1

Improving Atmospheric Model Performance on a Multi-Core Cluster System

Carla Osthoff[1], Roberto Pinto Souto[1], Fabrício Vilasbôas[1], Pablo Grunmann[1], Pedro L. Silva Dias[1], Francieli Boito[2], Rodrigo Kassick[2], Laércio Pilla[2], Philippe Navaux[2], Claudio Schepke[2], Nicolas Maillard[2], Jairo Panetta[3], Pedro Pais Lopes[3] and Robert Walko[4]

[1]Laboratório Nacional de Computação Científica (LNCC)
[2]Universidade Federal do Rio Grande do Sul (UFRGS)
[3]Instituto Nacional de Pesquisas Espaciais (INPE)
[4]University of Miami
[1,2,3]Brazil
[4]USA

1. Introduction

Numerical models have been used extensively in the last decades to understand and predict weather phenomena and the climate. In general, models are classified according to their operation domain: global (entire Earth) and regional (country, state, etc). Global models have spatial resolution of about 0.2 to 1.5 degrees of latitude and therefore cannot represent very well the scale of regional weather phenomena. Their main limitation is computing power. On the other hand, regional models have higher resolution but are restricted to limited area domains. Forecasting on limited domain demands the knowledge of future atmospheric conditions at domain's borders. Therefore, regional models require previous execution of global models.

OLAM (Ocean-Land-Atmosphere Model), initially developed at Duke University (Walko & Avissar, 2008), tries to combine these two approaches to provide a global grid that can be locally refined, forming a single grid. This feature allows simultaneous representation (and forecasting) of both the global and the local scale phenomena, as well as bi-directional interactions between scales.

Due to the large computational demands and execution time constraints, these models rely on parallel processing. They are executed on clusters or grids in order to benefit from the architecture's parallelism and divide the simulation load. On the other hand, over the next decade the degree of on-chip parallelism will significantly increase and processors will contain tens and even hundreds of cores, increasing the impact of levels of parallelism on clusters. In this scenario, it is imperative to investigate the scale of programs on multilevel parallelism environment.

This chapter is based on recent works from *Atmosfera Massiva Research Group*[1] on evaluating OLAM's performance and scalability in multi-core environments - single node and cluster.

Large-scale simulations, as OLAM, need a high-throughput shared storage system so that the distributed instances can access their input data and store the execution results for later analysis. One characteristic of weather and climate forecast models is that data generated during the execution is stored on a large amount of small files. This has a large impact on the scalability of the system, especially when executing using parallel file systems: the large amount of metadata operations for opening and closing files, allied with small read and write operations, can transform the I/O subroutines in a significant bottleneck.

General Purpose computation on Graphics Processing Units (GPGPU) is a trend that uses GPUs (Graphics Processing Units) for general-purpose computing. The modern GPUs' highly parallel structure makes them often more effective than general-purpose CPUs for a range of complex algorithms. GPUs are "many-core" processors, with hundreds of processing elements.

In this chapter, we also present recent studies that evaluates a implementation of OLAM that uses GPUs to accelerate its computations. Therefore, this chapter presents an overview on OLAM's performance and scalability. We aim at exploiting all levels of parallelism in the architectures, and also at paying attention to important performance factors like I/O.

The remainder of this chapter is structured as follows. Section 2 presents the Ocean-Land-Atmosphere Model, and Section 3 presents performance experiments and analysis. Related works are shown in Section 4. The last section closes the chapter with final remarks and future work.

2. The Ocean-Land-Atmosphere Model – OLAM

High performance implementation of atmospheric models is fundamental to operational activities on weather forecast and climate prediction, due to execution time constraints — there is a pre-defined, short time window to run the model. Model execution cannot begin before input data arrives, and cannot end after the due time established by user contracts. Experience in international weather forecast centers points to a two-hour window to predict the behavior of the atmosphere in coming days.

In general, atmospheric and environmental models comprise a set of Partial Differential Equations which include, among other features, the representation of transport phenomena as hyperbolic equations. Their numerical solution involves time and space discretization subject to the Courant Friedrichs Lewy (CFL) condition for stability. This imposes a certain proportionality between the time and space resolutions, where the resolution is the inverse of the distance between points in the domain mesh. For a 1-dimensional mesh, the number of computing points n is given by L/d, where L is the size of the domain to be solved and d is the distance between points over this domain. In our case, the mesh is 4-dimensional (3 for space and 1 for time). The computational cost is of $O(n^4)$ if the number of vertical points also increases with n, where n is the number of latitude or longitude points in the geographical domain of the model. The resolution strongly influences the accuracy of results.

[1] http://gppd.inf.ufrgs.br/atmosferamassiva

Operational models worldwide use the highest possible resolution that allow the model to run at the established time window in the available computer system. New computer systems are selected for their ability to run the model at even higher resolution during the available time window. Given these limitations, the impact of multiple levels of parallelism and multi-core architectures in the execution time of operational models is indispensable research.

This section presents the Ocean-Land-Atmosphere Model (OLAM). Its characteristics and performance issues are discussed. We also discuss the parameters used in the performance evaluation.

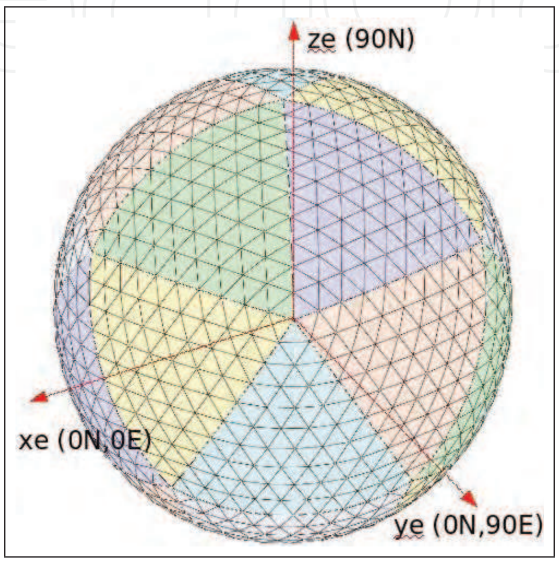


Fig. 1. OLAM's subdivided icosahedral mesh and cartesian coordinate system with origin at Earth center.

OLAM was developed to extend features of the Regional Atmospheric Modeling System (RAMS) to the global domain (Pielke et al., 1992). OLAM uses many functions of RAMS, including physical parameterizations, data assimilation, initialization methods, logic and coding structure, and I/O formats (Walko & Avissar, 2008). OLAM introduces a new dynamic core based on a global geodesic grid with triangular mesh cells. It also uses a finite volume discretization of the full compressible Navier Stokes equations. Local refinements can be defined to cover specific geographic areas with more resolution. Recursion may be applied to a local refinement. The global grid and its refinements define a single grid, as opposed to the usual nested grids of regional models. Grid refined cells do not overlap with the global grid cells - they substitute them.

The model consists essentially of a global triangular-cell grid mesh with local refinement capability, the full compressible nonhydrostatic Navier-Stokes equations, a finite volume formulation of conservation laws for mass, momentum, and potential temperature, and numerical operators that include time splitting for acoustic terms. The global domain greatly expands the range of atmospheric systems and scale interactions that can be represented in the model, which was the primary motivation for developing OLAM.

OLAM was developed in FORTRAN 90 and parallelized with Message Passing Interface (MPI) under the Single Program Multiple Data (SPMD) model.

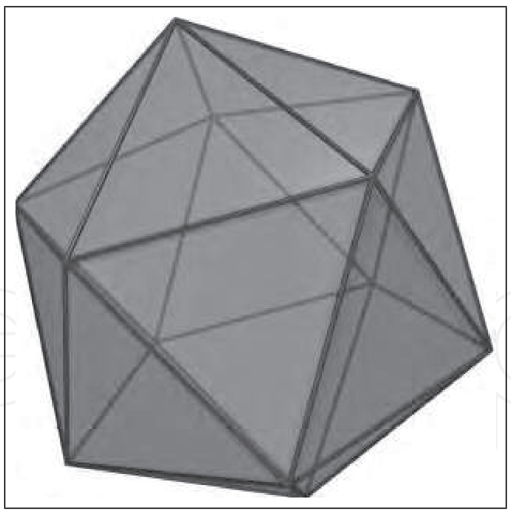


Fig. 2. Icosahedron object.

2.1 OLAM's global grid structure

OLAM's global computational mesh consists of spherical triangles, a type of geodesic grid that is a network of arcs that follow great circles on the sphere (Walko & Avissar, 2008). This can be seen in Figure 1.

The geodesic grid offers important advantages over the commonly used latitude-longitude grid. It allows mesh size to be approximately uniform over the globe, and avoids singularities and grid cells of very high aspect ratio near the poles. OLAM's grid construction begins from an icosahedron inscribed in the spherical earth, as is the case for most other atmospheric models that use geodesic grids. Icosahedron is a regular polyhedron that consists of 20 equilateral triangle faces, 30 triangle edges, and 12 vertices, with 5 edges meeting at each vertex, as represented in Figure 2. The icosahedron is oriented such that one vertex is located at each geographic pole, which places the remaining 10 vertices at latitudes of $\pm tan^{-1}(1/2)$, as shown in Figure 1. The numerical formulation allows for nonperpendicularity between the line connecting the barycenters of two adjacent triangles and the common edge between the triangles.

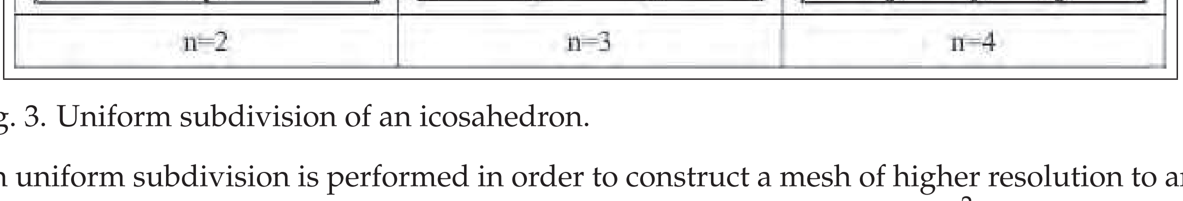


Fig. 3. Uniform subdivision of an icosahedron.

