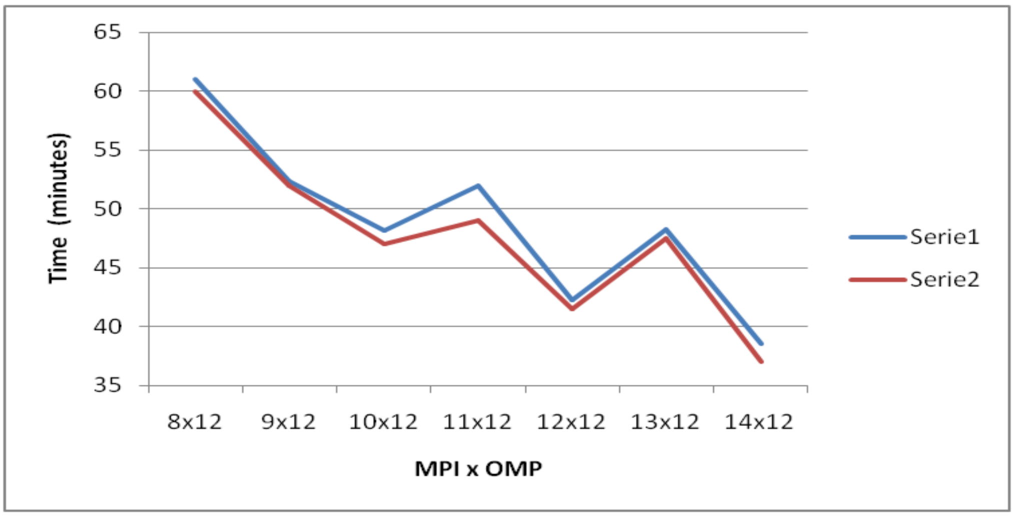


Figure 4.14: Comparative run times on ibm-opteron.q

The above figure shows the comparative results in terms of duration, obtained after 1 hour-simulation on the Opteron nodes.

Chapter 5

# Conclusions and future work

The WRF modeling system has proven to be a high quality mesoscale data assimilation and forecasting system. WRF needs a lot of computational power to manage high-resolution domains, sustaining even active, moving, nested domains. National Center for Information Technology (NCIT) Cluster, located at Politehnica University of Bucharest, has the necessary resources to host, run and develop such a project, adding itself to the community that tries to improve this model.

In this thesis we tried to improve some of WRF functionalities, such as the distributed memory communications. The results that we obtained show that our implementation has a maximum impact when the bandwitdh of the interconnection network on which the model is run is comparable to the length of the messages exchanged.

In other words, the result of the optimization is visible and significant when dealing with low interconnection bandwidths (e.g. the TCP/IP used on the ibm-quad and ibm-nehalem nodes of the NCIT Cluster) or when the processed domains are large enough.

For example, the optimization's results were accentuated while testing with the Europe Benchmark (discussed in the previous chapter), which has much larger inputs and requires more memory and computational resources, despite the fact that the interconnection network used was fast enough (Infiniband).

Another factor that influenced the performance was the processors on which the runs took place, but without changing the degree of performance improvement. For instance, although the nehalem nodes offer a higher computational performance than the quad node, the difference between the two versions of the model (the initial and the optimized one) kept the same for all the three measured parameters: run time, speedup and efficiency.

The WRF performance improvement is necessary for future coupling (use

WRF output as input) with other numerical forecast models, such as FLEXPART (Lagrangian particle dispersion model), which is discussed in another thesis [Con12].

As the WRF model runs faster with the same hardware and computational resources, the input domain can be proportionaly enlarged and thus any model that processes WRF forecasts may as well extend its coverage area.

The present model implementation may still be improved in terms of distributed memory communications, as well as memory access and cache faults. The international community has been extending WRF capabilities in order to make it compatible with a wide range of forecast models.

In my opinion, the tendency is to transform WRF into a global model of weather prediction, while keeping the present grid resolutions (e.g. 1-3-5 km).

