Parallel conversion of satellite image information for a wind energy generation forecasting model

Germán Gadea, Andrés Flevaris, Juan Souteras, Sergio Nesmachnow,
Alejandro Gutiérrez, and Gabriel Cazes

Universidad de la República, Uruguay
{ggadea,aflevaris,jsouteras,sergion,aguti,agcm}@fing.edu.uy

Abstract. This paper presents an efficient parallel algorithm for the problem of converting satellite imagery in binary files. The algorithm was designed to update at global scale the land cover information used by the WRF climate model. We present the characteristics of the implemented algorithm, as well as the results of performance analysis and comparisons between two approaches to implement the algorithm. The performance analysis shows that the implemented parallel algorithm improves substantially against the sequential algorithm that solves the problem, obtaining a linear speedup.

1 Introduction

Wind prediction is crucial for many applications in environmental, energy, and economic contexts. The information about wind is important for weather forecasting, energy generation, aircrafts and ship traffic, dispersal of spilled fuel prediction, coastal erosion, and many other issues. In the last thirty years, researchers have made important advances in methods and models for wind prediction [1,4].

The Weather Research and Forecasting (WRF) model [8] is a flexible and efficient mesoscale numerical weather prediction system developed by a collaborative partnership including several centers, administrations, research laboratories and universities in the USA. With a rapidly growing community of users, WRF is currently in operational use at several centers, labs and universities through the globe, including our research group at Instituto de Mecánica de los Fluidos e Ingeniería Ambiental (IMFIA) de la Facultad de Ingeniería, Universidad de la República, Uruguay.

In our context, WRF is applied to analyze the availability of wind energy, which depends on the wind speed, in order to perform accurate forecasting in the range of 24-48 hours, required for the integration of wind energy into the power grid. These predictions are a valuable help for the power grid operators to make critical decisions, such as when to power down traditional coal-powered and gas-powered plants.

Soil information is very relevant for wind forecasting using WRF, since the terrain type directly affects the wind received by the generators (usually placed at 100 m from the ground). The soil information used by the WRF is outdated, and in the case of Uruguay, the last actualization was performed in the early 1990's. Thus, there is a specific interest on updated soli information in order to improve the accuracy of wind forecasting.

In the WRF model, the information about soil is stored in files using a proprietary binary format. so, in order to perform the soil information update, it is needed to convert the information obtained from satellite images to the binary format used in WRF. The conversion process also includes performing a change of projection due to the input information from satellites has a different projection than the output information.

In order to perform the soil information update not only for our country, but at planetary scale, a large number of satellite images need to be processed. In this context, applying high performance computing (HPC) techniques is a valuable strategy to reduce the execution time required to process the large volume of information in the images. More than 300 images have to be processed in the world scenario, with a total size of 27 GB, and the sequential algorithm demands 1710 minutes of execution time.

The main contributions of the research reported in this article are: i) to introduce two parallel versions—using shared memory and distributed memory approaches—of an algorithm that process the satellite images to update the soil information used for wind prediction in the WRF model, and ii) to report an exhaustive experimental analysis that compares the computational efficiency of the shared and distributed memory parallel versions. The experimental results demonstrate that both parallel implementations achieve good computational efficiency, and the distributed memory is the most efficient method for parallelization.

The rest of the manuscript is organized as follows. Next section present a review of related work on parallel algorithms to process satellite images for wind prediction. Section 3 describes the design and details of the proposed parallel algorithm. The description of the two variants implemented are presented in Section 4. The experimental analysis that compares the two parallel versions and the sequential one is presented in Section 5. Finally, Section 6 summarizes the conclusions of the research and formulates the main lines for future work.

2 Related work

Several works have recently addressed the satellite image processing problem using high-performance computing techniques. These works are mainly focused on processing data received directly from the satellites, making a classification of pixels in order to obtain land cover information [3,5]. Maulik and Sarkar [3] proposed a strategy for satellite image classification by grouping the pixels in the spectral domain. This method allows performing the detection of different land cover regions. A parallel implementation of the proposed algorithm following the master-slave paradigm was presented in order to perform the classification efficiently. The experimental analysis on different remote sensing data performed on INRIA PlaFRIM cluster varying the number of processors from 1 to 100, demonstrated that the proposed parallel algorithm is able to achieve linear speedup when compared against a sequential version of the algorithm.

