APCM: An Auto-Parallellism Computational Model

Increasing the performance of MPI applications in multi-core environments

André Luiz Lima da Costa, Josemar Rodrigues de Souza
Supercomputing Center for Industrial Innovation
(*Centro de Supercomputação para Inovação Industrial*) – CSII
SENAI CIMATEC
Salvador, Brazil
andrelc@fieb.org.br, josemar@fieb.org.br

Abstract—Given the availability of computer clusters based on multi-core processors, the hybrid programming model has become an important ally of high-performance computing users in improving the performance of their parallel applications. However, creating hybrid applications is a complex task because it requires developers to be familiar with two distinct parallel programming models. Against this background, this article introduces APCM, an auto-parallelism computational model. APCM's goal is to create hybrid parallel applications, i.e., OpenMP (memory programming) and a message-passing interface (MPI), from MPI applications. This goal is achieved in a simple, automated manner that is transparent for the user while increasing application performance. In the article's conclusion, we present consistent results that attest the efficacy of the proposed model.

Keywords—parallel programming; hybrid model; MPI; OpenMP; auto-parallelism

I. INTRODUCTION

There are two main reasons why hybrid programming models have been drawing increasing attention from programmers and researchers in the last few years: (a) a bundled message-processing interface (MPI) [1] and OpenMP (memory programming) [2] is an established commercial product supplied by several compiler vendors that facilitates the integration of these two platforms and improves their joint performance; and (b) supercomputers are better suited to running hybrid applications because they now consist of a number of multi-core machines, which increases their processing power [3].

Several studies [3][4][5][6] demonstrate the benefits of choosing the hybrid programming model, which uses the advantages of both programming models to provide the desired application performance increase compared with non-hybrid parallel applications. However, to parallelize an application that uses two distinct concepts, such as MPI (distributed memory programming) and OpenMP (shared memory programming), is difficult because the developer must have a command of both programming models to obtain satisfactory results from the hybrid version.

In this context, the present study aims to present a computational model capable of increasing the performance of parallel algorithms using a web application, which promotes automated parallelism using OpenMP to an already parallel algorithm previously written in C using MPI. Thus, this study creates a new hybrid algorithm (MPI + OpenMP) in a transparent manner for the developer and facilitates the increased performance of the application, which has a simple and intuitive interface that does not require the MPI developer to know OpenMP to create the hybrid algorithm.

II. RELATED STUDIES

During the elaboration of this study, several studies that refer to automated code-generation were found, of which [7][8][9][10] were conspicuous because they possessed an overarching goal: transforming a serial algorithm into a new, parallel version. As in the present article, most of these studies start from a C program and add parallelism using OpenMP. The difference between these studies and the model presented here is that instead of starting from a serial program, this model starts from an already parallel algorithm based on MPI and yields a new, hybrid version after adding OpenMP to it. Several important points were observed in the analysis of these previous studies, which are common to them all. These points were used as benchmarks for the present study:

- A focus on the **parallelization of loops** because these segments result in less complexity when introducing parallelism in the algorithm;
- Using **OpenMP** to achieve parallelism because it is a simpler language than, e.g., MPI, PVM and open computing language (OpenCL);
- **Steps taken** to achieve parallelism, as observed in the related works, are: parallelizable loop identification, OpenMP directive configuration and parallel code insertion.

III. APCM

A. The Model

The goal of the auto-parallelism computational model (APCM) is to increase the performance of parallel applications that use MPI in multi-core environments. This goal is achieved through the automated creation of shared memory parallelism using the OpenMP standard library. APCM transforms MPI algorithms into a hybrid version using MPI and OpenMP. In addition to exploiting the parallelism of dividing work among the machines in the cluster, the new algorithm can exploit the parallelism provided by OpenMP, which divides the work sent to each node among its respective cores.

Because the process of creating the new hybrid algorithm does not require much processing power, APCM was implemented as a web tool, more specifically as a service available in the cloud, where researchers from around the world can use it without having to download, install and configure anything in their clusters, as is the case in the studies cited under "Related Studies".

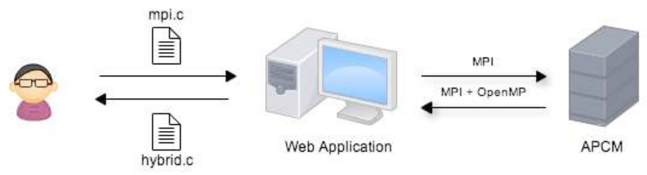


Fig. 1. APCM use flow.