# Bibliography

[BM11] Rodica Claudia DUMITRACHE Bogdan MACO. Sistem de prognoz numerc a vremii la rezolutii nalte cosmo-wrf. aplicatii. *Revista stiintifica a Administratiei Natinonale de Meteorologie*, pages 69–76, 2011. [cited at p. 8]

[Con12] Adrian Constantinescu. The weather research and forecast model - optimizations and flexpart coupling. Technical report, University Politehnica of Bucharest, 2012. [cited at p. 1, 34]

[JM] D. Gill J. Michalakes, J. Dudhia. The weather research and forecast model: Software architecture and performance. Technical report, National Center for Athmospheric Research, Boulder, CO. [cited at p. 1, 3, 4, 7]

[Kim06] Hee-Sik Kim. Performance of wrf using upc. Technical report, CRAY Korea Inc., 2006. [cited at p. 11, 12, 13]

[Man07] Giri Mandalika. Building enterprise applications with sun studio. Technical report, Sun Microsystems., 2007. [cited at p. 22]

[NCA11] NCAR. *Advanced Research WRF User's Guide*, January 2011. [cited at p. 3, 5, 6]

[oA] University of Alaska. User-oriented wrf benchmarking collectio. http://weather.arsc.edu/WRFBenchmarking/index.html. [cited at p. 30, 31]

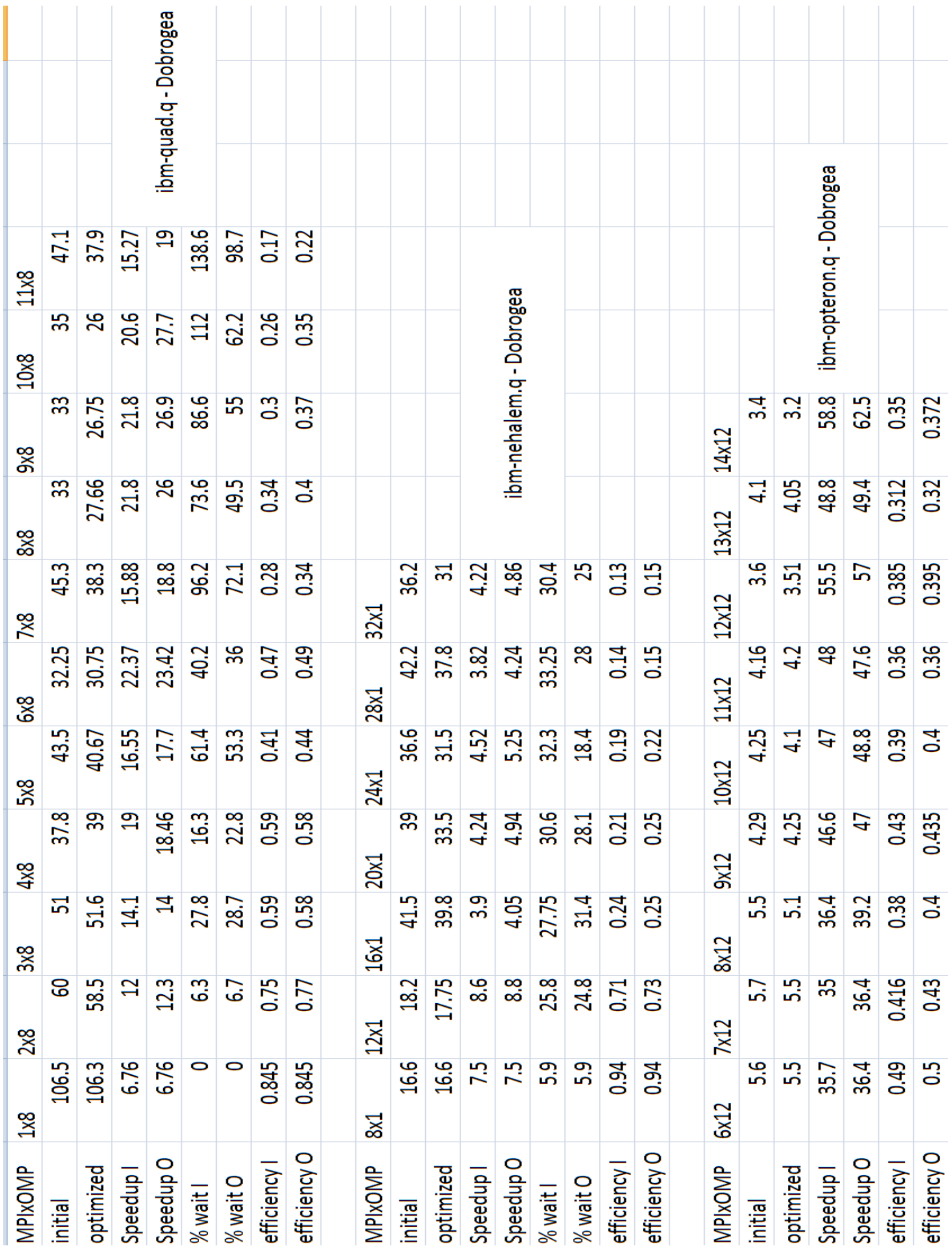
[Pol10] Politehnica University of Bucharest. *The NCIT Cluster Resources User's Guide*, April 2010. [cited at p. 19, 20]