Nakamura et al. [5] described how the researchers at Tokyo University of Information Sciences receive MODIS data to be used in one of the major fields of research: the analysis of environmental changes. Several applications to analyze environmental changes are developed to execute on the satellite image data analysis system, which is implemented in a parallel distributed system and a database server.

Sadykhov et al. [9] described a parallel algorithm based on fuzzy clustering for processing multispectral satellite images to enforce discrimination of different land covers and to improve area separation. A message passing approach was used as basis of parallel calculation because it allows simple organization of interaction between of calculating processes and synchronization. Experimental testing of developed algorithms and techniques has been carried out using images received from Landsat 7 ETM+ Satellite. However, the article does not report experimental analysis focused on evaluating the performance improvements when using parallel computing techniques.

Plaza et al. [6,7] described a realistic framework to study the parallel performance of high-dimensional image processing algorithms in the context of heterogeneous networks of workstations (NOWs). Both papers provided a detailed discussion on the effects that platform heterogeneity has on degrading parallel performance in the context of applications dealing with large volumes of image data. Two representative parallel image processing algorithms were thoroughly analyzed. The first one minimizes interprocessor communication via task replication. The second one develops a polynomial-time heuristic for finding the best distribution of available processors along a fully heterogeneous ring. A detailed analysis of parallel algorithms is reported, by using an evaluation strategy based on comparing the efficiency achieved by an heterogeneous algorithm on a fully heterogeneous NOW. For comparative purposes, performance data for the tested algorithms on Thunderhead (a large-scale Beowulf cluster at NASA Goddard Space Flight Center) are also provided. The experimental results reveal that heterogeneous parallel algorithms offer a surprisingly simple, platform-independent, and scalable solution in the context of realistic image processing applications.

Unlike the previously commented articles, our research corresponds to a later stage, which took the land cover information generated by some source and convert it to another format to feed the WRF model. We have designed and implemented two parallel implementations of the conversion algorithm, using shared memory and distributed memory approaches. Both parallel implementations of the conversion algorithm are currently operative in our cluster infrastructure (Cluster FING), allowing to perform an efficient conversion of satellite images downloaded from NASA satellites, in order to update the information used by the WRF climate model.

3 Conversion of satellite images to binary files

This section describe the main decisions taken to design the parallel conversion algorithm, and the main features of the parallel model used.

3.1 Design considerations

The proposed algorithm implements the conversion of satellite images from the Terra and Aqua (NASA) satellite to binary files supported by the WRF model.

In the design phase of the algorithm, it was necessary to analyze the input and output files of the conversion process and the way that the data is contained. After that, the design of a strategy for converting the information from the input format to the output format was devised.

In order to reduce the large execution times required by a sequential implementation of the algorithm when processing a large number of images, a parallel implementation was conceived in order to assure a more efficient processing. The parallel implementation is capable to convert the full domain requiring significantly lower execution times, allowing the researchers to scale up and processing world-size scenarios.

3.2 Data-parallel: domain decomposition

The work domain of the WRF model is a grid that covers all the world. Each cell in this grid represents an area of 600×600 kilometers of land cover. Thus, a straightforward domain decomposition is suggested by using the WRF grid. Since the WRF grid is the output of the conversion process, an *output domain descomposition* is used.

The domain decomposition is achieved by generating small cells that are represented by matrices with dimensions 1200×1200 (rows by columns). This data partition is important because it divides the amount of data that each process in the parallel algorithm has to work with.

3.3 Parallel model

Taking into account the characteristics of the algorithm to be parallelized, the selected domain partition, and mainly because there is no need to use border information in order to generate one cell of the output domain, a master-slave model was adopted to implement the parallel algorithms. The use of this model to implement the communication between the processes seems to be appropriate for the conversion algorithm, because the slave processes participating in the conversion process do not have to share information between each other. Figure 1 presents a graphic representation of the proposed parallel algorithm, showing the interaction between the different processes.