Figure 1 illustrates how users interact with APCM. Users access the model through a web application, which is available on the Internet, whereby they upload the MPI application that they want to modify. Then, the web application sends the application to be converted into a new, hybrid algorithm that once completed is sent back to the user, ready to be executed in multi-core environments. Thus, parallelism is transparently achieved without the user being required to know OpenMP, which enables the user to focus only on the MPI programming.

During the transformation process of an MPI algorithm into a hybrid algorithm, APCM performs several well-defined steps that have been validated in similar projects. Figure 2 shows the flow of the 5 macro-steps undertaken by APCM. Each step is started only after the previous step is completed.

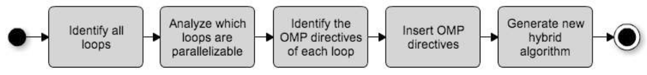

Identify all loops
Analyze which loops are parallelizable
Identify the OMP directives of each loop
Insert OMP directives
Generate new hybrid algorithm

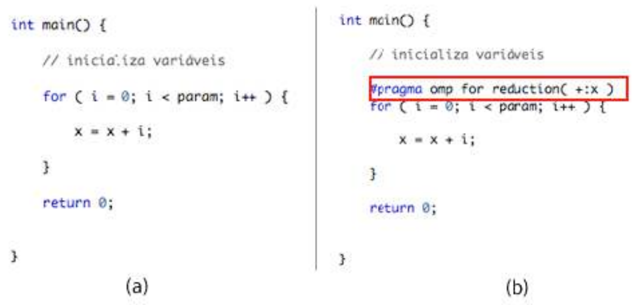
Fig. 2. Steps taken to implement parallelism

The details of each step leading to parallel code generation, as shown in Figure 2, are as follows:

- **Identify all loops**: This is the first step, during which the application sweeps the entire submitted algorithm, mapping all *for* loops. This step is highly important because parallelism will be inserted into iterative loops, which are the sections of the algorithm in which automated parallelism exploitation is achieved at a lower complexity cost and the sections usually preferred when parallelization is performed manually;
- **Analyze which loops are parallelizable:** After identifying the loops, the application checks which of them can be parallelized. Thus, it checks whether the loop contains any variables that depend on previous iterations, whether it executes any MPI command (e.g., MPI_Bcast(), MPI_Send(), MPI_Recv()) or if a loop is contained in another loop;
- **Identify the OMP directives of each loop:** In this step, the application identifies which OpenMP configuration is best for each parallelizable loop in the algorithm. Thus, it determines which clauses of OpenMP (*reduction, private, shared, schedule*, etc.) are best suited to each loop to extract the maximum performance from each of them;
- **Insert OMP directives:** After identifying the directives of each loop, the application inserts the OpenMP code into the algorithm sent by the user without changing any part of the originally submitted code;
- **Generate new hybrid algorithm:** Finally, the application generates the new, hybrid version of the algorithm in a new file, which is available for the user to download and use.

Because it is a model as opposed to a framework or compiler, APCM assumes that the user will abide by certain pre-requisites so that the algorithm is correctly converted and that the hybrid version may potentially increase the performance of the user's application. The model requires the following:

1. The algorithm must be written in the C programming language, stored in a file with the .c extension and have no syntax errors;
2. The parallel application should use only MPI;
3. The algorithm should use the concept of matrix multiplication and the master/worker model. APCM has only been validated for applications with such a configuration, and its efficiency is not guaranteed for algorithms that deviate from this pattern;
4. It is strictly necessary that the curly braces ("{") be opened on the same line as the function to which they pertain, never on the line below;



```
int main() {

    // inicia.iza varidveis

    for ( i = 0; i < param; i++ ) {

        x = x + i;

    }

    return 0;


}
```

(a)

```
int main() {

    // inicializa varidveis

    #pragma omp for reduction( +:x )
    for ( i = 0; i < param; i++ ) {

        x = x + i;

    }

    return 0;


}
```

(b)

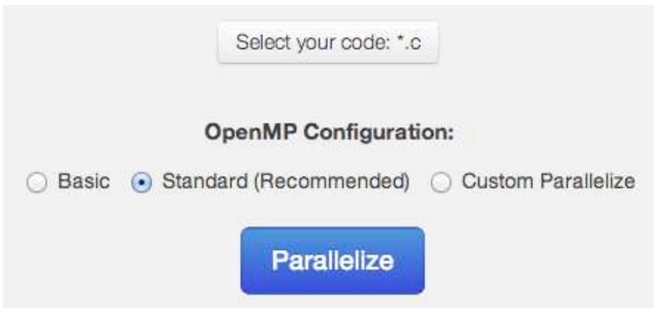
Fig. 3. Example of algorithm parallelization process.

