OpenMP implementation of the horizontal diffusion method of the weather research and forecasting (WRF) model

Ronald M. Gualán-Saavedra[1], Lizandro D. Solano-Quinde[1], Brett M. Bode[2]

[1] Department of Electrical, Electronic and Telecommunications Engineering, University of Cuenca, Ecuador.

[2] National Center for Supercomputing Applications, University of Illinois, USA.

Autores para correspondencia: {ronald.gualan, lizandro.solano}@ucuenca.edu.ec

Fecha de recepción: 28 de septiembre 2015 - Fecha de aceptación: 12 de octubre 2015

ABSTRACT

This article analyzes and implements the Horizontal Diffusion method of the Weather Research and Forecast (WRF) model, using the OpenMP API to exploit the multi-core power of an Intel Xeon computer. The main scope of the study is to assess well established concepts related to scalability and cache efficiency by mean of three experiments where the use of appropriate profilers shows to be of great utility to understand computing behavior and consequently to choose some optimization approaches.

Keywords: OpenMP, Multi-core programming, Weather Research and Forecasting (WRF) model, Horizontal Diffusion method, Dynamic Solver, Intel Xeon CPU.

RESUMEN

En este artículo se analiza e implementa el método de Difusión Horizontal del modelo WRF (Weather Research and Forecast), utilizando la API OpenMP para explotar el procesamiento de equipos multi-núcleo Intel Xeon. El objetivo principal del estudio es evaluar conceptos bien establecidos relacionados con la escalabilidad y eficiencia de caché por medio de tres experimentos en los que el uso de perfiladores apropiados demuestra ser de gran utilidad para comprender el comportamiento y, en consecuencia, ayuda a elegir algunos enfoques de optimización.

Palabras clave: OpenMP, Programación multi-core, Weather research and forecasting (WRF), Método de difusión horizontal, Solver dinámico, CPU Intel xeon.

1. INTRODUCTION

Climate modeling has been fully exploiting the most recent advancements in hardware and software of HPC computing since its beginnings. This area has had and will continue having a big relevance because it aims to understand and predict the behavior of climatic phenomena that govern the planet, and consequently it supports resource management planning, disaster prevention, study of effects of climate change on society and the environment, among others.

Multi-core computing is a mature, relevant and extended technology in HPC computing. Clusters of multi-core computers are the dominant architecture in Top500.org with an overwhelming 86% over the MPP (Massively Parallel Processing) architecture with only a 14% (Top500.org, n.d.). Additionally, since the number of cores keeps increasing over generations, HPC applications are required to harness higher degrees of parallelism in order to satisfy their growing demand. Therefore, as a prerequisite for productive use of large-scale computing systems, the HPC community needs

mechanisms and tools to analyze, profile and optimize parallel applications in order to effectively and efficiently use HPC computing resources.

In this regard, OpenMP (Dagum & Enon, 1998) is the *de facto* standard for shared memory programming that allows harnessing multi-core computing on single nodes. OpenMP provides constructs for sharing work among threads, e.g., to assign the iterations of a loop to individual threads (Lorenz *et al.*, 2012). Although the use of OpenMP is relatively easy, it still requires some work to achieve an efficiently use of computing resources.

In this study, we aim to implement the Horizontal Diffusion method, i.e. a computationally intensive routine of the popular climate model WRF, using the OpenMP API. The focus of the study lies on profiling and testing over different parameters and configurations in order to experiment well-established concepts related to scaling and cache efficiency.

2. OPENMP IMPLEMENTATION OF THE HORIZONTAL DIFFUSION METHOD

The Horizontal Diffusion method is part of the Dynamic Solver of the ARW-WRF (Advanced Research WRF), and implements diffusion along coordinate surfaces formulation, which basically consists in calculating coefficients for substance distribution along coordinates (Skamarock *et al.*, 2008). The source code of this method is mainly composed of some conditionals containing a nested loop. The loop is a triple nested loop that traverses and calculates each cell of a three dimensional grid, which is a representation of a portion of the planet. Thus, the heavy processing portion of the method relies on the nested loops. Since this method is used several times per timestep during WRF simulations, it usually appears in the list of the most computationally intensive methods (Gualán-Saavedra & Solano-Quinde, 2015).

OpenMP was created to facilitate conversion of a serial (single-threaded) program to a parallel (multi-threaded) program, by identifying and demarcating sections to be parallelized with appropriate compiler directives based on different cases. For instance, there is an OpenMP construct to automatically parallelize loops. Therefore, parallelizing the loops using OpenMP is summarized in two main tasks: 1) identifying shared variables and independent variables; and 2) inserting OpenMP directives in the original source and evaluate performance using different settings until accomplish the best runtime.