[RRK07] S. Emmerson Rew R. K., G. P. Davis. Netcdf user's guide for c, an interface for data access, version 3. Technical report, National Center for Athmospheric Research (NCAR)., 2007. [cited at p. 20, 21]

[YHK08] John Michalakes Ying-Hwa Kuo, Joseph B. Klemp. Mesoscale numerical weather prediction with the wrf modal. Technical report, National Center for Athmospheric Research (NCAR)., 2008. [cited at p. 1, 4, 6, 7]

# Appendices

## .1 Appendix1 - Results of Dobrogea benchmark



ibm-quad.q - Dobrogea

| MPIxOMP | 1x8 | 2x8 | 3x8 | 4x8 | 5x8 | 6x8 | 7x8 | 8x8 | 9x8 | 10x8 | 11x8 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| initial | 106.5 | 60 | 51 | 37.8 | 43.5 | 32.25 | 45.3 | 33 | 33 | 35 | 47.1 |
| optimized | 106.3 | 58.5 | 51.6 | 39 | 40.67 | 30.75 | 38.3 | 27.66 | 26.75 | 26 | 37.9 |
| Speedup I | 6.76 | 12 | 14.1 | 19 | 16.55 | 22.37 | 15.88 | 21.8 | 21.8 | 20.6 | 15.27 |
| Speedup O | 6.76 | 12.3 | 14 | 18.46 | 17.7 | 23.42 | 18.8 | 26 | 26.9 | 27.7 | 19 |
| % wait I | 0 | 6.3 | 27.8 | 16.3 | 61.4 | 40.2 | 96.2 | 73.6 | 86.6 | 112 | 138.6 |
| % wait O | 0 | 6.7 | 28.7 | 22.8 | 53.3 | 36 | 72.1 | 49.5 | 55 | 62.2 | 98.7 |
| efficiency I | 0.845 | 0.75 | 0.59 | 0.59 | 0.41 | 0.47 | 0.28 | 0.34 | 0.3 | 0.26 | 0.17 |
| efficiency O | 0.845 | 0.77 | 0.58 | 0.58 | 0.44 | 0.49 | 0.34 | 0.4 | 0.37 | 0.35 | 0.22 |

ibm-nehalem.q - Dobrogea

| MPIxOMP | 8x1 | 12x1 | 16x1 | 20x1 | 24x1 | 28x1 | 32x1 |
|---|---|---|---|---|---|---|---|
| initial | 16.6 | 18.2 | 41.5 | 39 | 36.6 | 42.2 | 36.2 |
| optimized | 16.6 | 17.75 | 39.8 | 33.5 | 31.5 | 37.8 | 31 |
| Speedup I | 7.5 | 8.6 | 3.9 | 4.24 | 4.52 | 3.82 | 4.22 |
| Speedup O | 7.5 | 8.8 | 4.05 | 4.94 | 5.25 | 4.24 | 4.86 |
| % wait I | 5.9 | 25.8 | 27.75 | 30.6 | 32.3 | 33.25 | 30.4 |
| % wait O | 5.9 | 24.8 | 31.4 | 28.1 | 18.4 | 28 | 25 |
| efficiency I | 0.94 | 0.71 | 0.24 | 0.21 | 0.19 | 0.14 | 0.13 |
| efficiency O | 0.94 | 0.73 | 0.25 | 0.25 | 0.22 | 0.15 | 0.15 |

ibm-opteron.q - Dobrogea

| MPIxOMP | 6x12 | 7x12 | 8x12 | 9x12 | 10x12 | 11x12 | 12x12 | 13x12 | 14x12 |
|---|---|---|---|---|---|---|---|---|---|
| initial | 5.6 | 5.7 | 5.5 | 4.29 | 4.25 | 4.16 | 3.6 | 4.1 | 3.4 |
| optimized | 5.5 | 5.5 | 5.1 | 4.25 | 4.1 | 4.2 | 3.51 | 4.05 | 3.2 |
| Speedup I | 35.7 | 35 | 36.4 | 46.6 | 47 | 48 | 55.5 | 48.8 | 58.8 |
| Speedup O | 36.4 | 36.4 | 39.2 | 47 | 48.8 | 47.6 | 57 | 49.4 | 62.5 |
| efficiency I | 0.49 | 0.416 | 0.38 | 0.43 | 0.39 | 0.36 | 0.385 | 0.312 | 0.35 |
| efficiency O | 0.5 | 0.43 | 0.4 | 0.435 | 0.4 | 0.36 | 0.395 | 0.32 | 0.372 |

Figure .1: Results