Figure 3 shows a simplified example of a transformation performed by APCM. Session (a) shows an algorithm written in C, which satisfies all model requirements. Session (b) shows the same algorithm as in (a) with the added OpenMP code and no changes to the original algorithm.

B. The Application

APCM was implemented as a web application and written in the PHP[1] programming language using the *Bootstrap*[2] front-end framework. The application is available at *http://andrecosta.info/apcm/*.

The application enables the user to obtain the hybrid code in only two clicks, one to select the MPI algorithm and another to generate parallelism.


Select your code: *.c

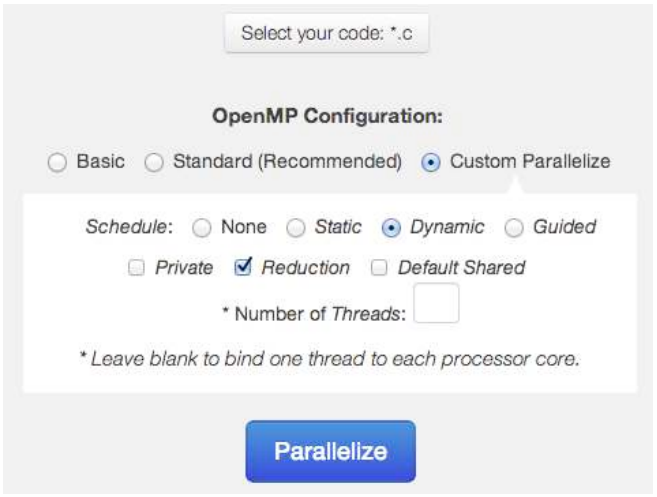
OpenMP Configuration:

Basic [✓ Standard (Recommended)] Custom Parallelize

Parallelize

Fig. 4. APCM application.

As shown in Figure 4, the application enables the user to choose the best configuration from three options, with which OpenMP will be embedded. Users without knowledge of OpenMP can simply leave the default “Standard” option selected. Users who with a certain degree of knowledge about shared memory programming may find the other two options useful. “Basic” uses only a minimal set of directives to parallelize, and “Custom” enables the user to choose the clauses that best suit the submitted algorithm.


Select your code: *.c

OpenMP Configuration:

Basic Standard (Recommended) [✓ Custom Parallelize]

Schedule: None *Static* [✓ *Dynamic*] *Guided*

Private [✓ *Reduction*] *Default Shared*

* Number of *Threads*:

* *Leave blank to bind one thread to each processor core.*

Parallelize

Fig. 5. APCM application with Custom options active.

Selecting the “Custom” option, as shown in Figure 5, enables the user to configure OpenMP according to his or her needs. The clauses that can be configured in APCM are as follows:

- ***Schedule***: defines process scheduling among threads. Options are: *static, dynamic* and *guided*;
- ***Private:*** defines private variables in iterative loops. In APCM’s case, private variables are used as loop iteration counters;
- ***Reduction:*** highlights in memory the variables that are incremented or decremented in each loop iteration;
- ***Default Shared:*** defines all variables as global as default, with the exception of private variables;
- ***Number of threads:*** indicates the number of threads to be created by OpenMP. If this field is left blank, OpenMP will try to identify the number of cores in the machine it is running and create one thread for each core.

A series of tests was performed to arrive at the standard configuration available in the APCM web application in which the clauses available in the application were compared with one another. Thus, eight algorithms with different OpenMP configurations were prepared: basic configuration without clauses and no definition of the number of threads (so-called pure OMP), using only static scheduling, using only dynamic scheduling, using only guided scheduling, using only private defaults, using only reduction defaults, using only shared defaults and, finally, using a different number of threads (in this case, four). All of the algorithms had the same base, i.e., a 10,000 x 10,000 matrix multiplication application using MPI on four processes in the master/worker model, the only difference being their respective OpenMP directives.