In the OpenMP data model, shared variables are accessed by all threads, while private variables are copied separately for each thread, so each thread has its own version of the variable, thus avoiding concurrency issues or synchronization (Rauber & Rünger, 2013). In the case of the Horizontal Diffusion method, there are three groups of variables according to their functionality: input/output arrays, temporary variables, and index variables for loops. Most parameters are arrays of two or three dimensions, which are used as input for the arithmetic. Therefore, default configuration for variables in the OpenMP parallel region of the Horizontal Diffusion method is shared. The remaining variables are defined as private variables and need to be manually specified in the *OpenMP parallel* construct.

The directive provided by OpenMP to automatically parallelize a region comprised of a loop, is called *parallel for*, and causes the loop iterations to be distributed among the threads of the block (Rauber & Rünger, 2013). Although, not all loops can be parallelized, the Horizontal Diffusion method nested loop can be parallelized because each iteration can be performed independently of the others. A fragment of the original and the OpenMP version of the Horizontal Diffusion method are presented in Figure 1.

An important advantage of the Horizontal Diffusion nested loop, is not having synchronization requirements. The synchronization methods are computationally expensive and require special care during development and testing.

```
for (j = J(j_start); j <= J(j_end); j++) {
 for (k = K(*kts); k <= K(ktf); k++) {
  for (i = I(i_start); i <= I(i_end); i++) {
   mkrdxm = (msftx[P2(i-1,j)] / msfty[P2(i-1,j)]) * mu[P2(i-1,j)] * ...;
   // ...
   tendency(i,k,j)=tendency(i,k,j) + ...
}}}
```

(a) Original

```
#pragma omp parallel default(shared) private(j,k,i,chunk,mkrdxm,...)
{
 chunk = (J(j_end) - J(j_start) + 1) / omp_get_num_threads();
#pragma omp for schedule(static, chunk)
 for (j = J(j_start); j <= J(j_end); j++) {
  for (k = K(*kts); k <= K(ktf); k++) {
   for (i = I(i_start); i <= I(i_end); i++) {
    mkrdxm = (msftx[P2(i-1,j)] / msfty[P2(i-1,j)]) * mu[P2(i-1,j)] * ...;
    // ...
    tendency(i,k,j)=tendency(i,k,j)+ ...
 }}}
}
```

(b) OpenMP

Figure 1. Simplified code fragment for Horizontal Diffusion method**.**

3. EXPERIMENTS

Three tests were executed to assess performance when some relevant parameters and configurations are modified. The computation resources available for this study include three nodes with multi-core Intel Xeon CPUs as shown in Table 1. Each computing node has two NUMA CPUs.

Table 1. Multi-core computing nodes used for the experiments.

Node name	CPU	# CPUs	Base freq.	# Physical cores	# Logical cores
compute-0-2	Xeon(R) CPU X5650	2	2.67 GHz	12	24
compute-0-8	Xeon(R) CPU E5-2650	2	2.00 GHz	16	32
compute-0-12	Xeon(R) CPU E5-2660 v2	2	2.20 GHz	20	40

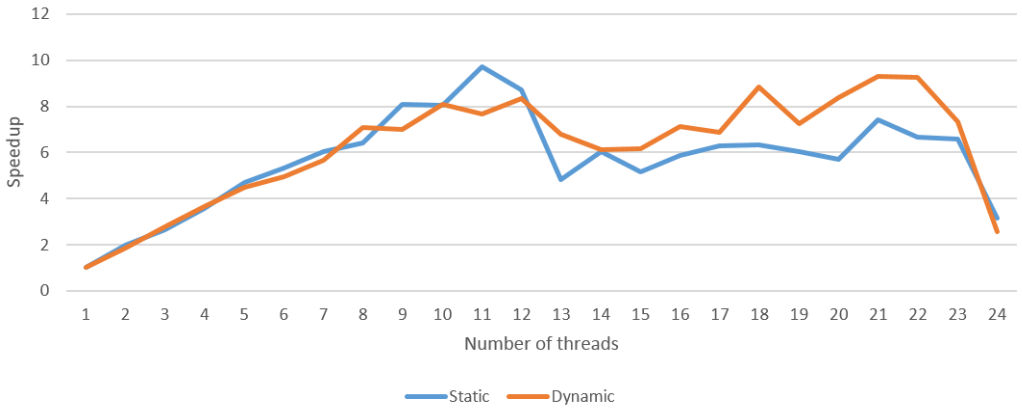
Test No 1: Assessing scheduling strategy

Scheduling strategy specifies the distribution of iterates to threads, through the *schedule* directive (Rauber & Rünger, 2013). OpenMP supports different scheduling strategies, but the most popular are static and dynamic scheduling strategies. In this experiment, we aim to assess the best scheduling option for the Horizontal Diffusion method on Intel Xeon CPUs.

