2010

Towards A Quasi High Level Compiler Comparative and Attributive Model for OpenMP Programs

Mohammed F. Mokbel
University of Windsor

Towards A Quasi High Level Compiler Comparative and Attributive Model for OpenMP Programs

by

Mohammed F. Mokbel

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2010

Your file Votre référence
ISBN: 978-0-494-80240-3
Our file Notre référence
ISBN: 978-0-494-80240-3

Declaration of Co-Authorship / Previous Publication

I. Co-Authorship Declaration

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Mr. Michael Wong. The collaboration is covered in Chapter 4 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author, and the contribution of co-authors was primarily through the provision of constructive criticism.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes 1 original paper that has been previously published/submitted for publication in peer reviewed journals, as follows.

Thesis Chapter	Publication title/full citation	Publication status*
Chapter 4	**Mokbel, M. F., Kent, R. D. and Wong, M.** (2010) An Abstract Semantically Rich Compiler Collocative and Interpretative Model for OpenMP Programs. The Computer Journal.	Submitted

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

In order to understand the behavior of OpenMP programs, special tools and adaptive techniques are needed for performance analysis. However, these tools provide low level profile information at the assembly and functions boundaries via instrumentation at the binary or code level, which are very hard to interpret. Hence, this thesis proposes a new model for OpenMP enabled compilers that assesses the performance differences in well defined formulations by dividing OpenMP program conditions into four distinct states which account for all the possible cases that an OpenMP program can take. An improved version of the standard performance metrics is proposed: speedup, overhead and efficiency based on the model categorization that is state's aware. Moreover, an algorithmic approach to find patterns between OpenMP compilers is proposed which is verified along with the model formulations experimentally. Finally, the thesis reveals the mathematical model behind the optimum performance for any OpenMP program.

DEDICATION

To

The unconscious, unknown of me, the source

My family for their endless support and encouragement

My mother Amina and brother Ali

...

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors, Professor Robert Kent and Professor Akshaikumar Aggarwal Their guidance and encouragement throughout my graduate studies helped me a lot when I needed it the most. Furthermore, I'm grateful to my thesis advisor, Professor Robert Kent for encouraging me to do my best while I was philosophizing, and giving me the choice to research the topics that I like His guidance, benevolence and comradeship were indispensable for me to complete my thesis work successfully.

I would also like to thank my committee members for their helpful comments and suggestions. I would like to thank Zina Ibrahim for correcting all the spelling errors in this work.

I would like to thank the whole RL C++ FrontEnd & Runtime Development team at IBM Toronto Lab for the fantastic 8 months I spent with them as a master co-op student Special thanks to my mentor Christopher Cambly who provided all the support and directions while working on the compiler performance analysis tasks. Special thanks to Michael Wong from IBM Toronto Lab who read the paper and provided valuable comments.

Thanks to many bands and musicians, their music helped me a lot while I was working on my thesis Among them are: Antonio Vivaldi specially The Four Seasons violin concertos and more precisely Winter season, Ludwig van Beethoven, Johann Strauss, Amadeus Mozart, Johann Sebastian Bach, Placebo, Coldplay, Era, The Killers and Travis.

Lastly but not least, thanks to Walter Isaacson for writing a great book about Albert Einstein: His Life and Universe. My Saturday's and Sunday's nights were just another dimension on Einstein's life, the greatest physicist the humanity ever have.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

"Give me extension and motion and I will construct the universe"
- Rene Descartes

INTRODUCTION

OpenMP [1] is a widely accepted Application Programming Interface API for shared memory parallel programming architecture It consists of a set of compiler directives, runtime library routines and environment variables OpenMP expands C/C++ and Fortran 77/90 languages to comprise additional denotative parallel semantics For OpenMP to work, it has to be supported by the compiler runtime system, because the code is introduced with very high level constructs that depend on the compiler low level conversion to multithreaded code Two of the most important features of OpenMP are incremental parallelism that often does not require any changes in the original source code to be parallelized with OpenMP and sequential equivalence, which preserves the consistency in the results between one and multiple threads code

The performance of OpenMP programs is closely coupled with the underlying environment, it is the hardware architecture and software optimization that take a major part in determining the performance of the program Processor architecture, pipelining, memory speed and bus interconnect bandwidth and the levels of cache all contribute to the overall performance of the program Commonly, for OpenMP programs performance to scale, the sequential version should be coded efficiently and resourcefully so that the conversion to OpenMP is not affected by the inefficiency of the serial version of the code Thus, the scalability of OpenMP application is bounded by the efficient design of the original sequential source code

For a thorough performance analysis of OpenMP programs, specially designed tools are needed to understand the internal behavior of the application Intel VTune, Intel Thread Profiler [2], Intel Parallel Studio [3] and others, are the kind of tools which provide low level profile information that

can help in determining the real causes of the performance problems. Some of these tools (e.g. VTune) rely on the hardware performance counters that are architecture specific for performance analysis and tuning. However, mastering these tools is not an easy task and requires a deep knowledge of each tool with respect to OpenMP performance.

In [4], Tian *et al* clearly state the complexity of conducting a fair comparison between commercial product compilers that support OpenMP. Since these compilers do not publish their internal implementation and techniques of optimization transformations, it becomes very difficult to construct a scientific comparative study among them. In this case, Reverse Code Engineering (RCE) each compiler would be the only method available to reveal the internal implementation of OpenMP for each compiler However, RCE is a very complicated process, in which reconstructing the logic of the implementation starts with disassembling the binary file and entails dealing with the low level assembly language.

On the other hand, EPCC OpenMP microbenchmarks v2.0 [5] [6] measure the synchronization and scheduling overheads incurred by OpenMP compiler directives of a specific OpenMP implementation (synchronization, loop scheduling and array operations), where the overhead cost incurred by a specific compiler directive is measured by comparing the sequential execution time for a section of code against the parallel execution of the same code containing the compiler directive. An important issue with EPCC microbenchmarks is that they do not take into consideration other important factors such as the effect of the single thread version on the runtime library that is, the difference between the original serial version of the application and single threaded version with OpenMP directives being enabled. Nonetheless, these benchmarks can be used to carry out a comparison among various OpenMP enabled compilers at the directive level. However, the complexity of the real world OpenMP applications is not appropriate for a synthetic benchmark like EPCC. Hence, we cannot decisively determine the performance of the compiler based solely on the overhead and synchronization differences in the OpenMP runtime library The design and structure

of the code might dramatically alter the performance of the OpenMP program since some of the optimization transformations are applicable to a specific code pattern. Moreover, the order of the optimizations implementation in the compiler analysis on the OpenMP code has an important effect on the overall performance of the program [4].

In [7], Pierattini noticed an interesting behavior in one of the reported barrier synchronization overhead. A sudden change in barrier construct speed at 32 processors on Origina2000 machine was due to a change in the implementation such that, the OpenMP runtime library uses a specific processor instruction (ll/sc) to implement the barrier up to 32 processors. Since that instruction does not scale beyond 32 processors, a change in the algorithm is detected (fetch&op). However, this kind of implementation behavior is not detectable by EPCC OpenMP microbenchmarks.

SPEC OMP benchmarks suite [8] evaluates the performance of Shared-Memory MultiProcessor SMP systems. The suite consists of 11 OpenMP large scientific applications that are compute-intensive. Eight applications are written in FORTRAN, and three in C. These benchmarks are good candidates to stress the compiler optimizer to generate an optimum code.

Hence, the need for methodical experimentation with different compilers that support OpenMP arises in order to justify the use of a specific compiler over the other. However, there is neither a systematic way nor a defined model that provide a high level dimensionality capable of assessing the behavior and performance of OpenMP programs and compilers.

In this thesis, we propose a comparative model that provides a methodical and well defined approach to solve these problems from the compiler perspective. Having a proper control over the states that constitute an OpenMP program helps in classifying the reasons behind each compiler performance differences. The program states are defined as follows: original code (OpenMP not activated), OpenMP code with 1 thread, the state when number of threads is equal to the number of cores and the state when number of threads is greater than the number of cores. Our model does not

work at the directive level, because we quantify the performance of the program as a whole based on the states partitioning we provide as a better solution.

OpenMP programs performance is, to a certain extent, determined by the level of optimization and efficiency of the compiler implementation of OpenMP as well as the operating system memory and thread scheduling management implementation. However, there are some architecture-specific decisions (e g., instruction selection, scheduling and vectorization) that each compiler can exploit to produce processor-specific optimized code, targeting one or more architecture as part of the same executable file. For example, PGI and Intel compilers provide Unified Binary [9] and CPU-Dispatch [10] technologies respectively, which allow the generation of a single executable file that has multiple binary code streams, each optimized for a specific architecture. Hence, a mapping of one executable to many compatible platforms is guaranteed to maintain good performance.

Therefore, all the optimization transformations are inherent to each compiler and not all the compilers have the same capabilities. To abstract these differences, we need a model that can support reasoning at a higher level without breaking the underlying variations. To our knowledge there has not been any attempt to devise an OpenMP compiler comparative model.

The model proposed in this thesis works at a higher level of abstraction to reason about the most probable performance problems It is complementary to the low level analysis phase. The state categorization it provides helps in pointing out specific characteristics of the OpenMP runtime system, compiler differences, and code design. Because the proposed model accounts for all the states that an OpenMP parallel program can take, it was successfully used to derive a concrete solution that provides an informative, structured and semantically rich compiler comparative model.

Since the state composition is explicitly profiled, an enhanced version of the standard performance metrics (speedup, overhead and efficiency) is proposed that are more precise in terms of OpenMP implementation.