TABLE I. OMP CLAUSE PERFORMANCE

! "#$%&#( )'* (+ , - . - &0"'1"#&#'2'3'#'4(35#6			
! "#$%&(	***) *$*&(***	***+, &. - (***	***/ 0 1(***
! "#$%##&	$"' (%) *&	$ (' +%, "&	+*+"%+) &
, **) , (&	+*) (%, $&	$"$"%(*&	+* (#%((&
, *) $-+#&	+! (, %"+&	++! #%*+&	+*' ' %' ' &
, ! *, %+) &	+#! *%*) &	+++, %+) &	+! +, %" "&
, ! +' %') &	+# () %#' &	+++) %) ' &	+) "+%(#&
12&3$*. (	***4. - , ' *&5#(***	***) 6$2 - (***	***7(*62 $- 8(***
($$*%(*&	$! "+%(, &	+! $*%(' &	($*#%' (&
($+"%*! &	$, +$%# (&	++* (%) #&	(##*%+, &
($) ' %$! &	$, +"%$"&	++") %+"&	(# (! %##&
() $*%$#&	$$) , %++&	+)) , %*+&	(($#%*' &
((' $%!) &	$+" "%$+&	+' #' %(#&	' +#) %+) &

The comparative tests were performed five times (Table I). The highest and lowest values were excluded, and the remaining three were averaged. Thus, a consistent result was obtained. Tests were performed in the multi-core cluster of the Computational Modeling Laboratory (Laboratório de Modelagem Computacional) of SENAI CIMATEC (Serviço

1 PHP – Hypertext Preprocessor: http://php.net.
2 Bootstrap: http://twitter.github.io/bootstrap/.

Nacional de Aprendizagem Industrial, Centro Integrado de Manufatura e Tecnologia – Integrated/Unified Center for Manufacture and Technology, National Service of Industrial Qualification), which consists of the following configuration: eight HP ProLiant DL120 G6 Intel Xeon Quad-core X3440 2.53 GHZ HyperThreading processors, 8 GB RAM, 2 TB storage and a Linux kernel 2.6 X86 64 GNU Ubuntu 10.10 operating system, which were interconnected by a D-Link Des-1024D 10/100 Fast Ethernet switch. Because the tests were comparative, the deficient communication between the nodes, which resulted from the low switch speed, did not hinder or influence the results.

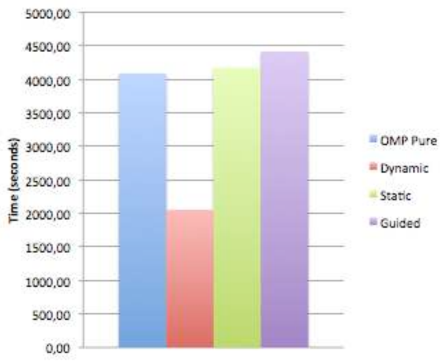


Fig. 6. Pure OpenMP x *Schedule* (*dynamic*, *static* and *guided*).

Figure 6 summarizes the results of the comparisons between the Pure OMP algorithm and the versions using the clauses *schedule static, schedule dynamics* and *schedule guided*. Of the three options for *schedule*, the only option that performed better than the pure version of OpenMP was the *dynamic* clause.

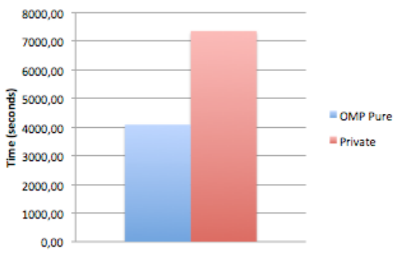


Fig. 7. Pure OpenMP x *Private*.

Using the *private* clause in the algorithm being tested increased its processing time compared with Pure OMP (Figure 7).

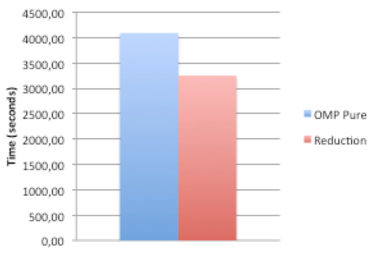


Fig. 8. Pure OpenMP x *Reduction*.

The comparison between Pure OMP and the algorithm version using the *reduction* OpenMP clause is shown in Figure 8. Using this clause decreases execution time and consequently improves the application performance.