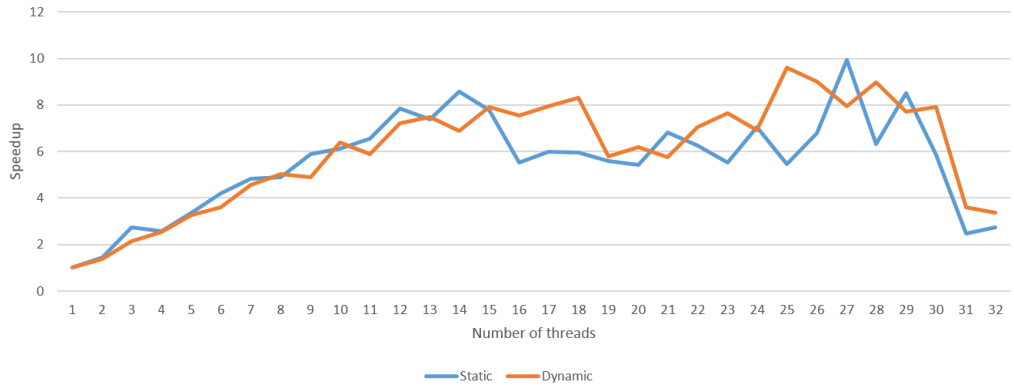
Figure 2 shows a speedup graph presenting the performance when using different number of thread and two scheduling strategies per compute node. Four observation can be drawn: (1) maximum speedup is achieved by using static scheduling; (2) speedups between static and dynamic scheduling

are alike; (3) speedup does not follow an exponential function behavior as would be expected; and, (4) speedup of the default number of threads (the maximum number of threads) obtains a speedup accounting for less than a half of the better speedup.

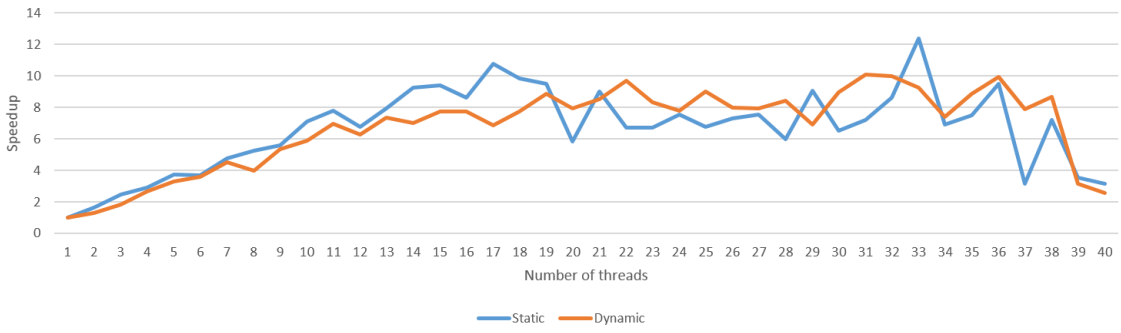
The first observation of this test is consistent with one obtained in (Gao & Schwartzentruber, 2011), where it is said that for a well load balanced case, as is the case of the Horizontal Diffusion method, static scheduling is better than dynamic scheduling due to less overhead in the OpenMP static schedule procedure.



(a) compute-0-2



(b) compute-0-8



(c) compute-0-12

Figure 2. Speedup using static and dynamic scheduling on three nodes.

The fourth observation is important because it implies that it is not recommendable to use the default number of threads for executing the Horizontal Diffusion algorithm, because it would explode only a small portion of the speedup potential.

The best speedups were obtained by using the static scheduling and they are: 9.73x, 9.94x and 12.38x for the compute-0-12, compute-0-8 and compute-0-12, respectively. The obtained speedups represent a significant improvement considering the small amount of work performed. However, the accomplished runtime is not proportional to the number of cores used. There are three major factors that avoid getting an ideal runtime parallelization: 1) overhead associated with the administration (creation, erasure) of threads; 2) synchronization mechanisms; and, 3) thread imbalance (Fürlinger, 2010). In this case, since the number of operations per iteration is relatively small, OpenMP overhead is likely to be the factor responsible for the small speedup.