An uniform subdivision is performed in order to construct a mesh of higher resolution to any degree desired. This is done by dividing each icosahedron triangle into N^2 smaller triangles, where N is the number of divisions. The subdivision adds $30(N^2 - 1)$ new edges to the original 30 and $10(N^2 - 1)$ new vertices to the original 12, with 6 edges meeting at each new vertex. This situation is represented in Figure 3. All newly constructed vertices and all edges are then radially projected outward to the sphere to form geodesics.

Figure 1 shows an example of the OLAM subdivided icosahedral mesh and cartesian coordinate system with origin at Earth center, using $N = 10$.

OLAM uses an unstructured approach and represents each grid cell with single horizontal index (Walko & Avissar, 2008). Required information on local grid cell topology is stored and accessed by means of linked lists. If a local horizontal mesh refinement is required, it is performed at this step of mesh construction. The refinement follows a three-neighbor rule that each triangle must share finite edges length with exactly three others.

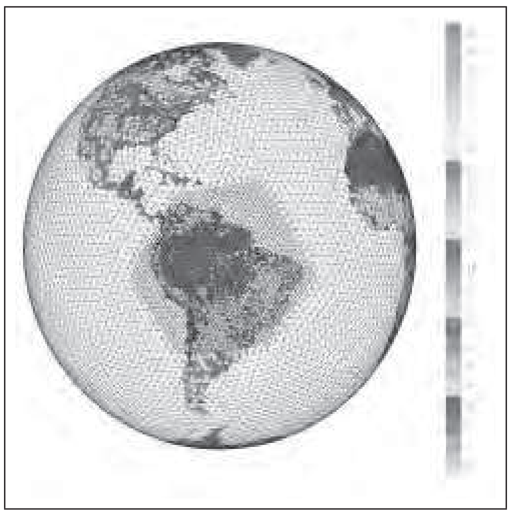


Fig. 4. Local mesh refinement applied to South America.

An example of local mesh refinement is shown in Figure 4, where resolution is exactly doubled in a selected geographic area by uniformly subdividing each of the previously existing triangles into $2x2$ smaller triangles. Auxiliary edges are inserted at the boundary between the original and refined regions for adherence to the three-neighbor rule. Each auxiliary line in this example connects a vertex that joins 7 edges with a vertex that joins 5 edges. More generally, a transition from coarse to fine resolution is achieved by use of vertices with more than 6 edges on the coarser side and vertices with fewer than 6 edges on the finer side of the transition. For more mesh refinement procedure details, refer to (Walko & Avissar, 2011), that presents OLAM's method for constructing a refined grid region for a global Delaunay triangulation, or its dual Voronoi diagram, that is highly efficient, is direct (does not require iteration to determine the topological connectivity although it typically does use iteration to optimize grid cell shape), and allows the interior refined grid cells to remain stationary as refined grid boundaries move dynamically. This latter property is important because any shift in grid cell location requires re-mapping of prognoses quantities, which results in significant dispersion.

The final step of the mesh construction is the definition of its vertical levels. To do this, the lattice of surface triangular cells is projected radially outward from the center of the earth to a series of concentric spheres of increasing radius, as in Figure 5. The vertices on consecutive spheres are connected with radial line segments. This creates prism-shaped grid cells having

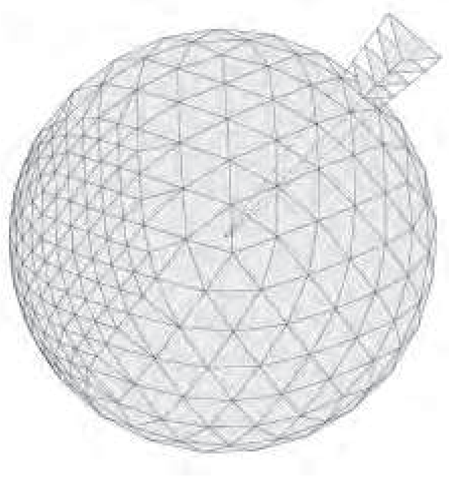


Fig. 5. Local mesh refinement (in the left portion of the image) and projection of a surface triangle cell to larger concentric spheres (in the right portion of the image).

two horizontal faces (perpendicular to gravity) and three vertical faces. The horizontal cross section of each grid cell and column expands gradually with height. The vertical grid spacing between spherical shells may itself vary and usually is made to expand with increasing height.

OLAM uses a C-staggered grid discretization for an unstructured mesh of triangular cells (Walko & Avissar, 2008). Scalar properties are defined and evaluated at triangle barycenters, and velocity component normal to each triangle edge is defined and evaluated at the center of each edge. The numerical formulation allows for nonperpendicularity between the line connecting the barycenters of two adjacent triangles and the common edge between the triangles.

Control volume surfaces for horizontal momentum are the same as for scalars in OLAM. This is accomplished by defining the control volume for momentum at any triangle edge to be the union of the two adjacent triangular mass control volumes. This means that no spatial averaging is required to obtain mass flux across momentum control volume surfaces.

OLAM uses a rotating Cartesian system with origin at the Earth's center, z-axis aligned with the north geographic pole, and x- and y-axes intersecting the equator at 0 deg and 90 deg E. longitude, respectively, as shown in the image of the Figure 1. The three-dimensional geometry of the mesh, particularly relating to terms in the momentum equation and involving relative angles between proximate grid cell surfaces, is worked out in this Cartesian system.

For more details regarding the OLAM's physics parameterizations (radiative transfer, bulk microphysics, cumulus parameterizations, turbulent transfer and surface exchange, and water and energy budgets for the vegetation canopy and multiple soil layers) see (Cotton et al., 2003) paper where most of RAMS development took place. The key point is that these parameterizations are all programmed as column-based processes, with no horizontal communication except possibly between adjacent grid columns. The importance of this is that no horizontally-implicit equation needs to be solved, and in fact the same is true of the dynamic core. This fact can have a huge impact on the communication between MPI processes and the overall efficiency of this communication. Basically, avoidance of horizontal elliptic solvers makes MPI much easier and more efficient.

2.2 OLAM's implementation

OLAM is an iterative model, where each timestep may result in the output of data as defined in its parameters. Its workflow is illustrated in Figure 6.

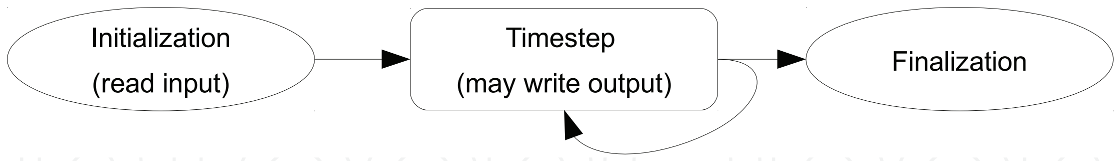


Fig. 6. OLAM's iterative organization.

OLAM input files are not partitioned for parallel processing, as each process reads the input files entirely. Typical input files are: global initial conditions at a certain date and time and global maps describing topography, soil type, ice covered areas, Olson Global Ecosystem (OGE) vegetation dataset, depth of the soil interacting with the root zone, sea surface temperature and Normalized Difference Vegetation Index (NDVI). Reading data over the entire globe is true only for files that are read at the initialization time. For those files that also need to be read during model integration, namely SST and NDVI, because their values are time dependent, the files are organized into separate geographic areas of 30 x 30 degrees each. An individual MPI process does not need to read in data for the entire globe.

After this phase, the processing and data output phases are executed alternately: during each processing phase, OLAM simulates a number of timesteps, evolving the atmospheric conditions on time-discrete units. After each timestep, processes exchange messages with their neighbors to keep the atmospheric state consistent.

After executing a number of timesteps, the variables representing the atmosphere are written to a history file. During this phase, each MPI process opens its own history file for that superstep, writes the atmospheric state and closes the history file. Each client executes these three operations independently, since there is no collective I/O implemented in OLAM.

These generated files are considered of small size for the standards of scientific applications: each file size ranges from 100KB to a few MB, depending on the grid definition and number of MPI processes employed. The amount of written data varies with the number of processes running the simulation (the number of independent output files increases with the processes count).

2.3 OLAM's configuration for the performance evaluation

In our experiments, the represented atmosphere was vertically-divided in 28 horizontal layers. Each execution simulates 24 hours of integration of the equations of atmospheric dynamics without any additional physical calculation (such as moisture and radiative processes) because we have interest only in the impact on the cost of fluid dynamics executions and communications. Each integration timestep simulates 60 seconds of the real time.

We executed tests with resolutions of 40km and 200km and with three implementations of OLAM:

1. **The MPI implementation**: The computation is divided among MPI processes.

2. **The Hybrid MPI/OpenMP implementation**[2]: Each MPI process creates OpenMP threads at the start of the timestep and destroy them after the results output. OpenMP threads execute the *do* loops from OLAM's highest cache miss/hotspot routine, named *progwrtu*.

   Therefore, this implementation uses a different level of parallelism, improving the application memory usage and generating less files. This happens because it maintains the same parallelism degree, but with a smaller number of MPI processes (each of them with a number of threads). As each output file correspond to one MPI Rank, the total number of generated files decreases compared to the MPI-only implementation.
3. **The Hybrid MPI/CUDA implementation**[3]: This implementation starts one MPI process on each core of the platform, and each MPI process starts threads on the GPU device. Due to development time reasons, we decided to implement for this work, only two CUDA kernels out of nine *do* loops from the hotspot routine. Therefore, each MPI process starts two kernel threads on the GPU device.