The model presented here is verified experimentally using three well known commercial compilers on three OpenMP programs The results are interesting because the model exposed the differences between the compilers in a subtle and unobtrusive way In the experimentation section, it is shown that OpenMP runtime library has no effect on the performance differences between compilers as much as it is the compiler optimizer capabilities and dependability

I.1 Contribution

This thesis presents, for the first time, a complete self-defined compiler comparative model for OpenMP parallel programming programs The model consists of 16 equations, divided into 3 categories, where each category verifies specific aspects of OpenMP programs performance in terms of the states definitions and compilers interactions The thesis also presents a compiler pattern detection algorithm to identify comparable behaviors across multiple OpenMP enabled compilers In addition, the following pages enfold an improved version of the standard performance metrics based on the model definitions which are more accurate in terms of OpenMP implementation The model definitions are also used as a base to find the model behind the optimum performance for a given OpenMP application

I.2 Thesis Organization

This thesis is organized in six chapters Chapter II provides an overview of the history, execution, internal and translation model of OpenMP along with a brief review of some OpenMP features and an example showing how OpenMP works While chapter III briefly discusses related work of OpenMP performance in terms of compiler optimization differences and various OpenMP performance monitoring proposals, chapter IV explains in details the proposed compiler comparative model that addresses the difficulties of conducting performance comparison among OpenMP enabled compilers An experimental evaluation of the model is presented in chapter V which proves

the applicability of the model. Finally, we conclude in chapter VI and overview possibilities for future work.

CHAPTER II

"The shortest path between two truths in the real domain passes through the complex domain" - Jacques Salomon Hadamard

OpenMP THE LANGUAGE

OpenMP [1] is a set of compiler directives, library routines and environment variables It supports C/C++ and FORTRAN as part of the specification to create a multithreaded programming language The unit of execution in OpenMP is a thread that shares the same address space with another thread It is based on the fork-join programming model and is designed with ease of use as one of its main goals The high performance computing communities as well as major industry vendors are not only supporting OpenMP evolution and adopting it as an influential programming paradigm for shared-memory parallel programming, but also adopting it for hybrid programming, such as the combination between OpenMP and MPI on distributed shared memory architectures

II. 1 Why OpenMP

The driving force behind OpenMP is that for the past fifteen years, there has been a need for a standardization of Symmetric Multi-Processing (SMP) architecture development Since shared memory architecture have been around for a long time, most of the major corporations of high performance shared memory multiprocessor computers have their own set of directives, consequently obstructing the portability of the code across different platforms To amend this, openmp org was established in 1996 to set a standard that is adequate for the high performance community and industry and to provide better code portability across shared memory platforms The standardization aspect of OpenMP is supported by all major vendors such as IBM, Intel, HP, SGI, Sun, NASA Ames and many others and is highly regarded in the high performance computing community

II. 2 History

OpenMP is considered young and promising. In 1997, the first specification of OpenMP for FORTRAN was released. The Architecture Review Board (ARB) released another version (OpenMP 2.1) for C/C++ and FORTRAN as separate specifications with more features. In version 2.5 FORTRAN and C/C++ were merged together to form a single specification. After that, OpenMP 3.0 was released in May 2008 with a major support for irregular parallelism through the tasking model to exploit unstructured parallelism efficiently [1] [11]. Figure 1 shows the evolution of OpenMP specification.

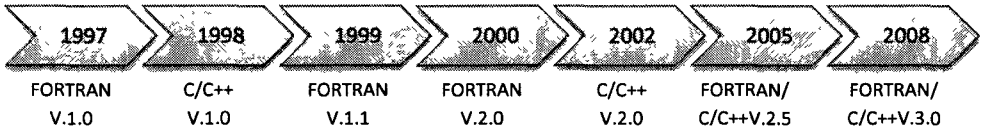


Fig. 1. Time line of OpenMP specification

II. 3 OpenMP Features

OpenMP supports the fork-join programming model. An OpenMP program starts with a single thread called the *master* thread, and when it encounters a parallel region with one of OpenMP constructs, a team of threads is created (forked) to execute the work in parallel. At the end of the parallel region all the spawned threads synchronize (through an implicit or explicit **barrier** construct) and terminate (join; with **nowait** clause, the threads at the end of the construct will immediately proceed to perform other work) and only the initial *master* thread continues. If the team of threads encountered another parallel construct in the same parallel region, then each thread of the original team will form another team of threads, and this is called nested parallelism. Figure 2 illustrates OpenMP's fork-join model.

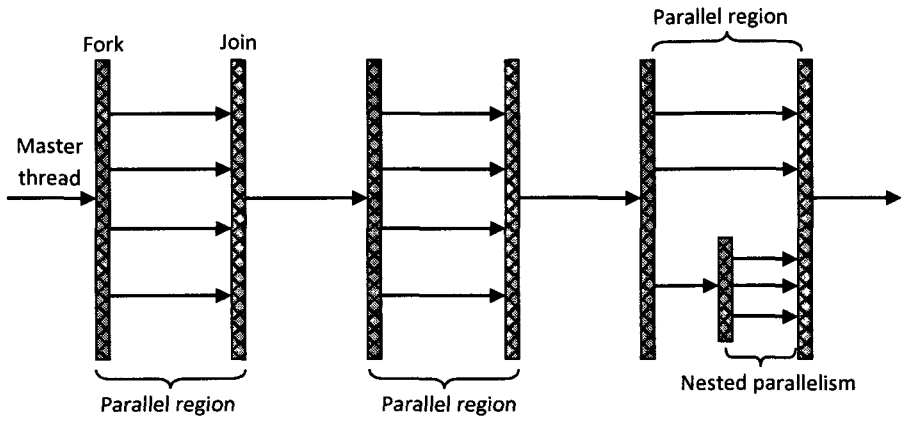


Fig. 2. OpenMP fork-join model

OpenMP directives are placed at main locations in the program source code. In C/C++, the directives are specified using the **#pragma** preprocessing directive, and in FORTRAN, they are specified using special comments that are identified by unique sentinels such as **!$omp**, **c$omp** or ***$omp**. Mostly, OpenMP directives apply to structured blocks with a single point of entry at the top and a single point of exit at the bottom. The compiler will check for OpenMP directives and generate the appropriate code to parallelize the designated code block.

$$\boxed{\textbf{\#pragma omp } directive-name\ [clause[[,]clause]\ldots]new-line}$$

$$\boxed{!\textbf{\$omp } directive-name\ [clause[[,]clause]\ldots]new-line}$$

Most of OpenMP directives enable the program to generate a team of threads to execute a specified region (in OpenMP, a region embraces all the code that is in the dynamic extent of a construct) in parallel. The C/C++ *for* loop or *DO* in FORTRAN is usually the main target for parallelization and especially in the scientific numerical applications. This can be accomplished in OpenMP via the work-sharing construct **omp parallel for** in C/C++ and **omp parallel do** in FORTRAN. In addition, the sections construct **omp sections**, clauses **private** and **shared** with the parallel work-sharing construct **for**, all lead to the creation of a team of threads.

The runtime library routines and environment variables provide a mean to monitor and affect threads, processors and the parallel environment. The runtime library routines are defined in the include header file "omp h" in the case of C/C++.

Environment variables manage the internal control variables ICV that are controlled by the OpenMP implementation, which in turn govern the behavior of the program at runtime in important ways (ICV: *nthreads-var, dyn-var, nest-var, run-sched-var, def-sched-var*).

The **schedule** clause is only supported with the loop construct, which manages the distribution of loop iterations among the threads. Consequently, an appropriate scheduler type that fits specific characteristics of the problem size and properties is a must for performance. OpenMP specification 3 0 supports five kinds of schedule clauses (**static**, **dynamic**, **guided**, **runtime** and **auto**). The schedule syntax is as follows.

schedule (kind, [chunk_size])

The schedule clause states how the iterations of the loop are assigned to the threads in the team. The granularity of this workload distribution is a *chunk* (the chunk_size need not be a constant), a contiguous, nonempty subset of the iteration space.

The **static** (iterations are divided as per the size of the chunk_size statically in a round-robin fashion) schedule has the least overhead, and most of OpenMP compilers enable it by default if no explicit schedule type is specified. For irregular and weakly balanced workloads, **dynamic** (the iterations are assigned to threads as the threads demand them) and **guided** types are helpful for these cases. With guided schedule, the size of the chunk decrease over time. The reason behind **guided** design is that initially, larger chunks are desirable because they reduce the overhead, and this design preserves the fairness among large and small thread work distribution And often load balancing is not of an issue toward the end of the computation. For **runtime**, the decision regarding the schedule

kind is made at runtime via the OMP_SCHEDULE environment variable. In **auto** kind, the scheduling decision is delegated to the compiler implementation or runtime system

Selecting an appropriate chunk size is not always easy as it depends on the code in the loop, the specific problem size and the number of threads used. So, delegating the selection of the scheduler to the compiler runtime system is a good choice to pick up the optimum scheduler based on the aforementioned factors as well as other hardware and software specific properties (if designed so).

Except static schedule type, the allocation is non deterministic and it could vary from run to run based on the load on the system.

The interested reader can refer to [1] for more information about OpenMP features.

II. 4 How it Works: An Example

Figure 3 shows a simple OpenMP program with a for loop. The preprocessing directives #ifdef _OPENMP and #endif are used to check if the include header file "omp.h" (OpenMP support) is enabled by the compiler or not. As we can see, OpenMP program is similar to the original sequential version, where OpenMP directives are inserted on top of the original program. The compiler will generate the multithreaded code by parsing OpenMP directives. Hence, all the low level implementation details are hidden from the user. A call to the runtime library routine omp_get_num_threads() is used which will return a value of 1, because no parallel region has been entered yet. Another call omp_get_wtime() to get the elapsed wall clock time in seconds for the for parallelized loop.

The upper bound of the loop _UB is set to be shared, so that all the threads have access to _UB The index _V is set to private because each thread must be given a unique and local copy of the loop variable _V so that it can safely modify the value. The default clause is set to none so that all the variables referenced in the specified construct will have to be explicity set to be either shared or

private (In C/C++, the syntax is default(none | shared)). So, it is to give variables a default data-sharing attribute. reduction clause (syntax: reduction(operator:list)) is used for specifying some forms of recurrence calculations involving mathematically associative and cummutative operators [1] so that they can be performed in parallel without code modification. Thus, we just have to identify the operation and the variable that will hold the result value, and the compiler will generate all the appropriate code. If we didn't specify variable _A as a reduction then the result of the summation would be zero.