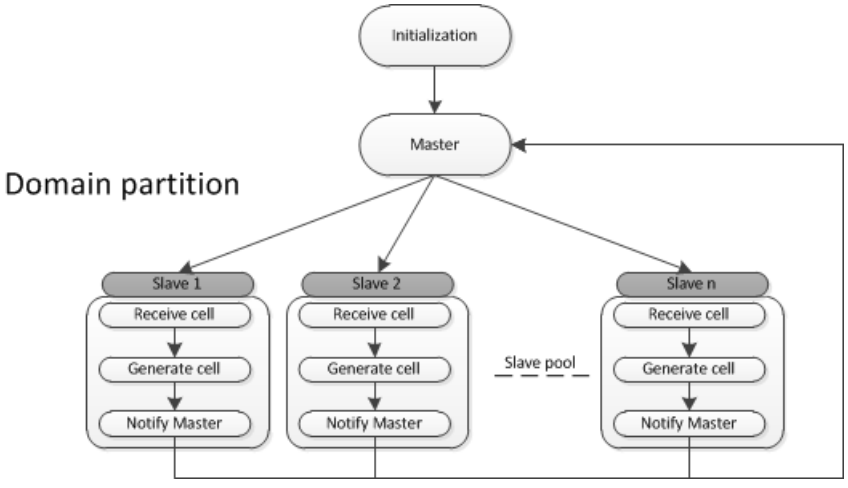


Fig. 1. Scheme of the proposed parallel algorithm.

In the proposed parallel model, the master process initializes all data structures, variables and control structures, and then continues fully dedicated to control the execution of the conversion process and implementing a dynamic load balancing schema for assigning tasks to the slave processes. On the other hand, the slaves processes receive the assigned work or cell, and then they execute the processing tasks in order to convert the land cover data, from the satellite format to binary format.

Two variants of the proposed algorithm were implemented with focus on two different paradigms of parallel computing. One variant was implemented using shared memory, and the other using distributed memory. Both algorithm variants follows the same general approach, but they are designed to execute on different parallel computing infrastructures. The shared memory algorithm is conceived to specifically execute on a multicore computer, while the distributed memory algorithm is able to execute on a distributed infrastructure, such as a cluster of computers.

Both algorithms have been tested in a hybrid cluster infrastructure formed by many multicore computers. Testing the algorithms in the same environment makes possible an analysis comparing the two implementations.

3.4 Load balancing

The two versions of the implemented algorithm gain efficiency by applying a correct load balance strategy. Taking advantage of the domain partition selected, the proposed strategy for load balancing is conceived to be performed in two steps. First, at the initialization step, the master process statically assigns to each slave process a cell to generate. This initial assignment assures that all the slave processes have a work to perform at the beginning of their execution. After that, in each execution step, while the master process has cells to generate and one of the slave processes finish its work, the master dynamically assigns to that slave another cell. The dynamic cell assignment performed by the master process keeps the conversion process running and generating cells, while minimizing the idle time. Each time that one of the slaves processes finishes the generation of a cell, the master process immediately assigns a new one to be generated, and the slave keeps working.

4 Parallel implementation of the conversion algorithm

This section presents the implemented parallel algorithms for the conversion process. In the shared memory algorithm, the master and slaves processes are threads, which are controlled and synchronized using mutexes In the distributed memory algorithm, the master and slaves are processes, which use message passing to perform communication and synchronization.

4.1 Data structures

The common data structures used by the both implemented algorithms are the ones listed in the Data structure frames 1.1 and 1.2.

```
struct descriptorHDF{
  int h;
  int v;
  char fileName[1024];
  char gridName[64];

  // coord SINUSOIDALES
  long double lowerLeftLat_syn;
  long double lowerLeftLon_syn;
  long double lowerRightLat_syn;
  long double lowerRightLon_syn;
  long double upperLeftLat_syn;
  long double upperLeftLon_syn;
  long double upperRightLat_syn;
  long double upperRightLon_syn;

  // coord GEOGRAFICAS
  long double lowerLeftLat_geo;
  long double lowerLeftLon_geo;
  long double lowerRightLat_geo;
  long double lowerRightLon_geo;
  long double upperLeftLat_geo;
  long double upperLeftLon_geo;
  long double upperRightLat_geo;
  long double upperRightLon_geo;
};
```

Data structure 1.1. descriptorHDF

The data structure descriptorHDF contains the fields to save the necessary information about the satellite imagery files. This data structure saves the coordinates in geographic projection and in sinusoidal projection. Both groups of coordinates indicate the area covered by the file. The data structure also contains the fields 'h' and 'v' that indicates the horizontal and vertical position of the HDF file at the satellite imagery grid, respectively. Other fields are used for the file name and for the object that contains the information to be converted.

```
struct celdaSalida{
  char nombreArchivoSalida[256];
  long double lowerLeftLat;
  long double lowerLeftLon;
  long double lowerRightLat;
  long double lowerRightLon;
  long double upperLeftLat;
  long double upperLeftLon;
  long double upperRightLat;
  long double upperRightLon;
};
```

Data structure 1.2. celdaSalida

The data structure celdaSalida is used in the algorithms to indicate a given slave process which cell of the output grid it must generate. The data structure contains fields for the cell coordinates, and the output binary file name. This structure is communicated between the master and slaves processes when the master assigns a cell to be generated by the slave process.