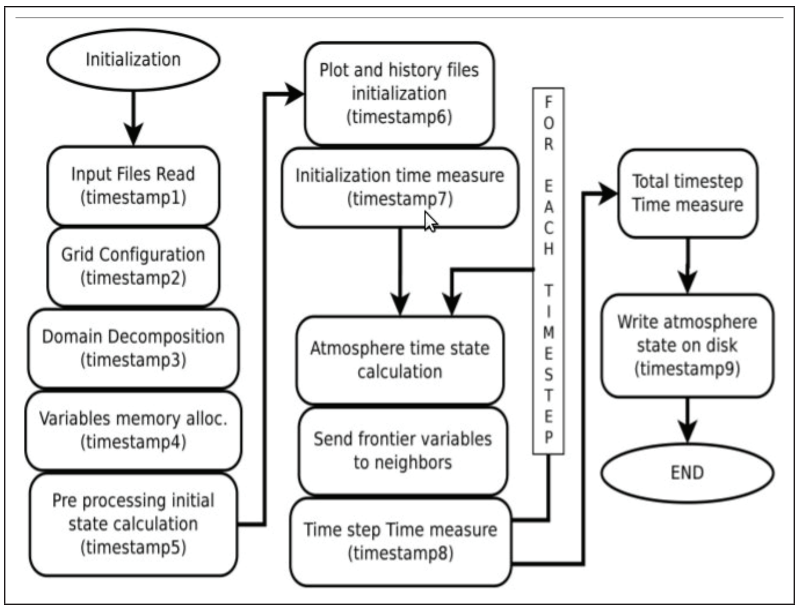

Initialization
Input Files Read (timestamp1)
Grid Configuration (timestamp2)
Domain Decomposition (timestamp3)
Variables memory alloc. (timestamp4)
Pre processing initial state calculation (timestamp5)
Plot and history files initialization (timestamp6)
Initialization time measure (timestamp7)
Atmosphere time state calculation
Send frontier variables to neighbors
Time step Time measure (timestamp8)
FOR EACH TIMESTEP
Total timestep Time measure
Write atmosphere state on disk (timestamp9)
END

Fig. 7. OLAM's algorithm fluxogram.

We present some evaluations obtained with Vtune Performance Analyser. In order to obtain, them, we divided the OLAM's algorithm in three major parts: the initialization, the atmospheric time state calculation and the output. Figure 7 presents this algorithm in details. Finally, we inserted timestamps barriers on selected points of OLAM source (a few module boundaries) in order to correctly assign partial execution times to OLAM main modules.

3. OLAM's performance evaluation

The experiments evaluated the performance of the three implementations of OLAM version 3.3 (MPI, MPI/OpenMP, and MPI/CUDA). The tests were conducted in a multi-core cluster system and in a single multi-core node.

[2] Both the MPI and the Hybrid MPI/OpenMP implementations were developed by Robert Walko (Miami University).

[3] The Hybrid MPI/CUDA implementation was developed by Fabrício Vilasbôas and Roberto Pinto Souto (LNCC).

- The multi-core cluster environment used in the performance measurements considering the Network File System (NFS) in Section 3.1 is a multi-core SGI Altix-XE 340 cluster platform (denoted Altix), located at LNCC. Altix-XE is composed of 30 nodes, where each nodes has two quad-core Intel Xeon E5520 2.27GHz processors, with 128Kb L1 cache, 1024KB L2 cache, 8192KB L3 cache and 24GB of main memory. The nodes are interconnected by an *Infiniband* network. The used software includes MPICH version 2-1.4.1 and Vtune Performance Analyzer version 9.1. The results presented here are the arithmetic average of four executions. The Hyperthread system is active and Turbo-boost increases processor speed when only one core is active.
- The environments used for the file system (PVFS and NFS) tests in Sections 3.2 and 3.3 were part of the Grid'5000[4] infrastructure. The tests were executed on the clusters Griffon (Nancy) – equipped with 96 bi-processed nodes with Intel Xeon (quad-core), 16 GB of RAM and 320 GB of SATA II hard disks – and Genepi (Grenoble) – 34 bi-processed nodes with Intel Xeon (quad-core), 8GB of RAM and 160GB GB of local SATA storage. Both clusters have their nodes interconnected by Gigabit Ethernet.

  For the tests with PVFS file system, 30 clients were evenly distributed among 4 data servers servers, and accessed the file system through a Linux kernel module. We used PVFS version 2.8.2 obtained from http://www.pvfs.org/. We modified OLAM in order to obtain the time spent by each I/O. The tests with the I/O optimization had up to 26 nodes (208 cores) using the shared NFS infrastructure present on the Genepi cluster.
- The performance measurements for the Hybrid MPI/CUDA implementation in Section 3.4 were made on a multi-core/many-core node, denoted prjCuda, located at LNCC, composed of a dual Quad-Core Xeon E5550, 2.67GHz, with 8 MB of L3 cache, 1MB L2 cache, 256KB L1 cache and 24 GB of RAM memory, GTX285 and Tesla C2050. The software employed included MPICH version 2-1.2.p1, Vtune Performance Analyzer version 9.1, CUDA toolkit version 4.0. and PGI Fortran version 11.2 compiler. The experiments evaluate parallel performance of the three implementations of OLAM. As we are running in one single node system, OLAM input and output phase execution times are the same for all implementations. Therefore, they were not considered in our analysis. All single node tests implement a 40km OLAM configuration.

3.1 Experimental results and analysis with NFS on the multi-core cluster environment

We present two experimental analysis in two distinct OLAM workload configurations.

- In the first experiment we performed a 200km OLAM configuration analysis, a typical horizontal resolution most commonly used for climate simulation.
- The second experiment performed a 40km OLAM configuration. This is a typical resolution for global forecast. This experiment decreases 5 times the horizontal distance between points in the globe, hence more points are necessary to cover the same area. The other parameters remain the same as in the 200km configuration. Since it is necessary 25 times the number of points as before to cover the same area at 5 times shorter space intervals, the number of calculations per timestep is now increased 25 times with respect to the previous experiment. Furthermore, it implies a 20-fold increase in the memory workload, thus increasing the the proportion of computing time with respect to data transfers.

[4] http://www.grid5000.fr/

3.1.1 Experiments with resolution of 200km

Figure 8 presents the ideal and measured speedups from 1 to 80 cores for the experiments with a 200km resolution. The results show the performance of the MPI and the Hybrid MPI/OpenMP implementations of OLAM. We observe that the performance for both implementations are similar up to 16 cores. As we increase the number of cores, the Hybrid MPI/OpenMP implementation presents a better performance than the MPI-only implementation.

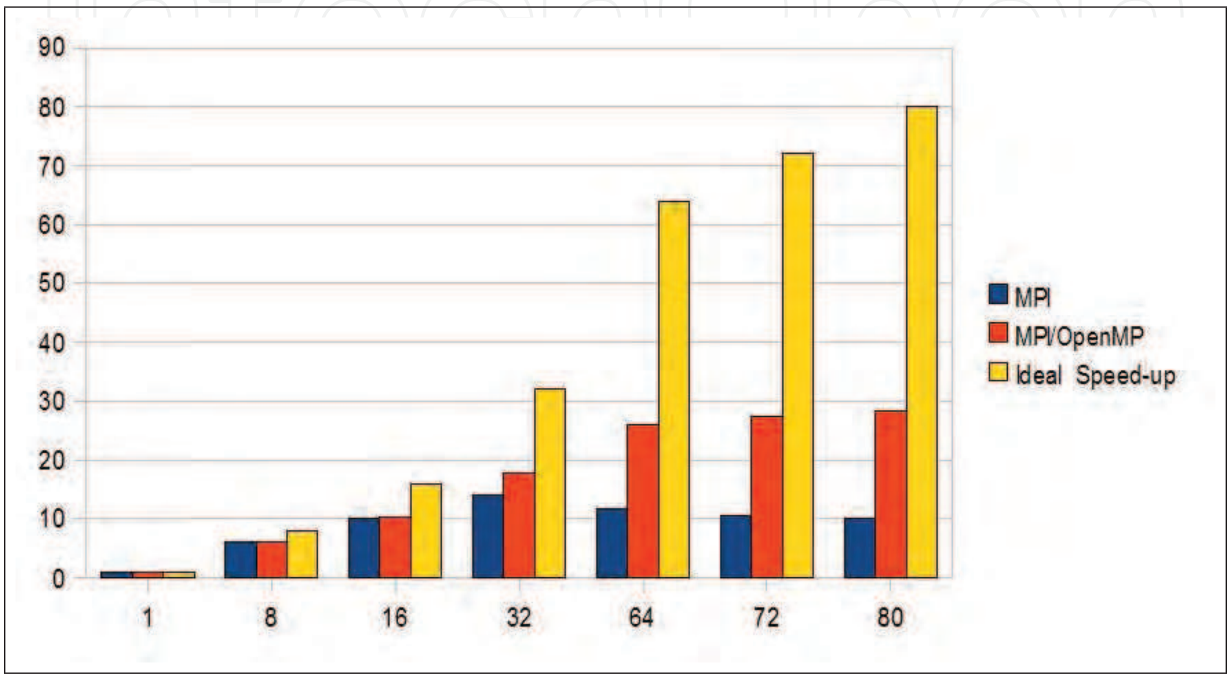


Fig. 8. 200km-resolution ideal and measured speedups for the MPI and the MPI/OpenMP implementations.

Previous works, (Osthoff et al., 2010) and (Schepke et al., 2010), evaluated the performance of OLAM on a multi-core cluster environment and demonstrated that the scalability of the system is limited by output operations performance. OLAM suffers significantly due to the creation of a large number of files and the small requests.