The schedule type is set to static. Using nowait clause eliminates the implied barrier on #pragma omp for loop construct, since there is no need to synchronize threads work at the end of the work-sharing construct in this example. Hence, better performance, and even with this simple example the difference is in microseconds. once you ensure the correctness of the parallelized program results, it is always a good strategy to try to pinpoint the places in the code where implcit barrier is not needed. It is safe to use nowait in this example, since the parallel region ends with a barrier anyhow and the value of _A is not used before the end of the region. Once the threads finish their works, the reduction will compute the summation and leave the results in variable _A.

The printf inside the for loop is to show which iteration gets executed by which thread via the call to omp_get_thread_num(). On a dual core machine, the number of threads would be two if it hasn't been explicity set to something else via omp_set_num_threads(int num_threads). And that gives a perfect speedup!

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3  #include "omp.h"
4  #endif

5  int _tmain(int argc, _TCHAR* argv[])
6  {
7         double _T1, _T2 = 0;
8         const unsigned int    _Ub  = 10;
9               unsigned int  _A, _V = 0;
10
11                #ifdef _OPENMP
12                int _MThrd = omp_get_num_threads();
13                #endif
14                printf("Number of Threads: %d\n", _MThrd);
15
16        #ifdef _OPENMP
17        _T1 = omp_get_wtime();
18        #endif
19        #pragma omp parallel default(none) shared(_Ub) \
20                                  private(_V) reduction(+:_A)
21        {
22        #pragma omp for schedule(static) nowait
23            for (_V = 0; _V < _Ub; ++_V){
24                _A+=_V;
25        printf("Iteration %d is carried out by thread %d\n", \
26                _V, omp_get_thread_num());
27                }
28        }
29        #ifdef _OPENMP
30        _T2 = omp_get_wtime();
31        #endif
32        printf("Time: %f \n", (_T2 - _T1));