4.2 Shared memory algorithm

The shared memory version of the conversion algorithm uses a pool of threads, implemented using the standard POSIX thread library (pthread). The algorithm is divided in three procedures, the procedure that runs the master thread, other for each slave thread, and the main procedure. The conversion algorithm has two phases. In the first phase, the main procedure initializes all the threads, mutexes and data structures. The data structures used by the algorithm are an array of 'descriptorHDF' and a matrix of 'celdaSalida'. The shared memory algorithm uses a set of global variables:

- turn: used by each slave thread to indicate the master when it has finished working.
- finish: used to indicates the slave treads to exit.
- iSlave, jSlave: this are two arrays of integer used to indicate which cell generates a slave. The indexes i and j indicate the position on the matrix 'celdaSalida'

In the second phase of the algorithm, the master thread first assigns one cell to each slave, and then it waits for the slave answers, to assign a new cell to the requesting slave. When all cells have been assigned, the master thread waits until all slaves finish their work, and then it sets the 'finish' variable to true, causing all slaves to exit. On the other hand, the slaves threads are implemented as a loop that generate cells and performs the conversion until the 'finish' variable is set to true.

On each loop step, the slave processes execute three main tasks. Initially, the assigned cell is received; after that, the assigned cell is generated, and finally the master process is notified that the cell has been generated and the slave process is available to get a new cell assigned. A pseudocode of the algorithm is presented in Algorithm 1 (main program), Algorithm 2 (master process), and Algorithm 3 (slave processes).

Algorithm 1 Main program

```
1: initialize mutexes
2: create and initialize master thread
3: create and initialize slave threads
4: //parallel execution
5: wait(for the exit of all threads)
6: destroy(all structures used)
```

4.3 Distributed memory algorithm

The parallel algorithm has been implemented to be executed in a distributed infrastructure such as a cluster of computers, using the C language and the Message Passing Interface library (MPI) [2]. MPI allows simple organization, communication, and synchronization of the master and the slaves processes.

For the distributed execution, the set of satellite images to convert (with a total size of 27 GB), was stored in the file system of the cluster, that is accessible by all running processes.

Algorithm 2 Master thread

```
1: for all slave thread do
2:     assing(cell to generate)
3:     unlock(slave)
4: end for
5: while has cell to be generated do
6:     wait(slave answer)
7:     receive(answer)
8:     assign(new cell to the slave who answered)
9:     unlock(slave who answered)
10: end while
11: while are slave working do
12:     wait(slave answer)
13: end while
14: finish ← true
```

Algorithm 3 Slave thread

```
1: isThereWork ← true
2: while isThereWork do
3:     lock(until work is assigned)
4:     if not finish then
5:         i ← iSlave[my_id]
6:         j ← jSlave[my_id]
7:         search(HDF files to use)
8:         stitch(HDF files)
9:         proyectAndSubset(cell to generate)
10:        generateBinaryFile()
11:        wait(turn to alert the master)
12:        turn ← my_id
13:        alert(master)
14:    else
15:        isThereWork ← false
16:    end if
17: end while
```

The creation of distributed processes is configured and built using the available TORQUE manager in the cluster FING. The number of processes to create, the cluster nodes to be used, the distribution of processes between nodes, and the priority of the scheduled job are indicated in a configuration file.

The structure of the algorithm closely matches the shared memory algorithm already described. In a first step, the master process initializes the data structures, and in the second stage, the master process begins to perform dynamic load balancing by assigning the work to the slave processes. Then, it waits the responses from the slaves processes, and then assigns new work to idle slaves. On the other hand, the slave processes execute a loop with the following steps: expect a cell to generate, running the conversion process of the received cell, and finally notifies the end of processing to the master process.

Slaves remain in this loop until the master process indicates to finish the execution. A pseudocode of the algorithm is presented in the algorithm 4.