3.1.2 Experiments with resolution of 40km

Figure 9 presents the ideal and measured speedups from 1 to 80 cores for the experiments with a 40km resolution. The results show the performance of the MPI and the Hybrid MPI/OpenMP implementations of OLAM. We observe that the performance of the MPI implementation is better up to 32 cores. As we increase the number of cores, the Hybrid MPI/OpenMP implementation performs better than the MPI-only one. This test shows that as we increase the dependency relationship between computing time and data transfers, output operations overhead decreases overall system performance impact.

In order to explain why the MPI implementation performs better than the Hybrid MPI/OpenMP implementation at lower numbers of cores, we inserted VTUNE Analyser[5]

[5] http://www.intel.com

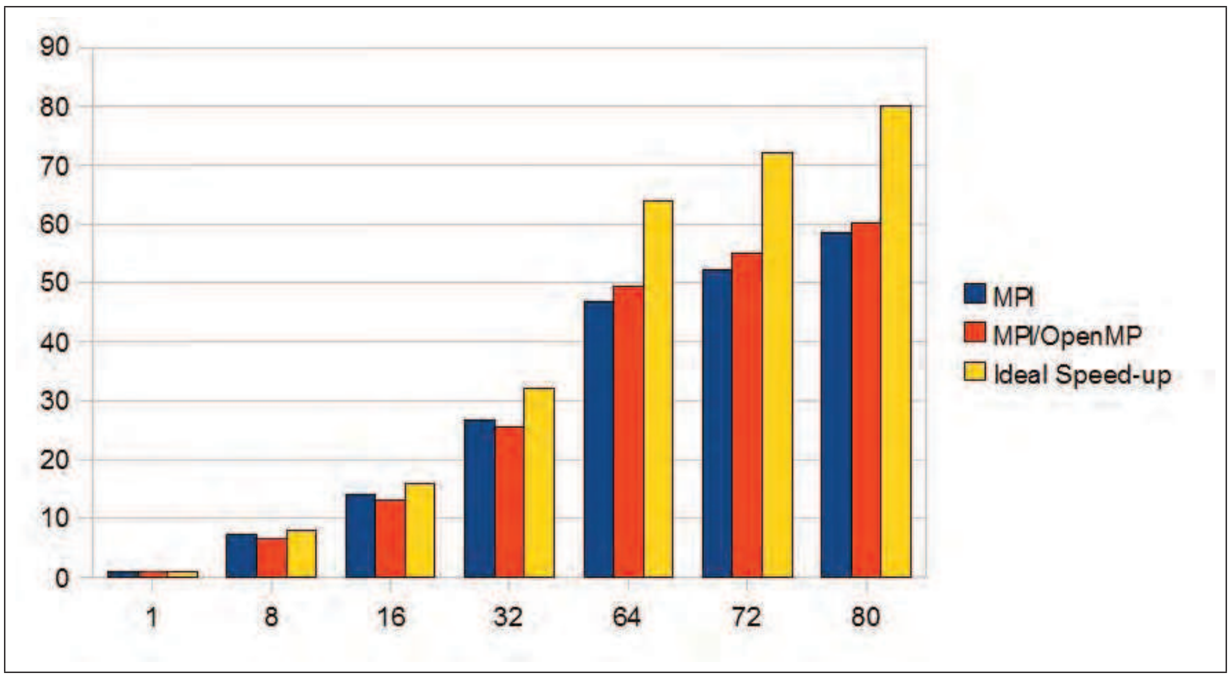


Fig. 9. 40km-resolution ideal and measured speedups for the MPI and the MPI/OpenMP implementations.

performance instrumentation in both codes and evaluated the memory usage in one node, varying the number of processes/threads. We observed that, for this configuration, the memory usage of the MPI implementation grows from 2GB for one process to 5GB to 8 processes. In the other hand, the memory usage of the Hybrid MPI/OpenMP implementation remains almost constant (2GB) as we increase the number of threads. These results confirm that the OpenMP global memory implementation improves OLAM's memory usage on a multi-core system.

The Hybrid MPI/OpenMP implementation parallelized the *do* loops from the subroutine *progwrtu*, This subroutine was responsible for up to 70% of the cpu time of each timestep (Osthoff et al., 2010; Schepke et al., 2010) and up to 75% of the cache misses. We observed that, after the parallelization with OpenMP, the subroutine became responsible for only 28% of the cpu time of each timestep. On the other hand, running with 8 cores, it means that only 28% of the code is running in parallel. This explains why we did not obtained the ideal speedup for the Hybrid MPI/OpenMP implementation.

3.2 Experimental results and analysis with PVFS on a multi-core cluster environment

This section presents the evaluation of the MPI and of the Hybrid MPI/OpenMP implementations in a multi-core cluster regarding I/O performance. We instrumented OLAM in order to obtain the time spent by each I/O operation. For each test instance, we considered on each step the greatest values between the processes. The results presented here are the arithmetic average of four executions. OLAM was configured to output history files at every simulated hour, resulting in 25 files per process (there is one mandatory file creation at the start of the execution).

3.2.1 The MPI implementation's analysis

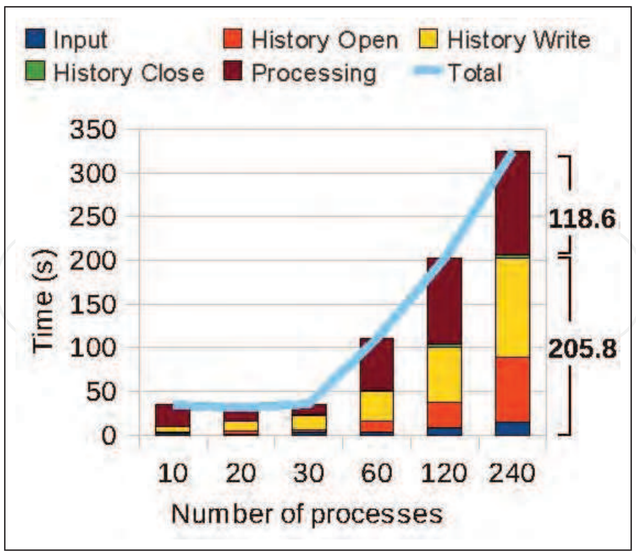


Fig. 10. Execution times for the MPI implementation with PVFS.

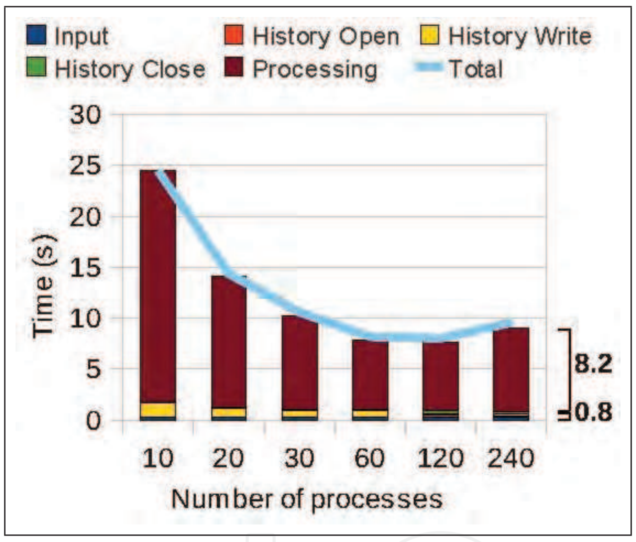


Fig. 11. Execution times for the MPI implementation using local files.

Figure 10 presents the test results for the MPI implementation using PVFS to store all the input and output files and Figure 11 presents the test results storing them in the local disks (Local files). For the second approach to work, the input files (that are previously created in only one node) have to be copied to all the nodes. Also, to work on a non-dedicated cluster, it may be needed to gather the output files on a master node.

Even when considering the times for scatter/gather operations on the input/output files, using the local disks had the best performance - around 36 times better (Figure 12). However, besides the disadvantage of the need to move all these files, the computing nodes do not have local disks in some clusters. This may happen for reasons like power consumption, price, etc. In the local files approach (Figure 11), we can see that, as the number of processes grows, both I/O and processing time decrease, benefited by the higher degree of parallelism. The lowest time is achieved by the 120 processes (4 processes per machine) configuration. PVFS results show a different behavior. In the results (Figure 10) from 10 to 30 processes (one process in each machine), we can see that the processing time decreases (as the degree of parallelism increases), but there is no speedup. This happens because I/O dominates the execution time as the concurrency in the PFS grows.

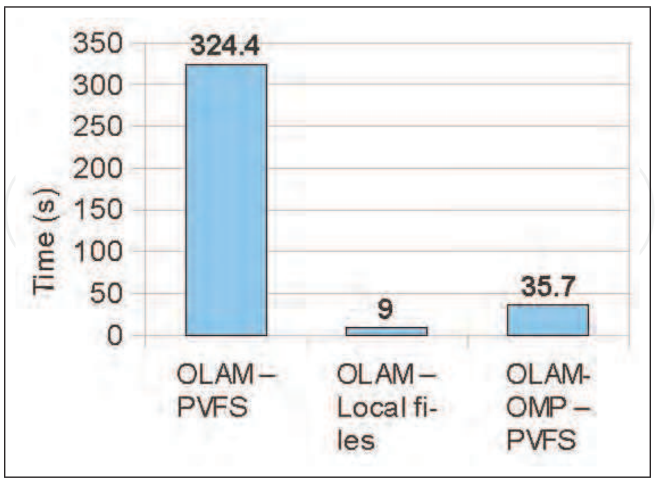


Fig. 12. Execution times for the MPI and the MPI/OpenMP implementations with PVFS and using local files.

When the number of processes in each machine becomes bigger than 1, both I/O and processing times increase. The first is affected by the high load on the file system and by the concurrent I/O requests from the processes on the same node. We believe that the rest of the simulation has its performance impacted by two factors. First, as small write operations are usually buffered, there may be concurrency in the access to the network when the processes communicate. Additionally, when more than one process is located in each machine, there may be competition for the available memory of the machine. We observed such phenomenon in previous works (Schepke et al., 2010).