33        return 0;
34 }
```

Fig. 3. Simple OpenMP program

II. 5 OpenMP Internals

It is the developer's responsibility to ensure that a proper synchronization between threads has been setup correctly to manage dependencies between them, and that the compiler will generates all the explicit threaded code. After that, in the translation phase of OpenMP directives with the program

source code, the compiler will generate a multithreaded object program Figure 4 shows OpenMP implementation

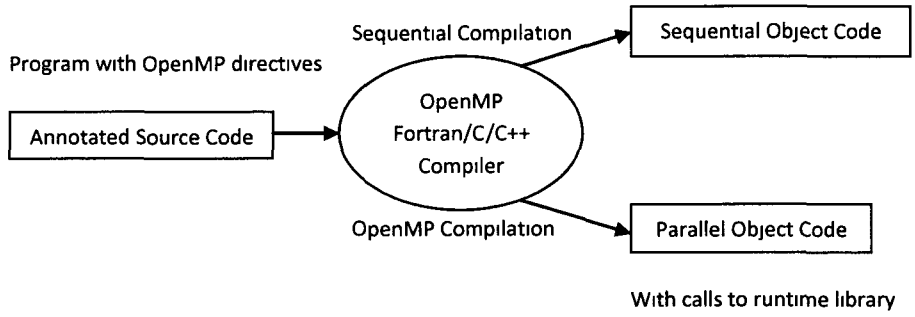


Fig. 4. OpenMP implementation [13]

The compiler ignores OpenMP pragmas if it is compiled sequentially without supplying the required OpenMP option during compilation phase Hence, the same code is used for generating OpenMP parallelized code and serial code just by providing the compiler with the correct OpenMP flag

Threading libraries abound, some of the research compilers merely generate a threaded version of the OpenMP code without relying on the compiler runtime system, hence breaking this integration phase In this case, the runtime is managed through a library with the appropriate calls from the threaded version There are no standard specific runtime libraries or memory management systems to be used with OpenMP as the details are compiler specific This allows for better OpenMP compilers research based of any, such as OMPi [12] for C, a source-to-source translator, where one has the choice of any back end to compile and link the code with (even different thread libraries) Thus providing a flexible environment for experimentation with different compilers

Translating OpenMP directives to a suitable threaded version takes a set of predefined steps by the compiler to ensure the correctness as well as the validation of the OpenMP constructs Parallel

regions must be handled to the thread runtime library routines appropriately. The compiler encapsulates the parallel region into a procedure by an explicit outlining of the parallel code. Preserving shared variables references is done via referencing the shared data accordingly while for private variables each thread will have it is own copy. The outlining phase incurs some overhead which needs to be carefully optimized to avoid hindering the optimizer work as compared to inlining transformation.

[13] and [14] provide a detailed explanation about OpenMP translation phases for many of the OpenMP language features such as parallel constructs, work-sharing constructs with their associated clauses and the runtime system.

CHAPTER III

"In times like these, it is helpful to remember that there have always been times like these" - Paul Harvey

LITERATURE REVIEW

Since our model is the first to tackle the OpenMP compilers comparison problem, there has not been any attempt in the literature for any proposal that has a significant relevance to this work. Hence, there are not enough research papers that resemble our studies. However, this work is self contained since everything has been defined in terms of the model definitions. Nonetheless, we list the most relevant work that touches on some aspects of the studies presented in this thesis.

Compiler optimization is one of the most important factors that can dramatically affect the performance of OpenMP programs. For example, loop optimization, especially loop unrolling improves cache utilization by improving data reuse and loop fusion improves Instruction Level Parallelism (ILP). Thus, the performance of OpenMP programs depends on the code generated by the compiler, the runtime system, compiler options, the library used, calling conventions, the inliner and the capability to generate optimized instructions for specific architecture.

For Memory bounded applications, the Intel compiler provides an option that enables performance tuning and heuristics that control memory bandwidth among processors. Hence, the compiler imposes a selective adjustment on the optimizer to be less aggressive with optimizations that consume more bandwidth, so that the bandwidth can be well-shared among multiple processors for a parallel program.

Tian *et al* [4] present many compiler optimization techniques for the Intel compiler with a special emphasis on the performance of OpenMP programs. They provided a major analysis on the effect of order of optimization phases in the compiler and how critical it is for achieving optimal performance

To test the effect of the OpenMP runtime library on the optimization transformations as compared to the original serial program, 1-thread application with OpenMP ON is compared against the original program Some applications such as the 310 wupwise_m SPECOMP2001 benchmark [8] achieved 88.53% of its serial execution performance, mainly because a less aggressive inlining was performed to reduce resource contention for a better scaling on large CPU count system. On the other hand, 328.fma3d_m benchmark obtained 111.83% of its serial execution performance simply due to an aggression loop invariant code motion being enabled when OpenMP was enabled, and it turns out that this optimization should applies to serial code as well.

In another part of the study, the authors examine the effect of different optimization levels and options on the performance improvement of SPEC OMPM2001 benchmarks, all compiled with Intel C++/Fortran compilers. The performance results show a gain of 3% from OMP +O2 to OMP+O2+IPO (Inter-procedural Optimizations). From OMP +O2 to OMP +O3, the performance gain is 22% and 19% versus OMP +O2 + IPO. This proves the importance of high level optimizations on the multithreaded-code generated for OpenMP programs.

Tian and Girkar [15] studied the effect of optimizations on performance of OpenMP programs. Using SPEC OMPL and OMPM 2001 benchmarks suite compiled at optimization level 3, a performance improvement of 4.3% to 28.3% on some of the benchmarks was achieved. The same study shows a performance gain ranging from 7% to 98% on 10 out of 11 SPEC OMPM2001 benchmarks suite compiled at optimization level O3 plus Inter-Procedural Optimization (IPO) option enabled (compiled with Intel Compiler v8 0). This proves how compiler optimization affects the overall performance of OpenMP programs In [16], Muller studied the effectiveness of various compilers optimization capabilities using simple OpenMP programs to test some of the optimization features.

The translation of OpenMP language features by the compiler has already been discussed in the literature [13] [14] [17] such as parallel constructs and work-sharing constructs, along with their

associated clauses and the runtime system. The performance of OpenMP varies from compiler to compiler, based on the code design pattern and primarily the optimization factor. Some compilers such as Quaver [18] undertake smart analytical decisions to optimize OpenMP barrier elimination whenever possible. Two methods have been used to generate OpenMP code, either on source-to-source transformations (e g., OMPi [12], odinMP [19], NanosCompiler [20]) or these transformations are done internally by the compiler (e g. Intel [21], PGI [22], MS [23]). The latter is better for vectorization and loop optimization.

In [24], Aslot proposed a Quantitative performance analysis Model for parallel programs. The purpose of the model is to quantify the reasons that limit scalability of parallel programs by analyzing the difference between measured and ideal speedup of the parallel program. This difference is subdivided into speedup components which represent the overhead factors responsible for suboptimal performance. However, the model relies on code instrumentation and hardware performance counters as part of the formulations which represent low level profile information.

Defining a performance monitoring interface for OpenMP is not an easy task and requires significant work at the language specification level as well as compiler integration. In [25], Mohr *et al* proposed an instrumentation (at the code or runtime system level) based monitoring interface called POMP. Since OpenMP specification does not support any performance interface as a set of directives, runtime libraries, or API's, POMP proposal aim is to make it to be part of OpenMP API specification. The objective of POMP interface is to develop a clear and portable API for OpenMP program that makes execution events visible to runtime monitoring tools, primary tools for performance measurement and debugging. However, OpenMP directives undergo a complex transformation by the compiler which poses significant challenges at the instrumentation and monitoring levels, which entails an instrumentation that is strongly attached with OpenMP directive processing

Bui *et al.* [26] present a fully integrated OpenMP run-time performance analyzer prototype. This analyzer does not interfere with static compiler optimizer as no instrumentation points are needed. It is totally developed to be part of the underlying runtime environment The analyzer adopts a light-weight sampling based technique to extract low level performance metrics, causing less overhead The instrumentation points are part of the OpenMP runtime library.

Profiling OpenMP programs is very important to understand the behavior of each construct and directive under different workloads To pinpoint the culprit section of the code, a profiler is needed to help identify any load imbalance that is induced as part of OpenMP work-sharing constructs and directives placement. In [27], Furlinger and Gerndt presented ompP OpenMP profiling tool to address this issue in a convenient way, especially to spot any synchronization and communication problems in parallel regions. Performance data representation is reported in a very expressive way, for each construct and directive. The semantics of each region is preserved so that the reported times and counts is self informative as per the region descriptor.

Tools are very important assets when it comes to examining the behavior of OpenMP programs and the system under analysis. Performance measurements entail careful instrumentation at the software and system level to provide descriptive and helpful results which can facilitate in determining the causes of the problem and even provide intelligent advisory solutions for debugging, creating and optimizing applications for multicore processors. This kind of solutions has been already addressed with Intel Parallel Studio [3] which can greatly help in designing, verifying, finding bottlenecks, pointing out memory and threading errors and tuning parallelized applications for better performance on multi-core machines

Formalizing performance problems with a well defined specification language is a necessary objective to simplify testing automatic and manual performance analysis tools That also help express the correctness and effectiveness of the language under investigation such as OpenMP, MPI, and HPF in a structured way However, this still requires the performance analysis tools to support