One of the most important tasks in the communication between the master process and slave processes is sending the array with the information of the input files (descriptorHDF). Since this is a large amount of information organized in an array of structs, the use of the common functions of MPI for sending data (MPI_Send) is ineffective, since they cause corruption in the information. For properly implement the communication, it was necessary to use specific MPI functions for sending arrays by performing a serialization of structures (functions MPI_Buffer_attach, MPI_Buffer_dettach and MPI_Bsend). The master process has to wait for messages sent by the slave processes to report that they have finished processing the assigned work, so the master process is blocked by applying the MPI_Recv function using the MPI_ANY_SOURCE flag, which allows receiving messages from any process. To find out which process is the source of communication, the master reads the status parameter returned by the operation and so is obtained the rank that identifies the slave process.

Algorithm 4 Distributed memory algorithm

```
1:  initialize MPI structures
2:  master process do
3:      initialize matrix of celdaSalida
4:      initialize array of descriptorHDF
5:  master process send array of descriptorHDF to all slaves
6:  if master process then
7:      for i = 1 → #process do
8:          send(cell[i], process[i])
9:      end for
10:     while not receive all slaves answers do
11:         waitForAnswer(answer)
12:         slave_id ← answer.slave_id
13:         cell ← answer.cell
14:         markCellAsGenerated(cell)
15:         if not allCellsGenerated() then
16:             cell ← nextCell()
17:             send(cell, process[slave_id])
18:         end if
19:     end while
20:     send finish message to all slaves processes
21: else if slave process then
22:     while not receive finish message do
23:         cell_id ← receive()
24:         generate(cell)
25:         sendMaster(cell_id, my_id)
26:     end while
27: end if
```

5 Experimental evaluation

This section presents the results of the experimental evaluation for the two parallel versions of the proposed algorithm.

5.1 Development and execution platform

Both implementations of the parallel conversion algorithms were developed in C. The distributed algorithm uses MPICH version 1.2.7, and the shared memory algorithm was implemented using the pthread library. The experimental evaluation was performed in a server with two Intel quad-core Xeon processors at 2.6 GHz, with 8 GB RAM, CentOS Linux, and Gigabit Ethernet (Cluster FING, Facultad de Ingeniería, Universidad de la República, Uruguay; cluster website: `http://www.fing.edu.uy/cluster`).

5.2 Data used in the experimental evaluation

The input of the conversion process is a set with more than 300 image files in HDF format, with a total size of 28 GB of information. As a baseline reference, converting all these images with a sequential process takes 1710 minutes.

5.3 Performance metrics

The most common metrics used by the research community to evaluate the performance of parallel algorithms are the *speedup* and the *efficiency*.

The speedup evaluates how much faster a parallel algorithm is than its corresponding sequential version. It is computed as the ratio of the execution times of the sequential algorithm (T_S) and the parallel version executed on m computing elements (T_m) (Equation 1). The ideal case for a parallel algorithm is to achieve linear speedup ($S_m = m$), but the most common situation is to achieve sublinear speedup ($S_m < m$), mainly due to the times required to communicate and synchronize the parallel processes.

The efficiency is the normalized value of the speedup, regarding the number of computing elements used to execute a parallel algorithm (Equation 2). The linear speedup corresponds to $e_m = 1$, and in the most usual situations $e_m < 1$.

We have also studied the *scalability* of the proposed parallel algorithm, defined as the ratio of the execution times of the parallel algorithm when using one and m computing elements (Equation 3).

$$S_m = \frac{T_S}{T_m} \qquad (1) \qquad e_m = \frac{S_m}{m} \qquad (2) \qquad Sc_m = \frac{T_1}{T_m} \qquad (3)$$

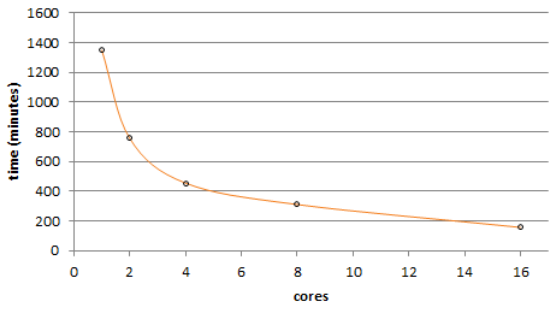
5.4 Performance evaluation and discussion

This subsection presents and analyzes the performance results of the implemented shared memory and distributed memory parallel algorithms. All the execution time results reported correspond to the average values computed in five independent execution performed for each algorithm and experimental scenario.