3.2.2 The hybrid MPI/OpenMP implementation's analysis

Figure 13 shows the results for the Hybrid OLAM/OpenMP implementation with PVFS. The observed behavior is similar to the one shown in Figure 10 in the sense that there is no significant speedup. However, the processing time dominated the execution time in this test, not the I/O time. The time spent in I/O did not grow linearly with the number of processes, indicating that, with this number of clients, the total capacity of the parallel file system was not reached. It is important to highlight that the use of OpenMP incurs in a smaller number of processes to fully utilize the system, since 8 threads are being used per process. The system is fully utilized with only 30 processes (240 threads). Comparing this configuration with 240 processes with the MPI implementation, the MPI/OpenMP implementation is around 9 times faster, and around 20 times faster considering only the I/O time.

This increase in performance happened due to the generation of a smaller number of files (one per process). Besides, there is no intra-node concurrency in the access to the file system, because there is only one process in each machine. This gain comes also from a better memory usage by the application. Still, the time obtained for the the MPI/OpenMP implementation is almost 4 times greater than the MPI-only implementation without the use of the parallel file system.

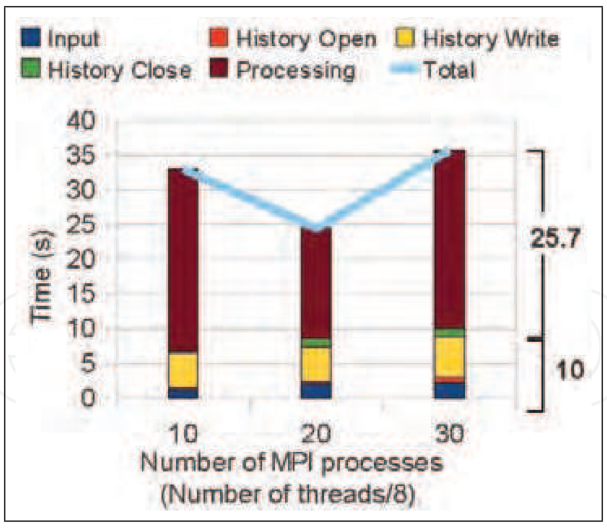


Fig. 13. Execution times for the MPI/OpenMP implementation with PVFS.

3.3 Trace visualization and close optimization for OLAM

To obtain the trace of OLAM we used the libRastro tracing library (Silva et al, 2003). LibRastro is used to generate execution traces of applications with a minimal impact on its behavior. To use the library, the source code of the target application must be modified at the points of interest in order to generate events. Beginning and end of these events are specified by two subroutine calls: `IN` and `OUT`, respectively. Each event has a name and optional parameters.

In the case of I/O operations of OLAM, the parameters are the name of the file, amount of data written/read, among others. Besides the I/O operations, we created events in all of the most important subroutines of OLAM, with the goal of identifying portions of the execution which are impaired by the I/O operations or other factors. Moreover, the detailed analysis of the application can identify the parts that do not scale.

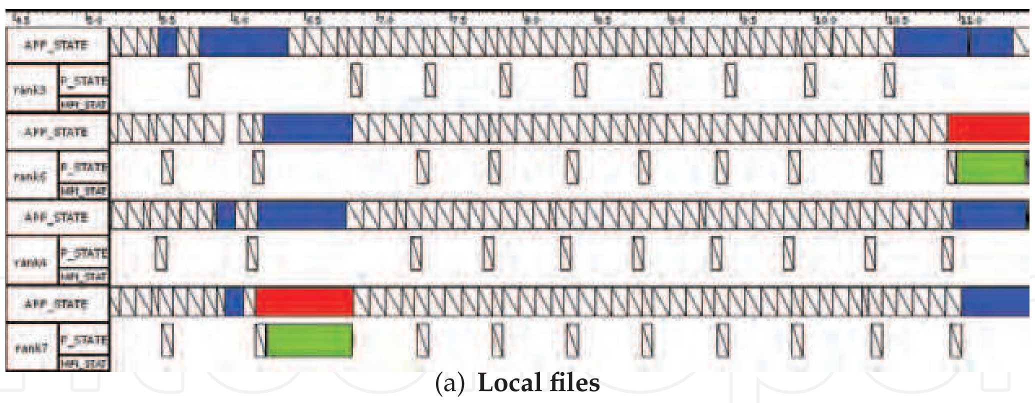
During execution with libRastro, each process of the application generates a binary trace file. These trace files must be merged and converted to a higher level language by an application-specific tool, because the semantics of the events change from one application to the other.

One high-level event description language is Pajé (Kergommeaux et al, 2000). Pajé allows the developer to describe events, states and messages between distinct containers (a container being any element that may have states, events or be source or destination of a message).

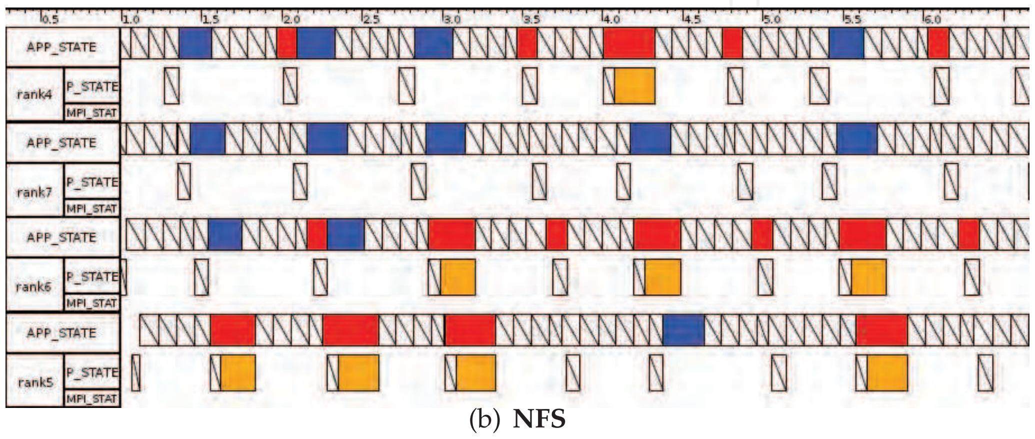
The developer has a great flexibility to create containers and the associated events in a way which best describes his code.

In our OLAM's modeling, each MPI Rank was represented by one Pajé container. The states of this container are the events obtained from the trace. Therefore, there are states inside other states (when one subroutine calls another).

Each event must be of a predefined *type*. Pajé groups events of the same type in an execution flow and automatically stacks one state inside the other as in the case of function calls. We defined the `APP_STATE` type in which we map events related to the OLAM application. There are also the `P_STATE` type, which corresponds to I/O utility functions, and the `MPI_STATE` type to which MPI events are mapped.



(a) **Local files**



(b) **NFS**

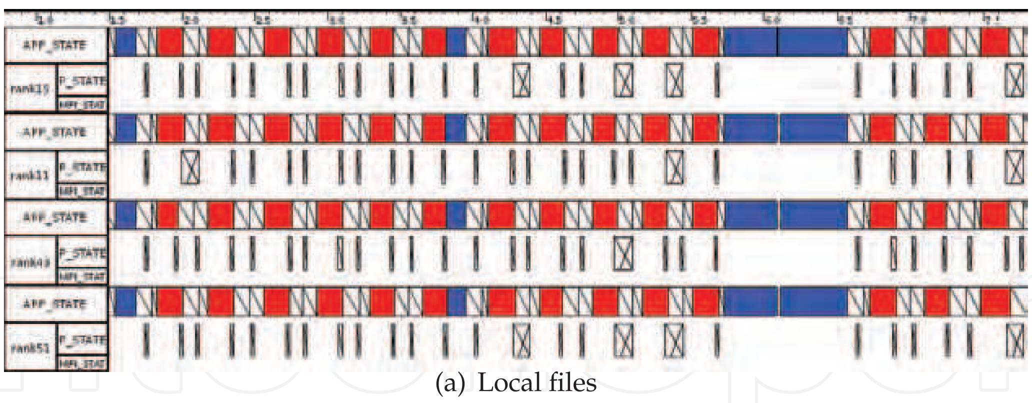
🟥 O_OUTPUT
🟦 PLOT_FIELDS
🟧 SHDF5_CLOSE
🟩 SHDF5_OREC
🟨 SHDF5_OPEN
🟦 TIMESTEP
▨ Aggregated Event

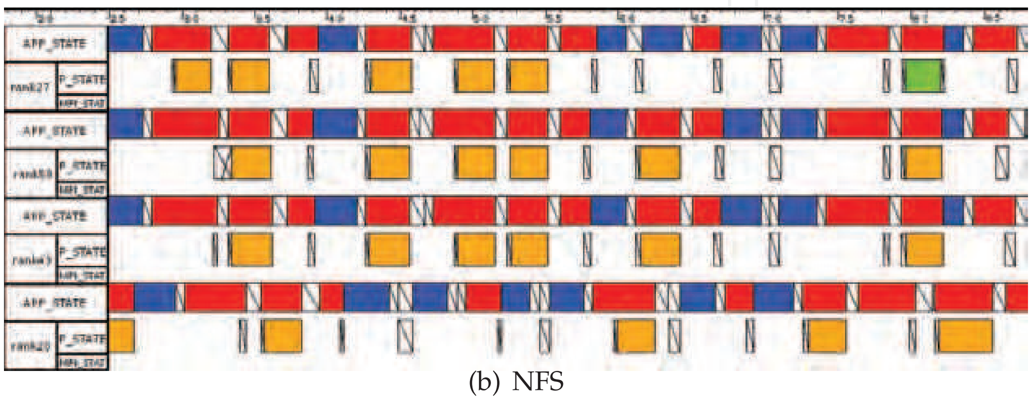
Fig. 14. OLAM execution with 8 Processes, 8 Threads each (64 cores)

The visualization was done via the Pajé Visualization Tool. It allows for a *gantt-chart* style, time based visualization of the events and states of the containers. The next section presents the results obtained.