enough performance information that correlates with the language specification. In [28], Fahringer *et al.* defined the APART Specification Language (ASL) for writing portable specifications of typical performance problems. To test the automatic performance analysis tools, Gerndt *et al.* [29] developed the APART Test Suite (ATS) which allows easy construction of synthetic positive and negative test programs for testing the correctness and effectiveness of those tools. In ASL terminology, a *performance property* characterizes a particular performance-related behavior of a program based on available or required performance data. For example, some of OpenMP load imbalances performance properties as defined in ASL: imbalance_in_parallel_region, imbalance_in_parallel_loop, imbalance_in_parallel_loop_nowait, imbalance_in_parallel_section, imbalance_due_to_uneven_section_distribution.

APART also defines the properties for synchronization, control of parallelism and inefficient serial execution. Each performance property is described by a boolean *condition*, which has an associated *severity* for expressing the relative importance of the property. So, a performance property is a *performance problem* if it is present and it's severity exceeds a preset threshold. On the other hand, a *performance bottleneck* is the most severe performance problem. In testing Hitachi tool, simple load imbalance problems weren't detected due to the unavailability of synchronization information. The author proposed the use of hardware performance counters to get hints to load imbalances in identifying false sharing. On the other hand, testing EXPERT tool proved to be able to detect performance problems automatically with synchronization overhead. However, detecting the reasons behind load imbalances is not automatic, showing the complexity and dependability of these performance analysis tools when the system under analysis does not fully support the required data.

CHAPTER IV

"And there must be simple substances, because there are compounds; for the compound is nothing but a collection or aggregatum of simples"
- Leibniz

PROPOSED MODEL

IV. 1 Introduction

The proposed model provides a topological assessment of the compiler differences in well defined formulations. The model enables an inclusive classification of OpenMP parallel programs. It works by dividing the program conditions into four states which account for all the possible cases that need to be evaluated Each state models and depicts specific attributes about OpenMP program performance. The first state ($Reference_{original}$) tells about the performance of the original non-parallelized program without any OpenMP pragmas enabled by the compiler. The second one ($Sequential_{1-thread}$) is to check for the overhead incurred by OpenMP directives and the runtime system with one thread as compared to $Reference_{original}$ state. The third state ($N_{threads} = N_{cores}$) is when the number of threads is equal to the number of processors/cores, and this state reports the perfect speedup that the program can achieve in an optimal situation. Lastly is the state where the number of threads is greater than the number of cores ($N_{threads} > N_{cores}$), which for some applications scales well to some extent (e.g. server-type applications) but needs to be quantified to ensure enough coverage is attained.

Each OpenMP enabled compiler is mapped to all of the four states for comparison. The structure of the model is shown in Table 1. Each intersection cell represents the timing for one of the states with a particular compiler. The notation in every cell is an abbreviation that links both a particular state with a specific compiler and is read clockwise starting from T letter, for example, $T_{ref}^{C_i}$ is the time for compiler C_i in reference state

IV. 2 Model Definitions

The model is defined using set notations to clearly describe the relationships and memberships between the program states and each compiler We strived to construct the definitions in simple proper formulations. However, Table 1 is a helpful visualization tool that provides an easy access to the set formulations and can be used in conjunction with set notation. Nevertheless, in the formulation phase, the rules need to be carefully observed. We believe that the proposed formulations are enough to account for almost all the major operations in the comparison phase. The model is flexible in its formulations in that it is easily extensible. However, our main concern is to conceptualize the low level details to a higher level of abstraction that is to sustain a semantically rich system.

We define C as the set that contains all the OpenMP enabled compilers. Implicitly, every C_i holds all the options that are to be used to conduct the experimental comparison. $\Phi_y^{\tau_z}$ is a globally restricted set which defines the conformance of options used for every comparison and compiler, consequently, it contains all the options that are supported by every compiler in the comparison. Virtually, it contains the set of locally restricted options that are drawn from the global set $\Phi_y^{\tau_z}$ for one comparison; hence, it needs to be updated for every comparison with different subset of options. And for the comparison to be valid, $\Phi_y^{\tau_z}$ has to hold true all the time and for every compiler The options for every compiler should be highly comparable to preserve a fair and unbiased comparison. τ is the set of all OpenMP programs. The subscript y is a tracking number used to differentiate between different sets of options for the same OpenMP program, such that more than one $\Phi_y^{\tau_z}$ can map to the same τ_z where $(y \geq z)$. Θ set contains all the states that each OpenMP program can spawn. In Ω we define a relation between τ_z and C in which every OpenMP program belongs to the set of states Θ and compilers C

The 1-argument predicate $\lambda(k)$ is introduced to account for the reference state α in Ω Because we defined τ_z as the set that contains all the OpenMP programs, then not all OpenMP programs belong

to Θ as defined in Ω. And this is controlled via omp_flag option that enables/disables the activation of OpenMP code.

$$\tau_z = \{Z | \forall (z \in Z), z \; is \; an \; OpenMP \; program\}$$

$$C = \{Q | \forall (q \in Q), q \; is \; an \; OpenMP \; enabled \; compiler\}$$

$$\Theta = \{S | s = \alpha \vee s = \beta \vee s = \gamma \vee s = \zeta\}$$

$$\Phi_y^{\tau_z} = \{X | \forall (x \in X)(x \in C_i \Leftrightarrow x \in C_{i+1}) \; for \; all \; i = 1, \dots, n \; and \; y = 1, \dots, n\}$$
$$\Omega = \{\eta \in \tau_z \times C | \eta = (z, q), where \; \forall \langle (z \in \Theta)_{\lambda(z)} \Rightarrow (z \in C) \rangle\}$$

$$\lambda(k) = \begin{cases} \exists! \, h_{=omp\_flag} \notin \Phi_y^{\tau_z}, & if \; k \in \Theta|_{k=\alpha} \\ h_{=omp\_flag} \in \Phi_y^{\tau_z}, & else \end{cases}$$

Table 1. The structure of OpenMP compiler comparison model.

Order	State	C_i	C_{i+1}	⋯	C_{i+n}
α	$Reference_{original}$	$T_{ref}^{C_i}$	⋯		$T_{ref}^{C_{i+n}}$
β	$Sequential_{1-thread}$	$T_{1\text{-}th}^{C_i}$	⋯		$T_{1\text{-}th}^{C_{i+n}}$
γ	$N_{threads} = N_{cores}$	$T_{N_{th}=N_c}^{C_i}$	⋯		$T_{N_{th}=N_c}^{C_i+n}$
ζ	$N_{threads} > N_{cores}$	$T_{N_{th}>N_c}^{C_i}$	⋯		$T_{N_{th}>N_c}^{C_i+n}$

α, β, γ and ζ are mutable variables that simplify the manipulation of each state. So, instead of referring to the state name in the formulations, it is easier to use a variable that plays many roles when assigning multiple denotations. The range (number of threads K) of ζ is application oriented. Less often, some applications show good performance when $N_{threads} >_K N_{cores}$ for small K. However, in [30], Suleman *et al.* proposed a mechanism that controls the number of threads based on the application behavior at runtime, it predicts the optimal number of threads based on the amount of data synchronization as well as the minimum number of threads required to saturate the off-chip

bus. Usually, the first three states provide strong evidence about the overall performance of the application. However, we can formulate the range as follows·

$$N_{cores} + 1 \leq \zeta_{N_{threads}} \leq N_{cores} + K \mid K \geq 1 \; and \; K \; is \; application \; oriented. \quad (1)$$

ζ can be quantified by taking the geometric mean for a specific range as implied by K:

$$\zeta = \left(\prod_{N_{threads}=N_{cores}+1}^{n=N_{cores}+K} T_{N_{threads}}^{C_i} \right)^{1/n}$$

The geometric mean is a viable metric to enumerate over all the threads in ζ state, since the purpose at this stage is to compute a single value for differential comparison against γ primarily and the other states.

For $\beta \; and \; \gamma \;\; \rightarrow \gamma \neq \beta \mid (\gamma_{N_{threads}} \wedge \gamma_{N_{cores}}) > 1$

The four states we defined target a specific behavior of the application. Each state serves a specific purpose. However, in order to determine the continuity of the behavioral evolution for a given OpenMP program, we have to consider each thread as a constituent part of the overall performance Hence, we define the Intermediate state I_γ^β which lies between β and γ states. I_γ^β state completes the accountability for every computational thread in the model. On the other hand, we do not consider I_γ^β as a major state since it does not serve a major functional interpretation by itself as compared to the other four states But, I_γ^β accounts for the expectancy of finding a time decreasing homogenous behavior as I_γ^β approaches γ I_γ^β can be quantified by taking the geometric mean as well.

We calculate the number of threads for I_γ^β as follows $\left(I_\gamma^\beta\right)_{N_{threads}} = \gamma_{N_{threads}} - 2$

IV. 3 Model Consideration for HT

Some architectures support Simultaneous Multi-Threading (SMT) such as Intel Xeon processor MP implementation of Hyper-Threading (HT) Technology [31] which enables the operating system to

see a single processor as two logical processors with each processor maintaining a separate run queue. Almost all of the physical resources are shared by the logical processors, such as cache, execution units, branch predictors, control logic and buses. Most importantly, HT provides better supports for Thread Level Parallelism (TLP) as it allows multiple independent threads to execute different instructions each cycle. As mentioned in [32] and [33] HT provides no potential gains unless the application program is multithreaded.

In addition to that, the operating system must support HT via HALT instruction optimization to avoid the idle loop. The compiler optimizer should take advantage of SMT/HT by leveraging the use of vectorization instructions such as Streaming-SIMD-Extensions (SSE and SSE2) with other inter procedural optimizations. Hence, SMT has no special effect on the definitions and states of the model and it is to be considered as two logical processors. The reason is that we still have the control over TLP.