Since this experiment confirmed that the statistical scheduling is the best option for the Horizontal Diffusion method, statistical scheduling is used in the remaining experiments.

Test No 2: Changing loop order

The traversing order in a multiple nested loop is important because it determines the spatial and temporal locality, i.e. the cache hit rate. Therefore, it is a relevant factor for performance behavior. Thus, in order to establish comparisons to understand the influence of the looping order in the context of multi-threaded programming with OpenMP, we tested several orders of looping and assessed the runtime. This experiment is performed on the compute-0-12 and the results are presented in Figure 3.

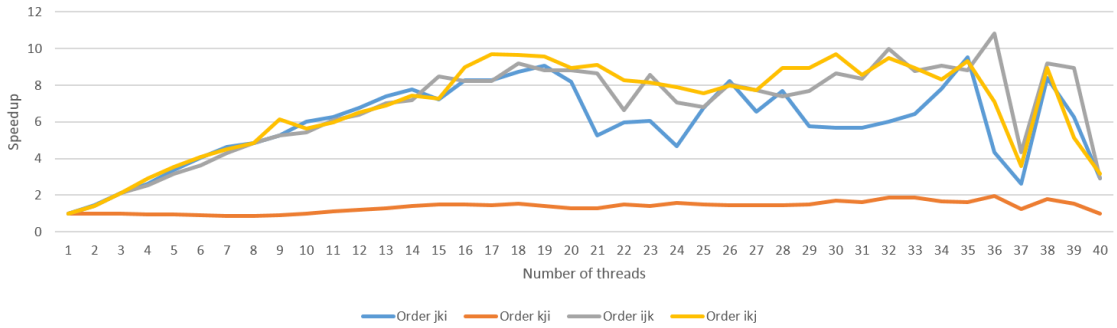


Figure 3. Speedup on compute-0-12 using several loop orders.

The maximum speedup achieved in this test was 9.52x, 1.96x, 10.81x and 9.71x in Order (j,k,i), i.e. the default order, (k,j,i), (i,j,k), and (i,k,j) respectively. The order (k,j,i) performed very poorly. The other loop orders perform alike.

This test demonstrates that using the correct order in the loop is of great importance, and confirms that the default order of the loop can be considered acceptable. A correct order increases spatial locality and consequently reduces the number of cache misses. Nowadays avoiding cache misses is considered essential in HPC development.

The *perf* tool (MediaWiki, n.d.) was used to assess cache misses and the correlation between loop order and cache misses on selected cases. The *perf* tool profiles the whole program that runs the Horizontal Diffusion method. The tasks executed by the program are: loading input variables of a use case from an input file, calling the Horizontal Diffusion method, and analyzing the output generated by the method. Since the cache metrics recorded with the *perf* tool are related to the whole program, the most important factor for this test is the rate comparing the rate of load cache misses for L1 versus the corresponding of the original loop order ($Rate/Rate_0$).

An interesting result shown in Table 2 is that for the order (k,j,i) the number of cache misses is less than the default order (j,k,i); however, the runtime is considerably greater. Further analysis is needed to understand this behavior. The other cases (i,j,k) and (i,k,j) show a considerable increase in the number of cache misses and consequently slower runtimes.

The two order cases (j,k,i) and (k,j,i) have a similar number of cache misses because they has spatial locality, since the innermost loop traverses the *i* dimension.

Table 2. L1 data cache miss rate comparison for test No 2 using *perf*.

Nested loop order (outer-inner)	Cache miss rate (R)	R/R_0
(j,k,i) (default order)	0,22%	-
(k,j,i)	0.21%	0.95
(i,j,k)	0.49%	2.22
(i,k,j)	0.78%	3.52

Test No 3: point of insertion of the OpenMP directive

This test consisted in running a number of simulations changing the location, i.e. point of insertion, where the directive OpenMP for parallelizing *for loops* (*#pragma omp for*) is included, within the three loops that make up the nested loop. Naturally, placing the directive over the outermost iteration is the initial choice for distributing the whole loop work between multiple threads. However, putting the directive over one of the inner loops causes that OpenMP fork-join behavior to be performed once per outer loops. This, as shown in Table 3, produces a loss of performance. For this experiment, compute-0-12 was used.

Table 3. Runtime comparison when inserting the OpenMP directive in a different level within the nested loop of the Horizontal Diffusion method.