In order to obtain traces from OLAM, we executed the instrumented version of the application on the clusters *Adonis* and *Edel* of Grid'5000. The tests were executed with 8 nodes using either the local file system or the shared NFS volume to store the execution output. We tested OLAM with and without OpenMP threads.

Figure 14 presents part of the Pajé visualization for the execution of OLAM with 8 processes, each with 8 threads, over local files and NFS. The rectangles on the left of the graph show the *APP_STATE* and *P_STATE*, as discussed in before. When the application enters the


(a) Local files


(b) NFS

🟥 O_OUTPUT

🟦 PLOT_FIELDS

🟧 SHDF5_CLOSE

🟩 SHDF5_OREC

🟨 SHDF5_OPEN

🟦 TIMESTEP

⊠ Aggregated Event

Fig. 15. OLAM execution with 64 Processes, 1 Thread each (64 cores)

OLAM_OUTPUT state, the underlying I/O functions are presented in the process container below. This can be more easily observed in Figure 14(a) at around 11 seconds: when the application enters the *O_OUPUT* state, the process calls a sequence of HDF5 helper functions, of which *SHDF5_OREC* takes the longest. This functions is responsible for writing the variables describing the atmospheric conditions to the output file of the process.

In Figure 14(a) we can see the first 9 seconds of execution for some of the processes of OLAM. In the *APP_STATE* flow, we can observe the execution of a sequence of *timesteps* (event TIMESTEP) after which the *olam_output* (event O_OUTPUT) function is called. At around 6.5 and 11 seconds of execution, we can observe that one O_OUTPUT event that takes longer to complete, something that can be observed in other parts of the execution and in other processes. The execution over NFS (Figure 14(b)) has a similar behavior, but the divergence between the normal I/O phases and the long ones is smaller. In these cases, we can observe that the function responsible for the divergence is *hdf5_close_write*, called from within *SHDF5_OREC*.

Figure 15 presents the visualization of traces generated when executing 64 processes of OLAM with no threads (classic MPI). In this case, since the data division becomes more fine-grained process-wise, the time of each timestep is smaller and there are more frequent I/O calls. Despite the overhead associated with creating more files for the same work, the performance was not penalized in the execution with local files (Figure 15(a)) due to the use of a fast local disk shared by each 8 processes. This overhead was made clear when output files are stored in the shared NFS volume (Figure 15(b)). In this case, most of the time spent in the P_STATE flow is in closing the file (SHDF5_CLOSE): small writes end up being delegated to the write-back mechanism in kernel, but the requests must be flushed by the time the file is closed.

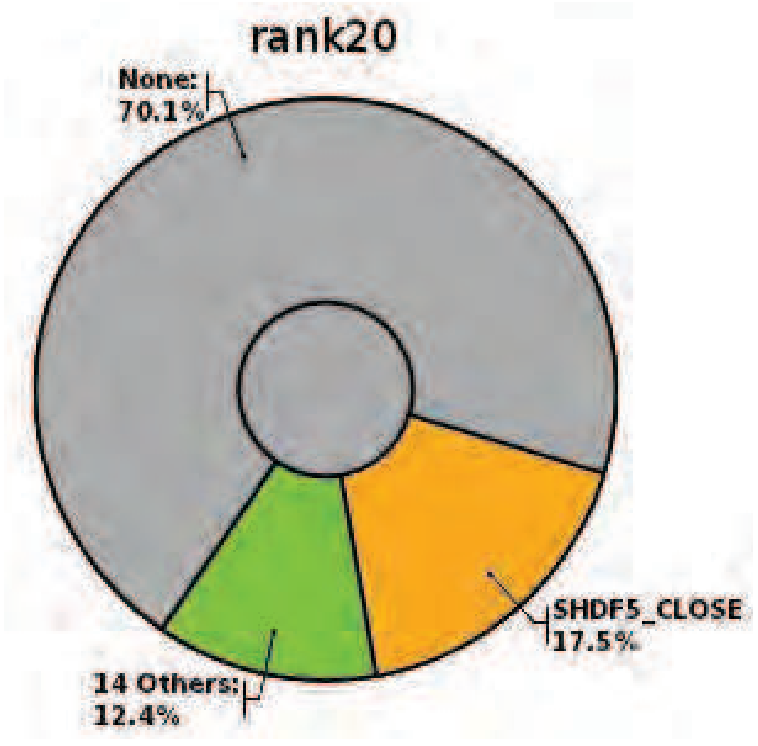
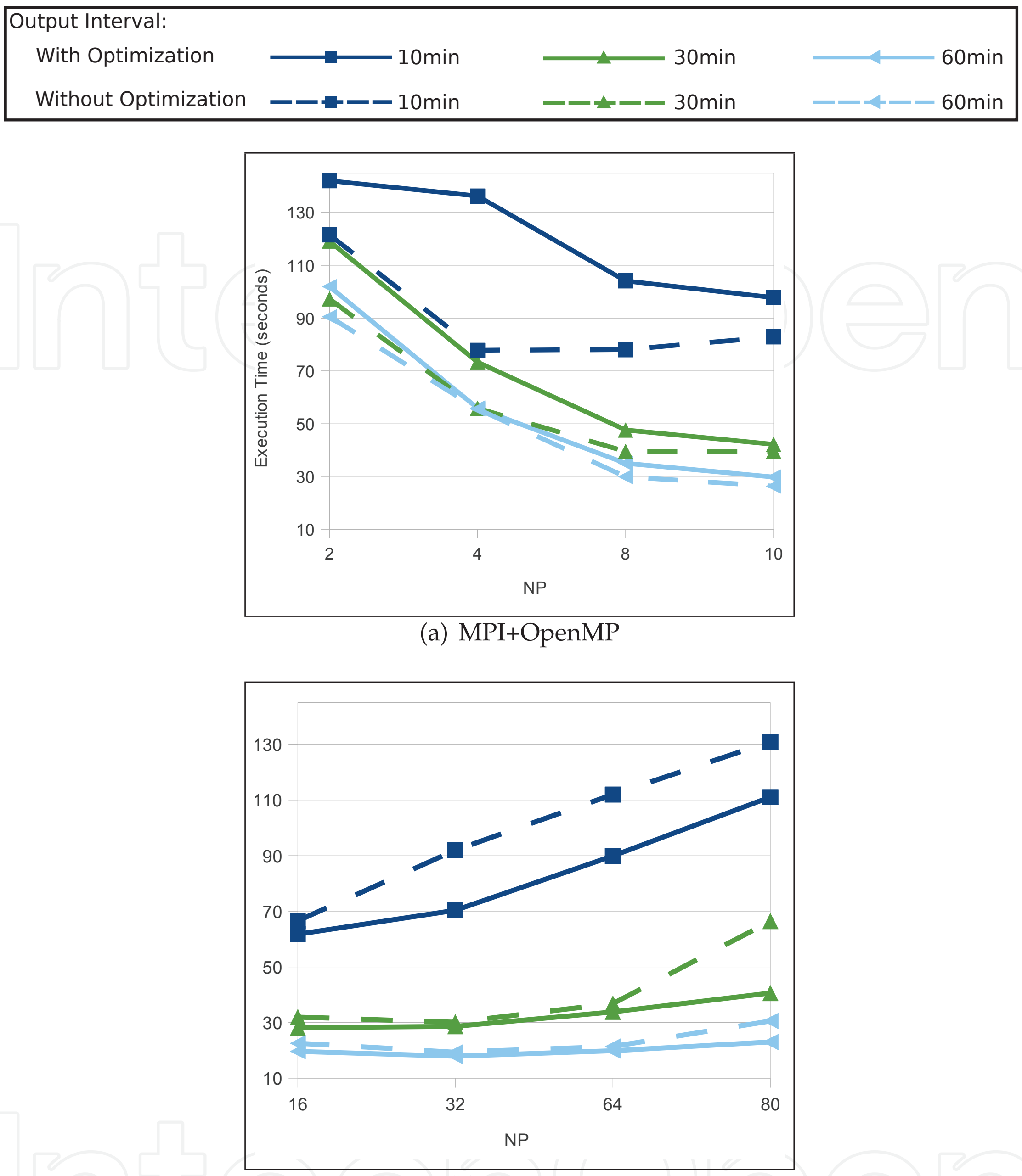


Fig. 16. Relative time spent on I/O states

Figure 16 presents the time spent on the I/O states of the application for a process from the trace of Figure 15(b). We have observed that I/O functions occupy around 25% of the execution time. Most of this time is spent on the *SHDF5_CLOSE_WRITE* function. This is due to the write-back mechanism, as said before, indicating that this is a good target to optimizations.

OLAM does not profit from the write-back mechanism due to the small size of it's I/O operations: by the time the process calls *SHDF5_CLOSE*, it's previous write requests have not been sent to the server. One way of forcing OLAM to profit from the background I/O operations offered by the file system is to change the order that the files are opened and closed. Instead of calling the close function after each process's I/O phase, the file can be closed before a new one is opened – i.e. during an *SHDF5_OPEN*.



(a) MPI+OpenMP

(b) Pure MPI

Fig. 17. Results for Close-on-Open optimization for MPI+OpenMP and pure MPI

This way, instead of having an order of events of *Computation – Open File – Write Data – Close File – Computation . . .*, the order is chained to *Computation – Close Previously Opened File – Open File – Write Data – Computation . . .*,

This change was evaluated in the *Adonis* cluster of Grid5000. We used the MPI+OpenMP version of OLAM and the pure MPI implementation, adjusting the number of OpenMP threads so that the number of cores used was the same as in the MPI version. OLAM was executed with three different output intervals: 10, 30 and 60 minutes of simulation. The smaller the intervals, the higher the number of generated files.