IV. 4 Horizontal X Vertical Formulations

We group the formulations in three categories: Horizontal: the ones that work per state, Vertical: per compiler and Horizontal X Vertical which provides a joint analysis about the overall performance of the compilers. The formulation below checks for the consistency of a specific compiler performance by taking the minimum timing across all the compilers and states.

$$\left.\begin{aligned} \alpha_{C_i} &\dashrightarrow min\left\{T_{ref}^{C_i} \cdots T_{ref}^{C_{i+n}}\right\} \\ \beta_{C_i} &\dashrightarrow min\left\{T_{1\text{-}th}^{C_i} \cdots T_{1\text{-}th}^{C_{i+n}}\right\} \\ \gamma_{C_i} &\dashrightarrow min\left\{T_{N_{th}=N_c}^{C_i} \cdots T_{N_{th}=N_c}^{C_{i+n}}\right\} \\ \zeta_{C_i} &\dashrightarrow min\left\{T_{N_{th}>N_c}^{C_i} \cdots T_{N_{th}>N_c}^{C_{i+n}}\right\} \end{aligned}\right\} if\left(\alpha_{C_i} = \beta_{C_i} = \gamma_{C_i} = \zeta_{C_i}\right) \quad (2)$$

$$\boxed{(2) \Longrightarrow C_i \text{ is uniformly superior else unevenly ranked.}}$$

In (2), the timing differences between the compilers across the states could be very small ($\pm k$), but the generality of the formulation remains the same. However, the differences should be

significant in order to obtain meaningful results. In the case where there can be more than one minimum, the compilers performance is equal.

To determine whether the compilers are close in their performance, we take the total sum σ_π of the standard deviations σ_x for each state across all the compilers, and then check if every σ_π over the σ_x is much less than σ_π. Let

$$\sigma_\pi = \sum_{x=\alpha}^{\zeta} \sigma_x \quad \textbf{such that} \quad if \left[\forall \left(\frac{\sigma_\pi}{\sigma_x} \right)_{x=\beta}^{\zeta} \right] \ll \sigma_\pi \tag{3}$$

$$\boxed{(3) \Longrightarrow \textit{the compilers exhibit huge differences, and they are not comparable.}}$$

$$else\ if \left[\forall \left(\frac{\sigma_\pi}{\sigma_x} \right)_{x=\beta}^{\zeta} \right] \leq \sigma_\pi \tag{4}$$

$$\boxed{(4) \Longrightarrow \textit{the compilers exhibit relative distribution, which means they are comparable.}}$$

α state is not included in the conditional evaluation, because of the effect it has on the distribution as compared to the other states. After all, it has nothing to do with the checking of the actual differences at the level of the library implementation.

IV. 5 Vertical Formulations

The following formulations provide informative messages that pinpoint the most probable causes of the performance degradation.

$$if \left[\left(T_{ref}^{C_\iota} \cong_\varepsilon T_{1\text{-}th}^{C_\iota} \right) \mid \varepsilon \textit{ is very small} \right] \tag{5}$$

$$\boxed{(5) \Longrightarrow \textit{OpenMP runtime library incurs no significant overhead.}}$$

$$if \left[\left(T_{ref}^{C_\iota} >_\varepsilon T_{1\text{-}th}^{C_\iota} \right) \mid \varepsilon \textit{ is small} \right] \tag{6}$$

$$\boxed{(6) \Longrightarrow \textit{could be due to optimizations.}}$$

$$if\left[\left(T_{ref}^{C_i} <_{\varepsilon} T_{1\text{-}th}^{C_i}\right) | \, \varepsilon \; is \; large\right] \tag{7}$$

$$\boxed{(7) \Rightarrow OpenMP \; runtime \; library \; incurs \; significant \; overhead.}$$

(5), (6) and (7) have already been discussed in the literature [14]; however, they are not adaptable enough to account for the sensitivity of the assessment of the differences across α and β states. Practically, the degree of difference is important to consider, since at this level, minimal variations cause each formulation to report a specific diagnostic message and that's why we reformulated them in terms of the model structure with better characterization and sensitivity.

Despite the fact that (6) appears counterintuitive, it seems that it is possible, since the compiler optimizer may find other opportunities for aggressive optimizations when OpenMP is enabled. In [4], the authors found a case where 328.fma3d_m SPEC OMPM2001 benchmark compiled with Intel compiler got 111.83% of its serial execution performance $\left(T_{ref}^{C_i}\right)$. And it was due to an aggression loop invariant code motion being enabled when OpenMP was enabled. (7) could also be due to less aggressive optimization when OpenMP is enabled.

$$if[\gamma_{C_i} < (\beta_{C_i} \wedge \alpha_{C_i})] \tag{8}$$

$$\boxed{(8) \Rightarrow \gamma_{C_i} \; is \; scaling \; appropriately \; for \; the \; same \; compiler \; system.}$$

$$else \; if\left[(\gamma_{C_i} < \alpha_{C_i}) | \, \gamma_{C_i} \cong \frac{\beta_{C_i}}{N_{cores}}\right] \tag{9}$$

$$\boxed{(9) \Rightarrow \gamma_{C_i} \; is \; perfectly \; scaling \; for \; the \; same \; compiler \; system \; (Optimal).}$$

$$\overline{Otherwise \; (9) \; does \; not \; scale \; at \; all.}$$

$$if\left[\left((\gamma_{C_i} <_K \zeta_{C_i}) \wedge (\zeta_{C_i} <_{K'} \beta_{C_i})\right) | (K' > K)\right] \tag{10}$$

$$\boxed{(10) \Rightarrow \zeta_{C_i} \; is \; progressively \; homogeneous.}$$

$$else\ if\left[\left(\left(\gamma_{C_i} <_{K\gg} \zeta_{C_i}\right) \wedge \left(\zeta_{C_i} <_{K'} \beta_{C_i}\right)\right) | (K' \gg K) \wedge \left[\left(\alpha_{C_i} \vee \beta_{C_i}\right) \neq \gamma_{C_i}\right]\right] \quad (11)$$

$$\boxed{(11) \Longrightarrow \textit{scalability is very bad and } \zeta_{C_i} \textit{ exhibits too much fluctuations.}}$$

(10) and (11) check for the relative consistency in the variations between the threads with respect to ζ state. The difference in the timing among the states should be harmonical and stable such that the difference should lie within the context of each state. However, sometimes the variations within ζ state exhibit irregular behavior in which comparing them against γ state confirms the existence of inconsistency in the performance of the compiler by showing huge differences that are not supposed to be there. In (11), if this condition $\left[\left(\alpha_{C_i} \vee \beta_{C_i}\right) \neq \gamma_{C_i}\right]$ holds false, this means that no parallelization occurred, hence, we should flag this instance of comparison as faulty due to an intrinsic problem by the compiler OpenMP runtime library which could be an optimization or code generation problem.

$$else\ if\left[\left(\left(\gamma_{C_i} <_{K\ll} \zeta_{C_i}\right) \wedge \left(\zeta_{C_i} <_{K'} \beta_{C_i}\right)\right) | (K' \gg K) \wedge \left(\beta_{C_i} > \alpha_{C_i}\right)\right] \quad (12)$$

$$\boxed{(12) \Longrightarrow \textit{the problem lies in } \beta_{C_i}.}$$

$$if\left[\gamma_{C_i} >_{K\gg} \left(\beta_{C_i} \wedge \alpha_{C_i}\right)\right] \vee \left[\left(\beta_{C_i} \wedge \gamma_{C_i}\right) >_{K\gg} \alpha_{C_i}\right] \quad (13)$$

$$\boxed{(13) \Longrightarrow \textit{Code design problem, most probably scheduling problem.}}$$

IV. 6 Horizontal Formulations

For the compilers performance to be relatively close they must register a small standard deviation σ that lies within an acceptable K, such that K is application oriented. Evaluating the reference state σ differences provide a strong indication about the actual deviations in the model as compared to the other states. Hence, the below formulations infer the hidden causes of the variations.

$$if\left[\sigma\left(T_{ref}^{C_i},\cdots,T_{ref}^{C_{i+n}}\right)\right] \gg K \tag{14}$$

$$\boxed{(14) \Rightarrow \textit{This is weak comparison. And it is highly probable that the same variations will be reflected on the other states.}}$$

$$if[\sigma(\beta,\gamma \text{ and } \zeta)] \gg K \tag{15}$$

$$\boxed{(15) \Rightarrow \textit{This is in perfect correlation with the above formulation Hence, the problem is not in the OpenMP runtime library, but compiler code generation and optimization factors.}}$$

IV. 7 *K*-stage Comparator

K-stage comparator provides a coarse-grained and scaled version of the actual timing numbers for every state and compiler. It unties the numbers from their unique states by dividing the time for each compiler state over the sum of the reference time for all the compilers as shown in the below formulation and Figure 5 Hence, an intermediate representation is revealed that shows the differences at a finer level such as, the minimal decrease or increase in the timing from state to state is smoothed to hide the insignificant minor differences between the states while the major ones are exposed appropriately Hence, differences that range between $\pm k$ for very small k between the states are not momentous and that meet the model formulations accurately.

$$K\text{-}SC_{\rho\in\Omega} = \frac{\rho}{\sum_{l\in\Theta}\forall(r\in C)_{l=\alpha}} | K \in \Theta. \tag{16}$$

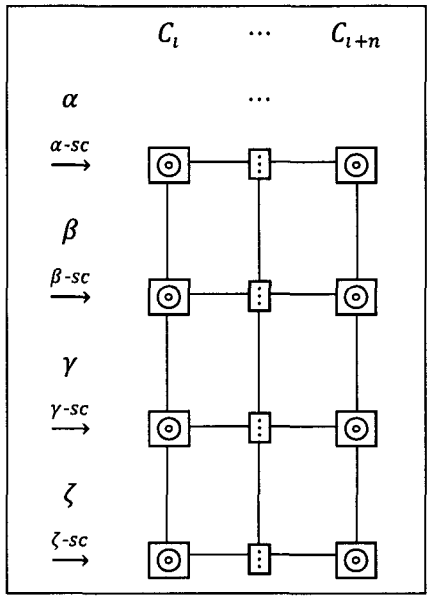


Fig. 5. *K*-stage comparator.

IV. 8 Detecting Compilers Patterns

Some compilers exhibit similar behavior on the same OpenMP program. That is, an increase or decrease in time from state to state is also reflected on more than one compiler. However, this doesn't mean that the performance is the same, but finding behavioral similarities is helpful in determining the real causes of the differences across multiple OpenMP programs. And this is done by binary encoding each state with respect to the previous and next state. The algorithm is presented in Algorithm 1 In the reference state, the compiler that registers the highest timing is set to 1 (L. 08), everything else to 0 (L. 09). After that, we set the next state to 1 or 0 if the previous state record is less (L. 12) or more (L. 14) respectively; otherwise if no change happened then set the value to the value of the previous state (L. 15). And this is best illustrated using the radar-diagram (later in the experimentation evaluation section, Figure 14) which conceptualize the structure of each compiler with respect to the four states and provides a coarse grain description about the model.

In algorithm 1, the computational complexity for finding the maximum in α state is $O(|C| - 1)$; $O(|\Theta||C|)$ for setting the flag over all the states across all the compilers, therefore the computational complexity for Compilers Pattern Detection algorithm is $O(|\Theta||C| + |C| - 1)$.

Algorithm 1. Compilers Pattern Detection

01. **Input**: C and Θ
02. **Output**: Radar diagram
03. **_begin**
04. **for** $\psi_{\omega}^{[\mu]}|\psi^{[\mu]} \in \Theta, \mu \in [1,4]$ /* *ψ loops over all the compilers in ω as stated at L.5* */
05. **for** $\omega := C_i|i \leftarrow 1$ *to* n
06. **switch** $\left(\psi_{\omega}^{[\mu]}\right)$ /* *μ is an index to loop over all the states in Θ in order* */
07. *case* $\psi_{\omega}^{[\mu]=\alpha}$
08. *set max* $\left(\psi_{\omega}^{[\mu]}\right) \leftarrow 1$;
09. **else** *set* $\psi_{\omega}^{[\mu]} \leftarrow 0$;
10. *case* $\psi_{\omega}^{[\mu>1]}$ /* *for all other cases/states β, γ and ζ* */
11. **if** $\left(\psi_{\omega}^{[\mu]} > \psi_{\omega}^{[\mu-1]}\right)$ **then**