Point of insertion of the OpenMp directive	Runtime (ms)
Level 1 (outer)	25.48
Level 2 (middle)	90.66
Level 3 (inner)	34.72

This test shows that the location of the *OpenMP for* directive, should be over the outer iteration, not over the inner ones. Not doing so causes a significant overhead associated with thread creation and distribution of work between them.

In order to assess the effect of the position of the *OpenMP for* within a triple nested loop, OMPP (Fürlinger, 2010) was used to analyze OpenMP overhead again. The overhead metrics for this test are shown in Table 4, where it can be seen that although overhead for Level 3 item was expected to be greater than the other levels, it resulted to be the smallest.

Table 4. Overhead comparison for the Test No 3.

Point of insertion of the OpenMp directive	Overhead (%)
Level 1 (outer)	22,05
Level 2 (middle)	58,76
Level 3 (inner)	20,89

4. CONCLUSIONS

In this study, the Horizontal Diffusion method was analyzed in the context of the OpenMP programming model as a simple and affordable method to exploit the multi-core computing power of high-performance computers. Subsequently, the Horizontal Diffusion algorithm was analyzed and the appropriate OpenMP directives were placed to run the main nested loop using multi-threaded programming. Once this was accomplished, different configurations were evaluated. Some

conclusions are drawn from the performed tests: 1) acceleration achieved using multi-threaded programming with OpenMP may be far from being proportional to the number of threads used; 2) the order of traversing a nested loop has a direct impact on the execution time due to cache miss rate; 3) when using the *omp for* in a nested loop, it should be inserted over the outer loop, otherwise they may cause a loss of performance; 4) using the maximum number of available threads to run an OpenMP program does not necessarily result in the best runtime.

In general, the use of profiling tools is of big importance to inquire internal behavior of HPC programs. The outputs or metrics obtained through profilers can be used to optimize the applications. It can be used in an iterative fashion until accomplish a good level of optimization.

A general guideline for optimizing an algorithm for using OpenMP can be summarized in the following steps: 1) Use the appropriate directives for parallelizing the chosen portion of the algorithm; 2) evaluate the two main scheduling strategies (i.e. static and dynamic) and different chunk sizes; 3) it is important to run the algorithm using several number of threads from one to the maximum allowed by the available hardware; finally, use some kind of profiler to further analyze the best set of parameterizations.

Although the use case of this paper was the Horizontal Diffusion method, the experiments and conclusions of this paper could be extended to other applications by mean of evaluation.

AKNOWLEDGMENT

This manuscript was developed in the context of the project "Integration of High Performance Computing and management of Big Data to analysis and prediction of climate in the Andean Zone of Ecuador" funded by the Research Department of the University of Cuenca (DIUC). Additionally, the authors wish to express gratitude to CEDIA for having provided access to the cluster used to perform the experiments.

REFERENCES

Dagum, L., R. Enon, 1998. OpenMP: an industry standard API for shared-memory programming. *Comput. Sci. Eng. IEEE*, 5, 46-55.

Fürlinger, K., 2010. OpenMP application profiling-state of the art and directions for the future. *Procedia Comput. Sci.*, 1, 2107-2114.

Gao, D., T.E. Schwartzentruber, 2011. Optimizations and OpenMP implementation for the direct simulation Monte Carlo method. *Comput. Fluids*, 42, 73–81.

Gualán-Saavedra, R.M., L.D. Solano-Quinde, 2015. *GPU Acceleration of the Horizontal Diffusion method in the Weather Research and Forecasting (WRF) Model*. Presented at the APCASE2015, IEEE, Quito, Ecuador.

Lorenz, D., P. Philippen, D. Schmidl, F. Wolf, 2012. *Profiling of OpenMP tasks with Score-P*. In: Parallel Processing Workshops (ICPPW), 41st Int. Conf. on, IEEE, pp. 444-453.

MediaWiki, n.d. Perf Wiki [WWW Document]. URL https://perf.wiki.kernel.org/index.php/Main_Page (accessed 5.25.15).

Rauber, T., G. Rünger, 2013. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media.

Skamarock, W., J.B. Klemp, J. Dudhia, D.O. Gill, D.M. Barker, M.G. Duda, Y.-Y. Huang, W. Wang, J.G. Powers, 2008. *A Description of the Advanced Research WRF Version 3*.

Top500.org, n.d. Home TOP500 Supercomputing Sites [WWW Document]. URL http://www.top500.org/ (accessed 9.27.12)