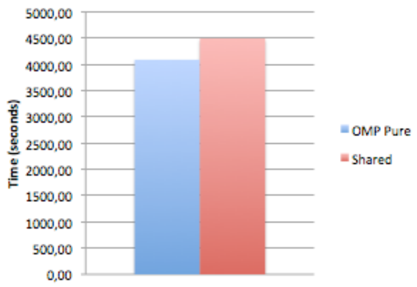


Fig. 9. Pure OpenMP x *Shared*.

Figure 9 shows the loss of performance by the algorithm using the *shared* clause as standard for loop variables. Pure OMP obtained better execution times.

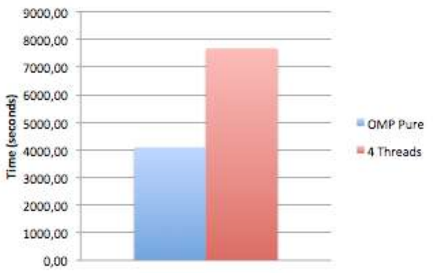


Fig. 10. Pure OpenMP x 4 *Threads*.

In conclusion, Figure 10 shows the comparison between Pure OMP using eight threads with the version of the algorithm configured to use only four threads, where again the basic version of OpenMP demonstrates the best performance. The option with four threads was used because the nodes in the computer clusters are quad-cores, i.e., they have four physical cores. However, these processors are built with HyperThreading technology, which simulates twice as many cores. Thus, the processors possess four virtual cores, and for this reason, Pure OpenMP used eight threads as default.

From the results here presented, it may be concluded that for the algorithm for matrix multiplication that was tested, which used a master/worker implementation with MPI, the OpenMP configuration that provides best performance is the configuration with the *reduction* and *schedule(dynamic)* in its compiler directives. Based on these results, this configuration was chosen as "Standard" for APCM because it was deemed to be the best option for users with no knowledge of OpenMP.

IV. Analysis of Results

To analyze the efficiency of APCM, we compare the average execution times of the pure MPI version with those of the hybrid version generated by this model. Tests were run on up to eight machines, always with one MPI process per node, in the same multi-core cluster from SENAI CIMATEC and using the same matrix multiplication algorithm from the previous tests, with a 10,000 x 10,000 matrix.

TABLE II. APCM PERFORMANCE

789./5': ".#(3';'<$."'= <>': ".#(3'			
125'. 88(95, #*(	: 3. 2$; . (<&%. (=). '5#- 8>(		?%@253. %. #*(=A>(
	B"C2&(	1, 2 (0 1?(	
!&	+) ! (-$'&	, !, *, -!$&	, !-$!&
,&	,) $ (-' "&	!* (+ (-+"&	, $-#!&
$&	! (#) -) !&	(, $+-) !&	, +-+*&
+&	!+$) -' '&	) +' *-' (&	, #-, *&
)&	!, !) -+#&	++, (-" (&	, (-+) &
#&	!*,) -$#&	$ (, (-! *&	, (-) !&
(&	", , -("&	$, +$-, *&	, ' -+) &
'&	') "-#, &	, ' #' -,) &	, "-" (&

Table II shows the average times of each version per number of parallel processes, with the percent performance gain of the hybrid version in relation to the original version. These data indicate the increase in performance through the use of APCM. Averaging the performance gain for different numbers of processes, we arrive at an average gain of 26,82%. That is, using APCM with its standard configuration yielded a 27% performance increase over the original MPI application, which is an impressive figure for high-performance computing applications.

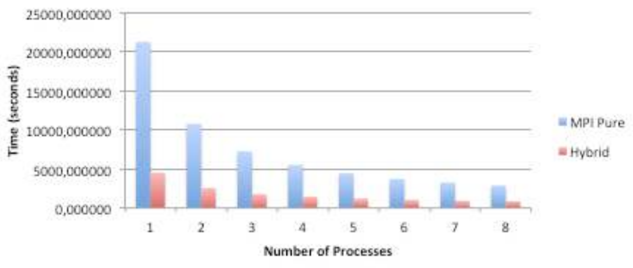


Fig. 11. Comparison of Results of Pure MPI x Hybrid.

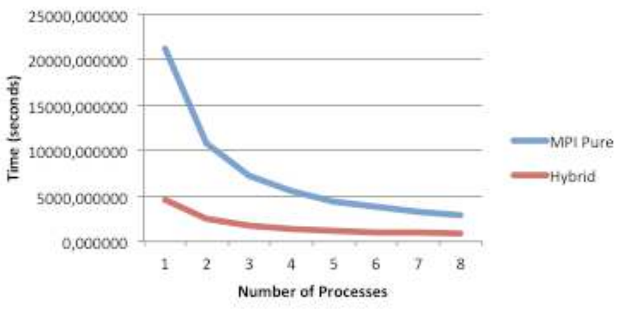


Fig. 12. Comparison of Results of Pure MPI x Hybrid.