12. *set* $\psi_{\omega}^{[\mu]} \leftarrow 1$;
13. **else if** $\left(\psi_{\omega}^{[\mu]} < \psi_{\omega}^{[\mu-1]}\right)$ **then**
14. *set* $\psi_{\omega}^{[\mu]} \leftarrow 0$;
15. **else** *set* $\psi_{\omega}^{[\mu]} \leftarrow \psi_{\omega}^{[\mu-1]}$;
16. **end if**
17. **end for**
18. **end for**
19. **_end**

IV. 9 Graph Theoretical Representation and Problem Modeling

The definiteness of the model allows it to be very flexible and easy to map to different theoretical representations. The model can be represented as a simple undirected graph as shown in Figure 6a, where each graph has its own compiler C_t that operates on the same OpenMP program P_t and states Θ.

All the compilers graphs share the same graphical representation and structure. Hence the graphs are isomorphic. And we define the model for a single graph instance as follows:

Let $G_{c_t} = \{V, E, \Lambda\}$, where $c_t \in C;\ \Theta \subset V, V = \{\alpha, \beta, \gamma, \zeta, p_t, c_t\},\ E = \{e_1^{c_t}, e_2^{c_t}, e_3^{c_t}, e_4^{c_t}, e_5^{c_t}\}$,

$|V| = 6, |E| = 5;\ G_{c_t}(6,5)$

Vertex p_t is of degree 5, whereas all the others are pendant vertices.

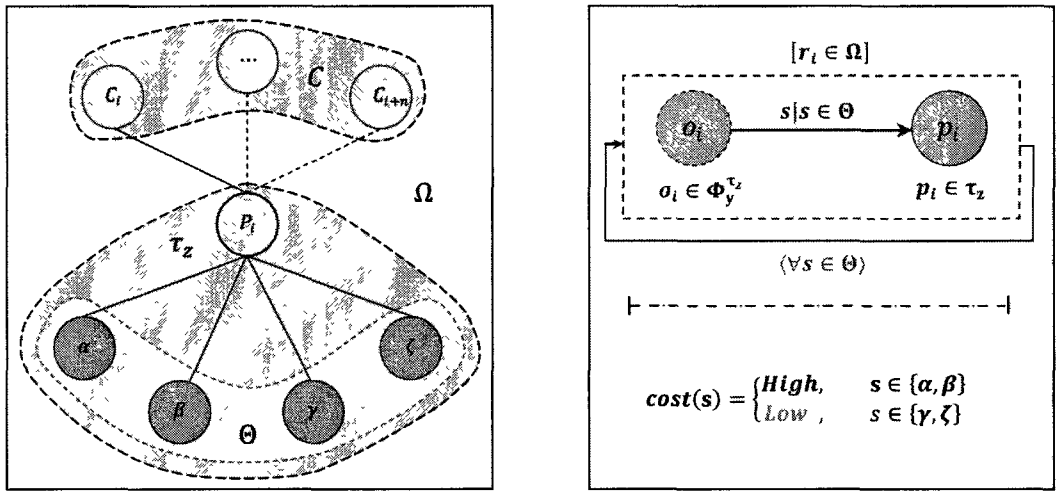
And the incident function is defined as:

$\Lambda : E \longrightarrow \{\{u, v\} | u, v \in V\}, \Lambda(e_1^{c_t}) = \{c_t, p_t\}, \Lambda(e_2^{c_t}) = \{p_t, \alpha\}, \Lambda(e_3^{c_t}) = \{p_t, \beta\}, \Lambda(e_4^{c_t}) = \{p_t, \gamma\}, \Lambda(e_5^{c_t}) = \{p_t, \zeta\}$.

All graphs instances are isomorphic such that $G_{c_t} \cong G_{c_{t+1}} \cong \cdots \cong G_{c_{t+n}}, if\ \exists\ a\ one - to - one\ correspondence\ \aleph : V \longrightarrow V_1 \longrightarrow \cdots \longrightarrow V_n \mid \{u, v\} \in E \Leftrightarrow \{\aleph(u), \aleph(v)\} \in E_2 \Leftrightarrow \cdots \Leftrightarrow \{\aleph(u), \aleph(v)\} \in E_n$.

The set of optimization options and sub-options that are available for each compiler are diversified and large. In cases like the GCC compiler, the number of optimization options can reach up to sixty. The problem is when we try to enumerate over all these options to achieve the best performance possible on a given architecture, application and compilation environment. The complexity increases exponentially as the number of options increase, for instance, for $k = 60$ optimization options, the

search space results in 2^k possibilities. This is a very important problem to tackle since it's not guaranteed that higher optimization levels would achieve better performance [34]. Each program can exhibit different performance on some of the options which are designed for specific architecture.



(a) Graph theoretical representation **(b)** Optimization options comparison process

Fig. 6. Graph mapping of the model.

And the way these options interact with each other is very complicated as there is no way to determine the feasibility of the combination of multiple options without trying them. Brute forcing all the possibilities through an iterative process is definitely an impractical solution. However, this kind of problem has already been discussed in the literature with viable solutions.

For example, in [35] [34] and [36] the authors tackled this problem efficiently to reduce the search space based on selective criteria's such as, random generation of compiler settings, an automatic procedure to select compiler options for a given application based on statistical analysis of profile information using Orthogonal Arrays and performance counters respectively. However, this subject

is beyond the scope of the thesis topic and we only briefly presented it to show how this problem can be mapped to OpenMP enabled compilers from the model perspective as shown in Figure 6b.

Thus, an OpenMP program p_i is compiled with a specific optimization option or multiple options o_i. And this process is repeated ($\forall s \in \Theta$) for each compiler r_i Linking and compilation are needed in α and β states only. We consider the cost of these two states to be high since γ and ζ states can be set via an environment variable without the need for recompilation. And this is a very expensive process for large programs. This mapping provides an approach to find the best possible set of optimization options that maximize the performance.

IV. 10 Compilers Comparisons Consistency

In order to avoid the unfairness that could occur in the experimentation and any other uncertainties, some universal characteristics have to be abided strictly. Therefore, we define another predicate that needs to be preserved consecutively between comparisons. The predicate states that, it is not allowed comparing the same set of compilers under different hardware and setup configurations expecting that the results would be almost the same or scaling proportionally. This would be different comparison and it has nothing to do with the model, since, it infers the consistency of the performance of each compiler on different hard/soft setups. It is certainly different, and we cannot just draw conclusions from one experiment.

Conducting thorough experimentations on both platforms (different OS's) is more of analyzing the behavior of the same compiler under different OS's which means a lot of factors could affect the performance: the libraries, optimizations (some optimizations takes advantage of specific OS features, designed only for specific version of the operating system), scheduling and memory management.. Considering that the implementation of OpenMP library is the same.

Hence, the model works per comparison and to compare it against another different comparison (different Hard/Soft) requires the model to be extended to accommodate for those differences.

Some programs, they are not memory greedy (less fluctuations), only computation intensive, hence we might get the same results on a similar architecture if the saturation level is scalable within the lowest architectural platform up to some threshold.

Therefore, we define the following 2-arg predicate $\mathfrak{U}(s,h)$ that needs to be checked at the beginning of each comparison. If the return is true then continue otherwise break.

$\wp = \{ARCH | \forall(arch \in ARCH), arch\ is\ a\ hardware\ architecture\ feature\}$

$\Re = \{SF | \forall(sf \in SF), sf\ is\ a\ software\ feature\}, where\ C \subset \Re.$

$\mathfrak{H} = \wp \cup \Re$

$$\mathfrak{U}(s,h) = \begin{cases} True, & if\ (\forall[s \wedge h] \in \mathfrak{H}) \\ False, & else \end{cases}$$

IV. 11 The Model Formulations Characteristics

The sensitivity of the formulations is very high. Some of them are intermixed with each other to provide a better problem resolution in such a way that the degree of interpretation of the joint formulations is more synthesized.

For example, formulation (7) is the conditional part in formulation (12) and relatively both lead to the same conclusion However, (7) consists only of two states while (12) operates on the four states including the condition part that is (7). Since (12) is more complex in terms of the states usage, it should be given a higher priority such that the order is enforced when executing (7) and (12) (if both are flagged for an OpenMP program). Thus, (12) is more decisive in terms of problem determination.

Conversely, (7) and (13) hold true in the case of SimpleAdd_x kernel (as shown in the experimentation section) But, the problem is not only the OpenMP runtime library overhead (formulation (7)) So, this formula and as reported in [14] failed to address this issue appropriately

because it has only two specific states which are not trained to detect any other problem possibilities. However, (13) provides a deeper look at what the actual problem might be In cases where scheduling is involved, care must be taken to ensure that it is not the runtime library's fault as stated in (7). It could be that the scheduler kind may not be appropriate for this type of program, or it could be that the chunk size is not properly set; hence, we cannot conclude that it is the runtime library problem unless everything else has been equally verified.

Note that formulations (8) and (9) are almost identical but the stringency in terms of the evaluation performed favors (9), since it converges to a higher limit in the comparison process, which is more accurate.

Formulations (14) and (15) form a semi-decision based order such that if the evaluation of (14) is true it is more likely that (15) will proves (14) conjecture. Hence, this mutual dependence between (14) and (15) allows (15) to concludes with a conclusive decision.

Based on the vertical formulations, we notice that four pairs of formulations work together to verify the culprit state such that each pair is mapped to one of the states with complete coverage of the four states. $[(5),(6)] \rightarrow \alpha$, $[(7),(12)] \rightarrow \beta$, $[(8),(9)] \rightarrow \gamma$ and $[(10),(11)] \rightarrow \zeta$. And this ensures the validity of the model in terms of problem determination across all the states.

IV. 12 2CA Optimum Performance Characterization

The model states classification provides complete performance coverage for a given OpenMP application Hence, based on the states definitions and formulations, we can characterize the behavior of an OpenMP program accurately such that, the optimum performance pattern should match the domains characterization as shown in Table 2 And this is one of the most important conclusions of the model.

Table 2. The 2CA model optimum performance characterization domains.

[1]	$\alpha < \beta$	$\alpha = \beta$	$\alpha > \beta$
[2]	$I_\gamma^\beta < (\alpha \wedge \beta)$		
[3]	$\gamma < I_\gamma^\beta$		
[4]	$\zeta < \gamma; for\ small\ k$	$\zeta = \gamma; for\ small\ k$	$\zeta > \gamma; for\ small/large\ k/k'$

For any OpenMP program, the normal behavior should resembles one of the possible shapes in Figure 7. If the performance does not match any of the possible shapes in Figure 7, then a performance problem is detected. What is worth noting here is the sensitivity of case 4 and especially when k' is large. In this case, when the number of threads is very large as compared to the number of cores available, we should notice a continuous steep rise in that section of the curve. Furthermore, in case 4, for small k, the rising of the curve should be balanced such that, the increase in time as the number of threads increases should be uniform and consistent. Nevertheless, this is not a rule and it should be considered as sensitivity metric which is application dependent.

Note that k and k' represent the number of threads and they are application oriented. Also to note that each state is represented in its unpacked form that is thread by thread. The states α, β, γ, ζ and I_γ^β in Figure 7 represent the execution time for an OpenMP program.

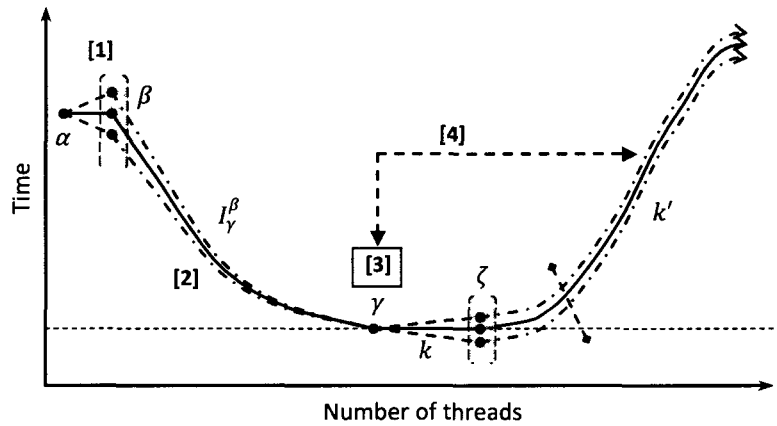


Fig. 7. The 2CA model optimum performance characterization.