Figure 17 presents the execution time for this set of tests. The continuous lines are the execution times for OLAM with the proposed Close-on-open optimization, while the dashed lines represent the execution time with the standard implementation.

We have previously shown that the OpenMP implementation performs better than the pure MPI one due to the smaller amount of files to be created. Because of this, the optimization did not provide performance gains for this version. We can observe that, as the number of processes increases, the difference between the times with and without optimization decreases. We expect, therefore, an improvement in performance for larger number of files with the MPI+OpenMP version of OLAM.

With pure MPI the file system is stressed with the larger amount of files. With 80 cores the proposed optimization performed 15% better than the original one for the 10min and 37% better in the 30min interval. Gains were also seen for the 60min interval configuration. The smaller the interval, and therefore the larger the number of generated files, the sooner the gain was observed. This indicates that, as previously stated, the improvement obtained by the optimization increases with the number of generated files and, therefore, we can expect to observe these gains in the OpenMP version too (note that more processes incur in more files).

3.4 Experimental results and analysis on a single multi-core node

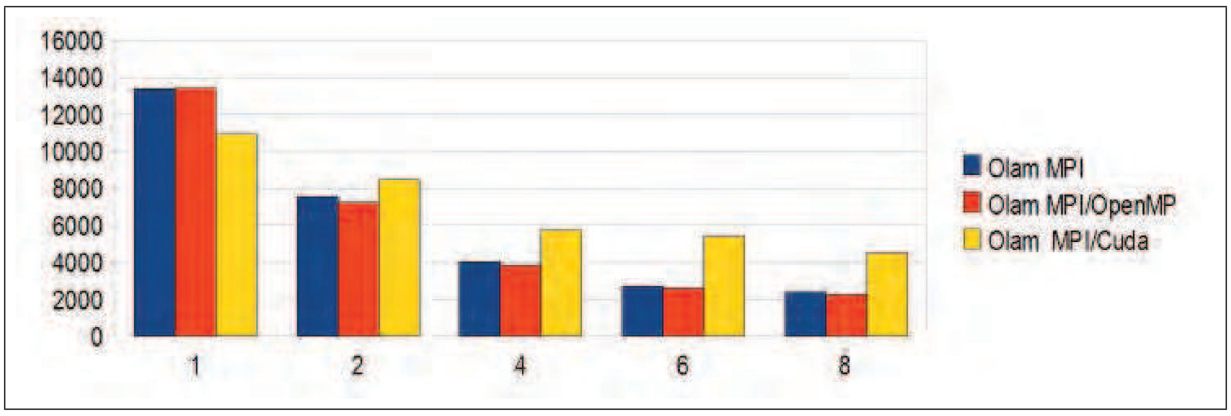


Fig. 18. Execution time for the three OLAM's implementations.

This section evaluates the three OLAM's implementations on a single multi-core node that has a GPU. Figure 18 presents OLAM's execution time in seconds as a function of cores for a resolution of 40km. The blue bar represents the MPI implementation's time, the orange bar represents the MPI/OpenMP implementation's execution time, and the yellow bar represents the MPI/CUDA implementation's execution time. This figure shows that the MPI and the MPI/OpenMP implementations perform quite similar for a single node and that the OLAM/CUDA implementation's performance decreases as the number of cores increases. The following sections discuss the results for each implementation.Figure 19 presents the three OLAMt's implementations speed-up. This figure shows that as we increase the number of cores OLAM MPI/CUDA speed-up gets worse than others implementations.

3.4.1 The MPI implementation's analysis

The MPI implementation starts one MPI process on each core. In order to explain OLAM MPI performance results we evaluate OLAM MPI performance with Vtune Performance Analyzer

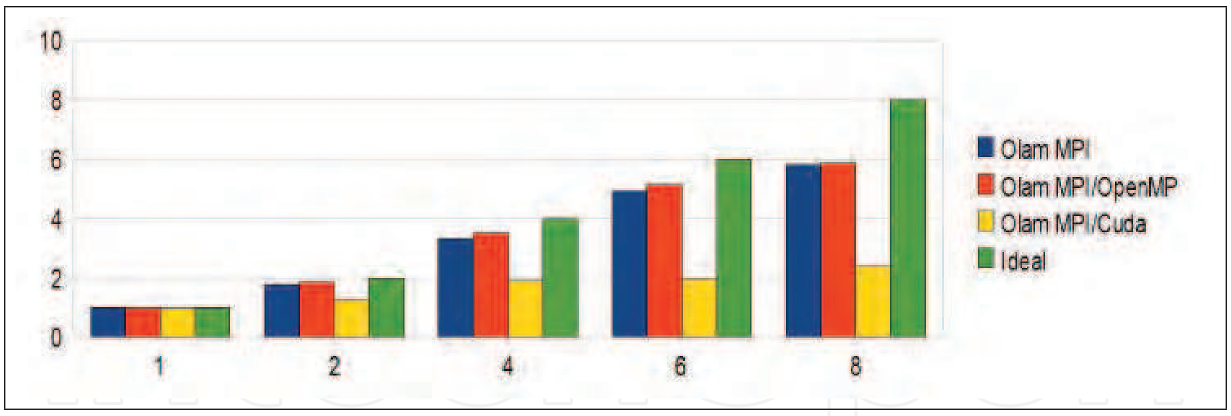


Fig. 19. Speed-up for the three OLAM's implementations.

[6]running with 8 processes on 8 cores. From previous work, (Osthoff et al., 2011a), we know that the OLAM MPI implementation scalability on 2 Quad-Core Xeon E5520 system is limited by memory contention, and that the hotspot routine presents the higher number of instruction cache miss. We observe that, for this system, the hotspot routine does not have the higher number of cache miss. Hotspot single system routine is a cpu-intensive calculation routine. Therefore, we conclude that the number of instruction cache miss has a lower performance impact in this processor architecture. Also, we observe that hotspot routine has no MPI communication overhead, therefore is a good candidate for future GPU code implementation. From Vtune analysis, we also observe that part of hotspot routines overhead are due to MPI communication routines execution time. We conclude that the speed-up is limited by, in decreasing order to: multi-core cpu processing power, multi-core memory contention and MPI communication routine overhead.

3.4.2 The hybrid MPI/OpenMP implementation's analysis

Hybrid OLAM MPI/OpenMP implementation starts one MPI process on the system, which then starts OpenMP threads on the cores of the platform. The OpenMP threads are generated on nine *do* loops from *higher number of cache miss* routine. From Vtune performance analyzer, we observed that the execution time and the number of cache misses of this routine decrease up to 50% in comparison to the MPI-only implementation. These results show that the use of OpenMP improved OLAM's memory usage for this architecture. On the other hand, we observe that CPI (Clock Per Instructions) increases from 1.2 on the MPI implementation to 2.7 on Hybrid MPI/OpenMP one, due to FORTRAN Compiler OpenMP routines overhead. We conclude OLAM MPI/OpenMP implementation still needs optimizations in order to obtain the desired speedup.

3.4.3 The hybrid MPI/CUDA implementation's analysis

The Hybrid MPI/CUDA implementation starts MPI process, and each process starts threads in the GPU device. As mention before, due to development time reasons, we implemented two CUDA kernels out of nine *higher number of cache miss* routine's *do* loops. Therefore, each MPI process starts two kernel threads on the GPU. In order to explain this implementation's results, we instrumented the two CUDA kernels to run with Computer Visual Profiler[7], which

[6] http://www.intel.com
[7] http://www.nvidia.com

Method	#Calls	GPU time (μs)	% GPU time
memcpyHtoD	1215	429677	0.83
memcpyDtoH	540	337662	0.65
kernel_loop1	135	185549	0.36
kernel_loop2	135	41151	0.08

Table 1. Execution time of the GPU kernels obtained from Compute Visual Profiler.

analyzes GPGPU systems' performance. We observed that these two loops account for up to 7% of each timestep's execution time.

First we run OLAM MPI/CUDA implementation with 1 MPI process starting 2 kernels for 5 timesteps with Compute Visual Profiler. Table 1 shows the kernels' execution times in details with 1 MPI process for 5 timesteps. These results show that GPU/memory transfer time is around three times bigger than the kernel execution time. We then conclude that the MPI/CUDA implementation needs optimizations in the transfers between GPU and main memory in order to improve performance.

A new experiment consisted in running the MPI/CUDA implementation varying the number of MPI processes from 1 to 8 (starting 2 CUDA kernels each). We observed that, as we increased the number of MPI processes, the data transfered from main memory to GPU's memory decreased from nearly 2MB for 1 MPI process to about 250KB for 8 processes. This reduction in the workload explains the MPI/CUDA implementation's poor performance for greater numbers of processes. Indeed, the gain of performance obtained using GPUs depends on the size of the problem[8]. These results explain the reason why as we increase the number of cores OLAM MPI/CUDA speed-up gets worse than others implementations.

Moreover, we observed that the GPU's utilization increased from 2% for 1 process to 6.5% for 8 processes, indicating that there is no access contention as we increase the number of processes. We plan to study this implementation for higher resolutions, obtaining heavier workloads.

4. Related work

Parallel applications scalability is the focus of several papers in the literature. In (Michalakes et al., 2008) the authors execute the high resolution *WRF* weather forecast code over 15k cores and conclude that one of the greatest bottlenecks is data storage. This same code was used in (Seelam et al., 2009) to evaluate file system caching and pre-fetching optimizations to many-core concurrent I/O, achieving improvements of 100%. I/O contention on multi-core systems is also a known issue in the literature and few strategies to mitigate the performance loss can be found. The work of (Wolfe, 2009) presents the lessons learned from porting a part of the Weather Research and Forecasting Model (WRF) to the PGI Accelerator. Like OLAM, the application used in our work, the WRF model is a large application written in FORTRAN. They ported the WSM5 ice microphysics model and measured the performance. This measurement compared the use of PGI Accelerator with a multi-core implementation and with a hand-written CUDA implementation. Finally, the work of (Govett et al., 2010) runs the weather model from the Earth System Research Laboratory (ESLR) on GPUs and relies on the CPUs for model initialization, I/O and inter-processor communication. They have shown

[8] case studies examples: http://www.culatools.com/features/performance

that the part of the code that computed dynamics of the model runs 34 times faster on a single GPU than on the CPU.