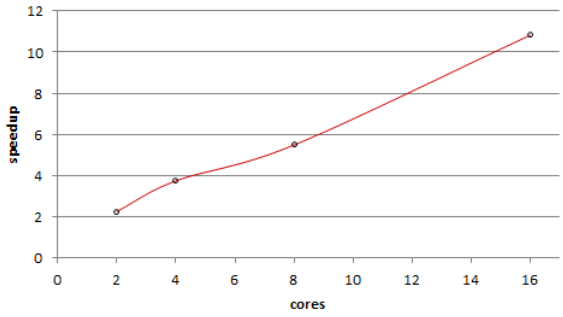
Shared memory implementation. The experimental analysis of the shared memory parallel implementation was performed on a single host, varying the number of cores used. Table 1 reports the execution time results and the performance metrics for the for the shared memory implementation, and Figures 2 and 3 graphically summarize the results.

# cores (m)	**time** (minutes)	**speedup** (S_m)	**efficiency** (e_m)	**scalability** (Sc_m)
1	1349.00	1.27	1.27	1.00
2	683.00	2.26	1.13	1.78
4	456.00	3.75	0.94	2.96
8	311.67	5.49	0.69	4.33
16	158.00	10.82	0.68	8.54

Table 1. Performance metrics for the shared memory implementation.

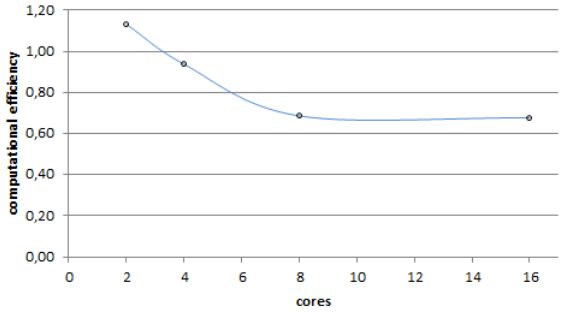


(a) Execution time

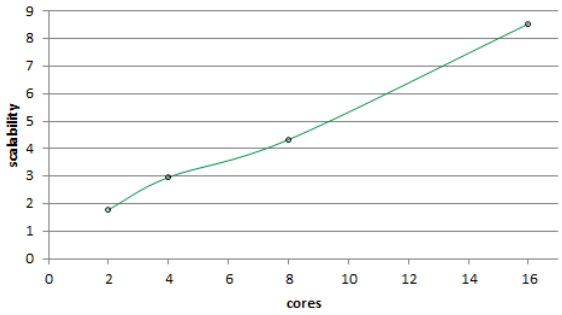


(b) Speedup

Fig. 2. Execution time and speedup for the shared memory implementation.



(a) Efficiency



(b) Scalability

Fig. 3. Efficiency and scalability for the shared memory implementation.

The results in Table 1 indicate that acceptable improvements in the execution times are obtained when using several cores in the shared memory parallel implementation of the conversion algorithm. Fig. 2(a) graphically shows the reduction in the execution times. While the sequential algorithm takes 1710 minutes to perform and the parallel version running on a single core takes 1349 minutes, the execution time is reduced down to 158 minutes when using the maximum number of cores available in the execution platform (16). Fig. 2(b) indicates that when using up to four cores, the parallel algorithm comes to obtain a linear speedup, and even superlinear speedup when two cores are used, mainly due to the time improvements in the image loading process. Using more cores yields to a sublinear speedup behavior, and both the computational efficiency and the scalability of the algorithm reduces, as shown in Fig. 3. Several factors can be mentioned as possible explanations for this behavior, including the overhead for the thread management, the simultaneous bus access, and also that the infrastructure used has four cores per processor, so the use of the memory bus for accessing the images have a larger impact when using more than four cores.

Overall, efficient results are obtained for the shared memory implementation of the parallel conversion process. Speedup values up to 10.82 are obtained when using 16 cores, and the efficiency values are almost linear.

Distributed memory implementation The experimental analysis for the distributed memory implementation was performed on a cluster infrastructure, varying the number of hosts and also the number of cores used in each host. Table 2 reports the execution time results and the performance metrics for the for the distributed memory implementation, and Fig. 4 and 5 graphically summarize the results.