Figures 11 and 12 show the advantage of opting for a shared memory programming model (OpenMP) in addition to the distributed memory programming model (MPI) in multi-core clusters, as initially presented by [3][4][5][6], where the processing time of the hybrid algorithm is shorter than the MPI version; i.e., it performs better.

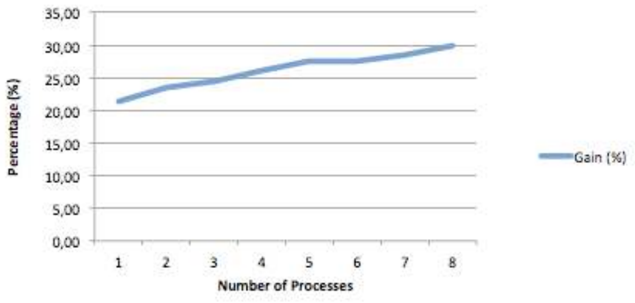


Fig. 13. Performance gain using APCM.

The performance gain obtained through APCM tends to increase with the number of nodes in which the application runs (Figure 12), which means that the solution, in addition to being efficient, is also scalable.

V. FINAL REMARKS

Based on what was presented in the preceding sections of this article, the goals originally set for the auto-parallelism computational model have been achieved, given that using APCM made possible a 27% increase in an application's performance.

To achieve this outcome, the model presented here was implemented through a web application that in an automated process that is transparent for the user implements shared memory parallelism, which creates a hybrid algorithm on top of the MPI algorithm submitted by the user.

The results analyzed herein were obtained from a matrix multiplication application in a multi-core cluster, which satisfied all model requirements. It is uncertain whether such performance gain will be achieved in MPI applications using the model in a scenario other than the one presented.

In conclusion, APCM has proved to be an effective tool for the optimization of parallel algorithms and thus relevant in the field of high-performance computing.

REFERENCES

[1] MPI. Message Passing Interface Forum. 1994. Available from: <www.mpi-forum.org>. Accessed on: 12 Apr 2013.

[2] OPENMPI. Open Source High Performance Computing. Available from: <www.open-mpi.org>. Accessed on: 12 Apr 2013.

[3] Lusk, E., Chan, A. Early Experiments with the OpenMP/MPI Hybrid Programming Model. IWOMP'08 Proceedings of the 4th international conference on OpenMP in a new era of parallelism. p. 37-47. 2008. ISBN:3-540-79560-X 978-3-540-79560-5.

[4] Osthoff, C., Grunmann, P., Boito, F., Kassick, R., Pilla, L., Navaux, P., Schepke, C., Panetta, J., Maillard, N., Dias, P. L. S., Walko, R. Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation. Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on.

[5] Dong Li, Supinski, B.R., Schulz, M., Cameron, K., Nikolopoulos, D. S. Hybrid MPI/OpenMP power-aware computing. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.

[6] Rabenseifner, R., Hager, G., Jost, G. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on.

[7] Torquati, M., Vanneschi, M., Amini, M., Guelton, S., Keryell, R., Lanore, V., Pasquier, F. X., Barreteau, M., Barrère, R., Petrisor, C. T., Lenormand, E., Cantini C., Stefani. F. An innovative compilation tool-chain for embedded multi-core architectures. in Embedded World Conference 2012. Nuremberg, Germany, 2/2012.

[8] Grosser, T., Zheng, H., Allor, R., Simburger, A., Groblinger, A., Pouchet, L. N. Polly – Polyhedral Optimization in LLVM. In Christophe Alias and Cédric Bastoul, editors, Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). INRIA Grenoble Rhône-Alpes, April 2011.

[9] Ragheshi, A. A Framework for Automatic OpenMP Code Generation. Dissertação de Mestrado. Department of Computer Science and Engineering, Indian Institute of Technology, Madras. 2011.

[10] Dave, C., Bae, H, Min, S. J., Lee, S., Eigermann, R., Midkiff, S.. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. IEEE Computer, vol. 42, no. 12, pp 36-42, Dec. 2009.