A scalability study with the NFS file system had shown that the OLAM model's performance is limited by the I/O operations (Osthoff et al., 2010). The work of (Schepke et al., 2010) presents an evaluation of OLAM MPI with VtuneAnalyzer and identifies a large amount of cache misses when using 8 cores. Then, experiments comparing the use of no distributed file system and of PVFS cleared that the scalability of these operations is not a problem when using the local disks (Boito et al., 2011). The concurrency in the access to the shared file system, the size of the requests and the large number of small files created were pointed as responsible for the bad performance with PVFS and NFS. Recent work from (Kassick et al, 2011) presented a trace-based visualization of OLAM's I/O performance and has shown that, when using the NFS File System to store the results on a multi-core cluster, most of the time spent in the output routines was spent in the close operation. Because of that, they propose to delay the close operation until the next output phase in order to increase the I/O performance. A new implementation of OLAM using MPI and OpenMP was proposed in order to reduce the intra-node concurrency and the number of generated files. Results have shown that this implementation has better performance than the only-MPI one. The I/O performance was also increased, but the scalability of these operations remains a problem. We also observe that the use of OpenMP instead of MPI inside the nodes of the cluster improves the application memory usage (Osthoff et al., 2011a).

5. Conclusion

This work evaluated the Ocean-Land-Atmosphere Model (OLAM) in multi-core environments - single multi-core node and cluster of multi-core nodes. We discussed three implementations of the model, using different levels of parallelism - MPI, OpenMP and CUDA. The implementations are: (1) a MPI implementation; (2) a Hybrid MPI/OpenMP implementation and (4) a Hybrid MPI/CUDA implementation.

We have shown that a Hybrid MPI/OpenMP implementation can improve the performance of OLAM multi-core cluster using either the local disks, a single-server shared file system (NFS) and a parallel file system (PVFS). We observe that, as we increase the number of nodes, the Hybrid MPI/OpenMP implementation performs better than the MPI one. The main reason is that the Hybrid MPI/OpenMP implementation decreases the number of output files, resulting in a better performance for I/O operations. We also confirm that OpenMP global memory advantages improve the application's memory usage in a multi-core system.

In the experiments in a single multi-core node, we observed that, as we increase the number of cores, the MPI/OpenMP implementation performs better than others implementations. The MPI/OpenMP implementation's bottleneck was observed to be due to low routine parallelism. In order to improve this implementation's speed-up, we plan to further parallelize the OLAM's most cpu-consuming routine. Finally, we observed that the MPI/CUDA implementation's performance decreases for numbers of processes greater than 2 because of the workload reduction with the parallelism. Therefore, we plan to further evaluate the performance with resolutions higher than 40km.

We also applied a trace-based performance visualization in order to better understand OLAM's I/O performance. The libRastro library was used to instrument and obtain the traces of the application. The traces were analyzed and visualized with the Pajé. We have shown that, when using the shared file system to store the results, most of the time spent in the output routines was spent in the close operation. Then, we proposed a modification to delay this operation, obtaining an increase in performance of up to 37%.

For future works we plan to further parallelize OLAM timestep routines in order to improve both the MPI/OpenMP and the MPI/CUDA implementations. Also, we plan to study OLAM's performance for higher resolutions and for unbalanced workload on single nodes and on multi-core/many-core clusters.

In order to include further I/O optimizations, we also intend to evaluate the parallel I/O library. This will be done aiming to reduce the overhead on creation of files, improving the output phases' performance.

6. References

Adcroft,C.H.A. and Marshall,J. Representation of topography by shaved cells in a height coordinate ocean model, *Monthly Weather Review, 125:2293Ű2315, 1997.*

Boito, F.Z.; Kassick, R. V.; Pilla, L.L.; Barbieri,N.; Schepke,C.; Navaux,P.; Maillard,N.; Denneulin, Y.; Osthoff,C.; Grunmann, P.; Dias, P. & Panetta,J. (2011). I/O Performance of a Large Atmospheric using PVSF, *Proceedings of Renpar20 / SympA14/ CFSE8* INRIA, Saint-Malo, France.

Cotton, W.r.; Pielke, R.; Walko, R.; Liston, G.; Tremback, C.; Harrington, J. & Jiang, H. RAMS 2000: Current status and future directions, *Meteorol. and Atmos. Phys., 82, 5-29, 2003.*

Govett, M. et al. (2010). Running the NIM Next-Generation Weather Model on GPUs, *Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melborne, Australia, pp. 729-796.

Michalakes, J., Hacker, J., Loft, R., McCracken, M. O., Snavely, A., Wright, N. J., Spelce, T., Gorda, B., & Walkup, R. WRF Nature Run., *Journal of Physics: Conference Series 125(1):012022*, URL http://stacks.iop.org/ 1742-6596/125/i=1/a=012022.

Ohta, K., Matsuba, H. & Ishikawa, Y. (2009). Improving ParallelWrite by Node-Level Request Scheduling, *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, pp. 196£203

Osthoff,C.; Schepke, C.; Panetta, J.; Grunmann, P.;Maillard, N.; Navaux, P.; Silva Dias, P.L.& Lopes, P.P. I/O Performance Evaluation on Multicore Clusters with Atmospheric Model Environment, *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing*, IEEE Computer Society, Petropolis,Brazil, pp. 49-55

Osthoff,C.; Grunmann, P.; Boito, F.; Kassick, R.; Pilla, L.; Navaux, P.; Schepke, C.; Panetta, J.; Maillard, N.& Silva Dias, P.L. Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation, *Proceedings of The 9th IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE Computer Society, Busan, Korea, pp. 69-75

Pielke, R.A. et al. (1992). A Comprehensive Meteorological Modeling System - RAMS, in: *Meteorology and Atmospheric Physics.* 49(1), pp. 69-91

Schepke, C.; Maillard, N.; Osthoff, C.; Dias, P.& Pannetta, J. Performance Evaluation of an Atmospheric Simulation Model on Multi-Core Environments, *Proceedings of the Latin American Conference on High Performance Computing* , CLCAR, Gramado, Brazil, pp. 330-332.

Seelam, S., Chung, I.H., Bauer, J., Yu, H., & Wen, H.F. Application level I/O caching on Blue Gene/P systems, *Proceedings of IEEE International Symposium on Parallel Distributed Processing*, IEEE Computer Society, Rome, Italy, pp. 1-8.

Walko, R.L.& Avissar, R. The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests. *Monthly Weather Review 136:4033-4044, 2008*.

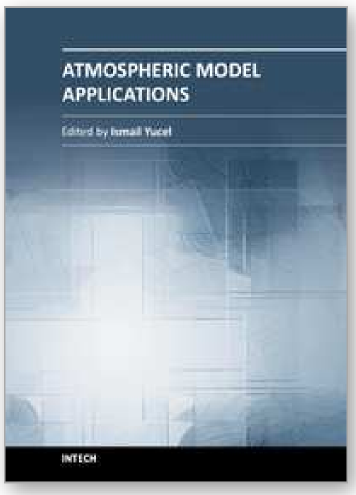
Walko, R.L.& Avissar, R. A direct method for constructing refined regions in unstructured conforming triangular-hexagonal computational grids: Application to OLAM. doi: 10.1175/MWR-D-11-00021.1 *Monthly Weather Review 139:3923-3937, 2011*.

Wolfe, M. The PGI Accelerator Programming Model on NVIDIA GPUs Part 3: Porting WRF. *In Thechnical News from Portland Group.* http://www.pgroup.com/lit/articles/insider/v1n3a1.htm

Da Silva, G.J; Schnorr, L. M. & Stein, B. O. Jrastro: A trace agent for debugging multithreaded and distributed java programs. *Computer Architecture and High Performance Computing, Symposium on*, 0:46, 2003.

Kassick, R. et al (2011). Trace-based Visualization as a Tool to Understand ApplicationsŠ I/O Performance in Multi-Core Machines, *Proceedings of of 23and International Symposium on Computer Architecture and High Performance Computing*, IEE Computer Society, Vitória, Brazil.

Kergommeaux,J.C.; Stein, B. & Bernard, P.E. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications, *Parallel Computing*, 26(10):1253 – 1274, 2000.



Atmospheric Model Applications
Edited by Dr. Ismail Yucel

ISBN 978-953-51-0488-9
Hard cover, 296 pages
Publisher InTech
Published online 04, April, 2012
Published in print edition April, 2012

This book covers comprehensive text and reference work on atmospheric models for methods of numerical modeling and important related areas of data assimilation and predictability. It incorporates various aspects of environmental computer modeling including an historical overview of the subject, approximations to land surface and atmospheric physics and dynamics, radiative transfer and applications in satellite remote sensing, and data assimilation. With individual chapters authored by eminent professionals in their respective topics, Advanced Topics in application of atmospheric models try to provide in-depth guidance on some of the key applied in atmospheric models for scientists and modelers.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Carla Osthoff, Roberto Pinto Souto, Fabrício Vilasbôas, Pablo Grunmann, Pedro L. Silva Dias, Francieli Boito, Rodrigo Kassick, Laércio Pilla, Philippe Navaux, Claudio Schepke, Nicolas Maillard, Jairo Panetta, Pedro Pais Lopes and Robert Walko (2012). Improving Atmospheric Model Performance on a Multi-Core Cluster System, Atmospheric Model Applications, Dr. Ismail Yucel (Ed.), ISBN: 978-953-51-0488-9, InTech, Available from: http://www.intechopen.com/books/atmospheric-model-applications/improving-atmosphere-model-s-performance-on-multicore-cluster-sytem-