# host	**# cores** (m)	**time** (minutes)	**speedup** (S_m)	**efficiency** (e_m)	**scalability** (Sc_m)
1	1	1313,00	1,30	1,30	1,00
	2	783,00	2,18	1,09	1,68
	4	421,33	4,06	1,01	3,12
	8	260,67	6,56	0,82	5,04
	16	158,67	10,78	0,67	8,28
2	2	805.00	2.12	1.06	1.63
	4	408.66	4.18	1.05	3.21
	8	188.33	9.08	1.13	6.97
	16	138.00	2.39	0.77	9.51
4	4	356.67	4.79	1.20	3.68
	8	192.33	8.89	1.11	6.83
	16	110.00	15.55	0.97	11.94

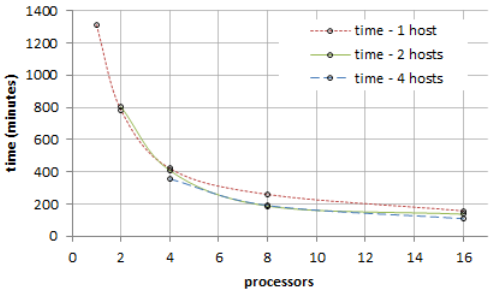
Table 2. Performance metrics for the distributed memory implementation.

The results in Table 2 indicate that notable improvements in the execution times are obtained by the distributed memory parallel implementation of the conversion algorithm, specially when distributing the processing in several hosts.

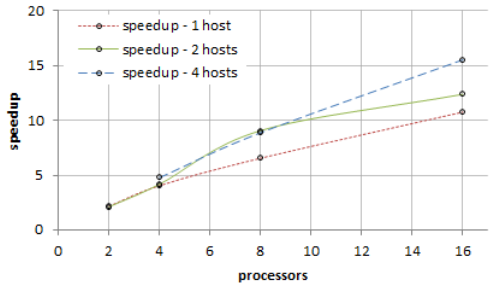
The sequential algorithm takes 1710 minutes to execute in the cluster infrastructure. When using one host, the best performance was reached when using 16 cores, with an execution time of 159 minutes. Using up to four cores, the algorithm comes to obtain a linear speedup as shown in Fig. 4(b). Just like for the sared memory implementation, using more than four cores yields to a sublinear speedup behavior, and both the computational efficiency and scalability of the algorithm reduces. Better results are obtained when distributing the processing in two hosts. The execution time is reduced to 138 minutes when using 16 cores. In this case, the linear speedup/computational efficiency behavior holds up to the use of eight cores, as shown in Figures 4(b), 5(a).

Finally, when using four hosts, the best results of the analysis are obtained. The execution time using 16 cores distributed in four hosts is 110 minutes, almost 16 times faster than the sequential one. The linear speedup/efficiency behavior holds with up 16 cores The scalability of the algorithm also increases up to a value of 11.94.

The previously presented results show that when using the distributed memory algorithm, the better approach is to distribute the execution across several hosts, rather than using additional cores into a single host. The possible reason for this behavior is that when more hosts are involved, more resources are available for the distributed image processing (i.e. memory, disk access, number of open files per process). This is consistent with the large number of files and memory used by the algorithm.



(a) Execution time



(b) Speedup

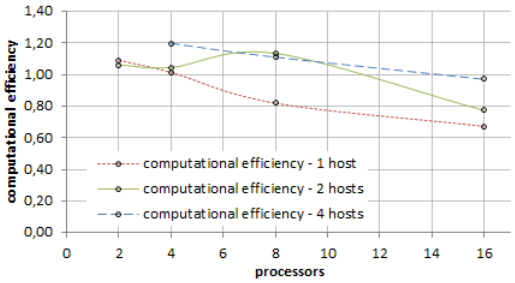
Fig. 4. Execution time and speedup for the distributed memory implementation.

Comparison: shared memory vs. distributed memory. By comparing the two implemented algorithms, the main conclusion is that the distributed memory version reaches better performance in the four hosts scenario. As it was already commented, this situation occurs due to the use of distributed resources, which improves the efficiency since there is a minimal communication between the master and the slaves, and no data exchange is performed between slaves. At the same time, by distributing the resources utilization, the waiting time for input/output and the time added by the operating system tasks do not impact significantly in the efficiency. The experimental results suggest that even better performance could be obtained when using an increasing number of hosts.

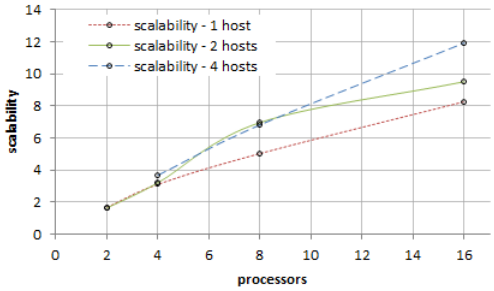
6 Conclusions and future work

This article presented an efficient algorithm for converting the full domain of land cover satellite images to the binary files within the WRF model. The proposed method is an important contribution, as it helps to efficiently generate better wind forecasts.

The implemented parallel algorithm has already been used to generate updated binary WRF files, an the results are now used for wind forecasting at wind farm Emanuelle Cambilargiu, Uruguay. In addition, the binaries for full world are published, and the outcome of this research is currently under examination by experts from NCAR (National Center for Atmospheric Research, USA) to be included in future releases of WRF.



(a) Efficiency



(b) Scalability

Fig. 5. Efficiency and scalability for the distributed memory implementation.

The parallel implementation of the conversion process provided an accurate and efficient method to perform the image processing. Two implementations, using shared and distributed memory, were implemented and analyzed. Regarding to the performance results, both algorithms allowed to obtain significant reductions in the execution times when compared with a traditional sequential implementation. The shared memory implementation did not scale well when more than four cores are used within the same host. On the other hand, the distributed memory algorithm have the best efficiency results: while the sequential algorithm took about 28 hours to perform the conversion, the distributed memory algorithm executing on 4 hosts takes 110 minutes to complete the process. A linear speedup behavior was detected for the distributed memory algorithm, and speedup values of up to **15.55** were achieved when using 16 cores distributed on four hosts. The computational efficiency is almost one, meaning that we are in presence of an almost-ideal case of performance improvement. The values of the scalability metric indicate that the distributed memory implementation of the proposed parallel algorithm scales well when more hosts are used, mainly due to the minimal need of communications between the master and slave processes. The presented results suggest that adding more hosts would improve even more the efficiency, and it also allow to tackle more complex image processing problems.

The main lines for future work are related with further improving the computational efficiency of the proposed implementations and studying the capability of tackling more complex versions of the satellite image processing problem. Regarding the first line, specific details about input/output performance and data bus access should be studied, in order to overcome the efficiency loss of the shared memory implementation. In addition, the scalability of the distributed memory implementation should be further analyzed, specially to determine the best approach for tackling more complex image processing problems (e.g. with better spatial and time resolution), eventually by using the computer power available in large distributed computing infrastructures, such as grid computing and volunteer-computing platforms.

7 Acknowledgments

The work of S. Nesmachnow, A. Gutierrez, and G. Cazes was partially funded by ANII and PEDECIBA, Uruguay.

References

1. J. Cornett and D. Randerson. *Mesoscale wind prediction with inertial equation of motion*. Number 48 in NOAA technical memorandum ERL ARL ;. Air Resources Laboratory, Environmental Research Laboratories, Las Vegas, 1975.
2. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with Message-Passing Interface*. MIT Press, 1999.
3. U. Maulik and A. Sarkar. Efficient parallel algorithm for pixel classification in remote sensing imagery. *Geoinformatica*, 16(2):391–407, April 2012.
4. C. Monteiro, R. Bessa, V. Miranda, A. Botterud, J. Wang, and G. Conzelmann. Wind power forecasting : state-of-the-art 2009. *Information Sciences*, page 216, 2009.
5. Akihiro Nakamura, Jong Geol Park, Kotaro Matsushita, Kenneth J. Mackin, and Eiji Nunohiro. Development and evaluation of satellite image data analysis infrastructure. *Artif. Life Robot.*, 16(4):511–513, February 2012.
6. Antonio Plaza, Javier Plaza, and David Valencia. Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. *J. Supercomput.*, 40(1):81–107, April 2007.
7. Antonio J. Plaza. Parallel techniques for information extraction from hyperspectral imagery using heterogeneous networks of workstations. *J. Parallel Distrib. Comput.*, 68(1):93–111, January 2008.
8. R. Rozumalski. *WRF Environmental Modeling System - User Guide*. National Weather Service SOO Science and Training Resource Center, release 2.1.2.2 edition, May 2006.
9. R. Sadykhov, A. Dorogush, Y. Pushkin, L. Podenok, and V. Ganchenko. Multispectral satellite images processing for forests and wetland regions monitoring using parallel mpi implementation. *Envisat Symposium*, pages 1–6, 2010.