It turns out that the shape in Figure 7 resembles the quadratic function shown in (17) Thus, the optimum performance of OpenMP programs must converge to the mathematical formulation (17) for the relationship between time and thread numbers However, the quadratic function does not fully model the actual behavior of the optimum performance since there are some variations along the curve which are not accounted for

$$G_{2CA}(X) = (X-\gamma)^2 \mid X \in (\Theta \cup I_{\gamma}^{\beta}) \tag{17}$$

The difference between α and β states is only one thread. Therefore, formulations (5), (6) and (7) model the exact behavior of this section of the curve as shown in Figure 7 γ state represents the division factor between $(\alpha, \beta, I_{\gamma}^{\beta})$ and ζ states. Since γ state depicts the optimum performance when the number of threads is equal to the number of cores, formulations (8) and (9) must hold true for γ to be optimum. Therefore, by the definition of formulations (8) and (9), γ should be less than α, β, γ and I_{γ}^{β}, otherwise it is not optimum. When number of threads is greater than the number of cores, formulation (10) precisely model this section of the curve. However, the performance of OpenMP programs is unpredictable at this level.

The optimality of the performance characterization shown in Figure 7 can be sorted in three categories such that, the dashed curve below the solid one is the most optimum, the solid curve is the optimum and the dashed curve above the solid one represents the less optimum. Algorithm 2 characterizes the performance of OpenMP programs as well as the optimality level.

The computational complexity for finding the optimality level for a single compiler is $O(|\Theta| + 1)$ and $O\big((|\Theta| + 1)|C|\big)$ for n compilers. However, the computational complexity for the unpacked form that is thread by thread is $O(N_{threads}|C|)$

Algorithm 2. OpenMP Optimum Performance Characterization

01. **Input:** $\Theta \cup I_{\gamma}^{\beta}$
02. **Output:** Performance Optimality Level POL
03. **_begin**
04. **if** $\left[(\alpha > \beta) \wedge \left(I_{\gamma}^{\beta} < (\alpha \wedge \beta)\right) \wedge (\gamma < I_{\gamma}^{\beta}) \wedge (\zeta <_{\{for\ small\ k\}} \gamma) \wedge \left(\zeta >_{\{for\ large\ k'\}} \gamma\right)\right]$ **then**
05. set POL ⟵ "*Most Optimum*"; /* *the dashed curve below the solid one* */
06. **else if** $\left[(\alpha = \beta) \wedge \left(I_{\gamma}^{\beta} < (\alpha \wedge \beta)\right) \wedge (\gamma < I_{\gamma}^{\beta}) \wedge (\zeta =_{\{for\ small\ k\}} \gamma) \wedge \left(\zeta >_{\{for\ large\ k'\}} \gamma\right)\right]$ **then**
07. set POL ⟵ "*Optimum*"; /* *the solid curve* */
08. **else if** $\left[(\alpha < \beta) \wedge \left(I_{\gamma}^{\beta} < (\alpha \wedge \beta)\right) \wedge (\gamma < I_{\gamma}^{\beta}) \wedge (\zeta >_{\{for\ small\ k\}} \gamma) \wedge \left(\zeta >_{\{for\ large\ k'\}} \gamma\right)\right]$ **then**
09. set POL ⟵ "*Less Optimum*"; /* *the dashed curve above the solid one* */
10. **else**
11. set POL ⟵ "*Not Optimum*";
12. **end if**
13. return POL
14. **_end**

IV. 13 An Inclusive Projection of 2CA Over S.O.E Standard Performance Metrics

The S O E standard metrics (18), (19) and (20) do not take into consideration the effect of the OpenMP runtime library as a constituent part of the serial execution time (T_1) as used in [14] [37] Hence, when we take the measured time for p cores and k threads$(T_{p,k})$, the overhead incurred by the runtime library becomes a part of the parallel execution time that has not been computed as part of T_1. And in this case, the results are influenced by the incompleteness of T_1 factor, in which the absence of β state causes the overhead metric to report more optimistic results and the others less pessimistic results (only α is considered).

$$Speedup = \frac{T_1}{T_{p\,k}} \tag{18}$$

$$Overhead = T_{p,k} - \frac{T_1}{P} \tag{19}$$

$$Efficiency = \frac{T_1}{P \times T_p} \tag{20}$$

Since β state is part of the serial execution it should be added to α to account for the overhead incurred by the runtime library. Therefore, γ state becomes free from the concealed effect of β state. The transformed metrics based on the model definitions are represented in $\_I(m,x)_{2CA}$ function, where m defines the metric type and x enables you to choose between either one of the sub-metrics for each metric based on the state you want to analyze. In S_1, $O_1$and E_1 metrics, β is introduced accordingly as part of the serial execution time whereas γ is simply multiplied by 2 to account for the newly introduced β state in the nominator. So, these metrics are only to examine γ state. S_2, O_2 and E_2 sub-metrics deal with ζ state for a specific thread number. Since γ and ζ states are both part of the parallel execution, we add them together such that γ is the base state for any ζ thread specific number x. The sub-metrics provide a mean to study the effect of an additional thread on the overall behavior of OpenMP programs.

When $[(\gamma \wedge \beta) > \alpha \mid \gamma < \beta\ ]$, the improved speedup and efficiency metrics report very optimistic ratios as compared to the standard metrics (Figure 9a). Since the original metrics do not capture the actual parallelism within the runtime library, that is, the difference between β and γ states, hence, they fail to acknowledge this phenomena. In Figure 9b, the differences in the overhead are due to the very high overhead incurred by the runtime library in β state (as shown in the table to the right of Figure 9b).

$$\_I(m,x)_{2CA} = \begin{cases} \overset{S_1}{\underset{S_2}{\begin{cases} \frac{\alpha+\beta}{2\times\gamma}, if\ x = \gamma_{th} \\ \frac{\alpha+\beta}{\gamma+\zeta_{th=x}}, else \end{cases}}}, if\ m = Speedup \\ \overset{O_1}{\underset{O_2}{\begin{cases} 2\times\gamma - \frac{\alpha+\beta}{N_{cores}}, if\ x = \gamma_{th} \\ (\gamma+\zeta_{th=x}) - \frac{\alpha+\beta}{N_{cores}}, else \end{cases}}}, if\ m = Overhead \\ \overset{E_1}{\underset{E_2}{\begin{cases} \frac{\alpha+\beta}{2\times N_{cores}\times\gamma}, if\ x = \gamma_{th} \\ \frac{\alpha+\beta}{N_{cores}\times(\gamma+\zeta_{th=x})}, else \end{cases}}}, if\ m = Efficiency \end{cases}$$

Figure 8 shows the differences across all the metrics when all the states exhibit regular and expected performance. Again, we notice how β is affecting each metric and especially the overhead. Since the difference between α and β states is significant, the overhead exposes it in association with γ state.

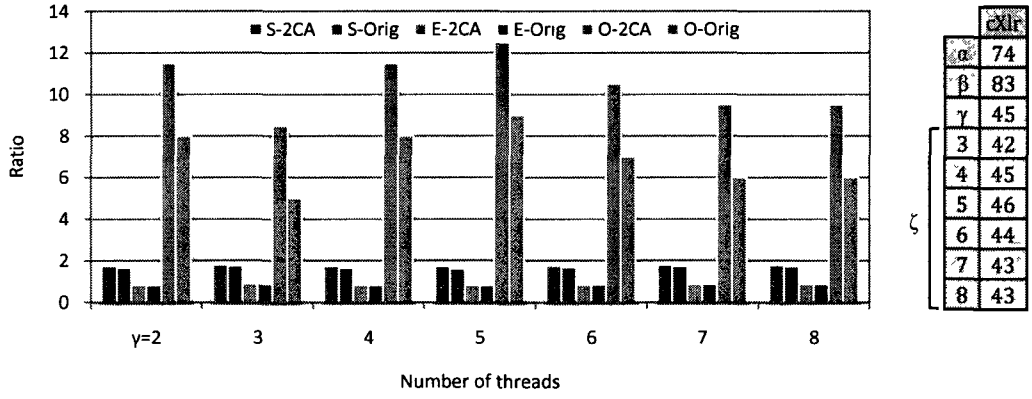
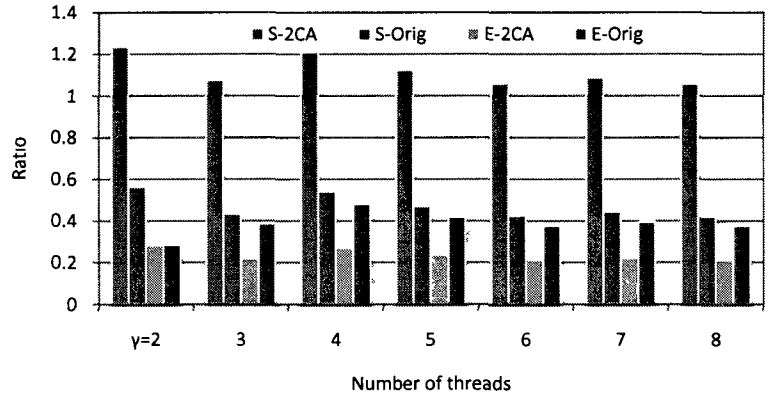
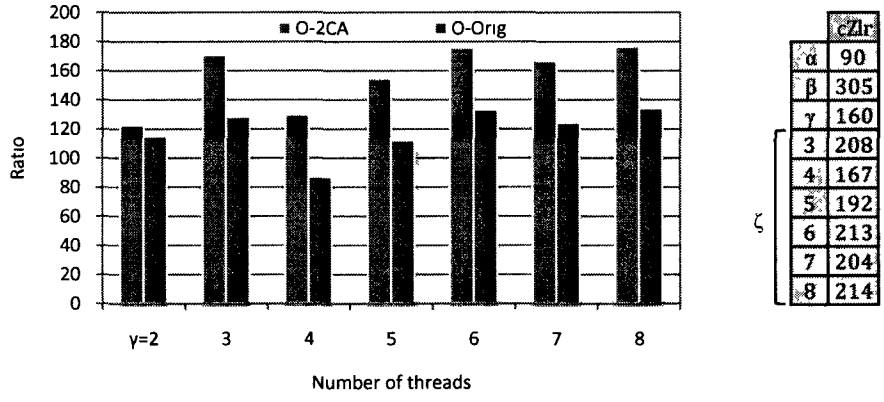


Fig. 8. N-Queens cXlr compiler performance metrics (Speedup, Efficiency and Overhead) evaluation: 2CA vs. Original metrics.



(a) Speedup and Efficiency



(b) Overhead

Fig. 9. SimpleAdd_s cZlr compiler performance metrics evaluation: 2CA vs. Original metrics.

Thus, the improved metrics defined in $\_I(m,x)_{2CA}$ function are more accurate in terms of OpenMP implementation. They provide a fine grained approach that helps unravel the implicit differences within each state. However, the 2CA metrics are not to be compared to the original ones since each one operates on a different set of states. In addition to that, 2CA metrics are to be considered as a special case of the original ones since they target only the OpenMP implementation.

For overhead analysis, equation (19) is a very simplistic model to reason about the actual hidden overhead in the case of OpenMP implementation. Even though it provides a rough estimate about the overhead of the whole code, a much more rigorous schema is needed that correlates with OpenMP constructs for a particular region of interest. In [38], Bane and Riley extend the overhead analysis to comprise multifaceted OpenMP structures that are more definite and precise in terms of OpenMP implementation. In [39], Yongjian *et al.* break down the overhead into detailed categories "such that each overhead class is just corresponding to one identified cause, then by measuring the overhead, we can directly trace back to their causes and thus reduce or even eliminate performance overhead in a recipe way."

In addition to that, a layered model for overhead analysis is proposed in [39] to help programmers locate and understand the performance at OpenMP language level. The abstract layer model of OpenMP implementation views the overhead as inefficiencies in the implementation which can be injected into the code at any of these abstract layers, and the performance of programs written in high level is determined by the implementation efficiency of each lower level abstract layer. The authors claim that the overhead at a specific abstract level can be inherent inefficiency, such as those caused by non-optimal implementations, or inefficiencies induced by high level reasons.

CHAPTER V

"Demonstration is also something necessary, because a demonstration cannot go otherwise than it does, And the cause of this lies with the primary premises/principles" - Aristotle

EXPERIMENTAL EVALUATION

The compiler comparative model presented in chapter four encompasses enormous details that need to be examined experimentally For this purpose, three major commercial compilers were chosen These compilers support OpenMP with their latest versions, PGI Workstation v9 01, Microsoft C/C++ Optimizing Compiler Version 15 00 21022 08 and Intel C++ Professional Version 11 1 Build 20090930 In this chapter, their names are kept anonymous in the analysis due to the sensitivity of the information derived

V. 1 Test Programs

Three C++ examples annotated with the appropriate OpenMP pragmas were used in the experimentation The first example deals with N-Queens problem (taken from Intel compiler samples) of board size 15 which achieves 2279184 distinct solutions and it uses the backtracking search algorithm This code is exploitable by the compiler for any optimization opportunities

The other one is a simple *for* loop kernel (Figure 3) that adds two variables with an upper bound of 10e7 and a stride value of 1 The purpose of the second example is to stress one computing unit such that the instruction mix is the same among all the threads, making all the threads share a non-idle resource which affect the throughput We refer to the first one as the N-Queens example and the second as SimpleAdd_x kernel (where x = s, g or d) s, g and d stand for static, guided and dynamic scheduler kinds respectively that are supported by OpenMP specification Hence, three versions were derived for SimpleAdd_x example, each one tests the performance of a specific scheduler type to track the consistency of the compilers performance behaviors

The third is a molecular dynamic simulations program [40] which consists of 2 heavy computational routines: the first is to compute forces and energies and the second is to compute the displacement and distance between two particles. The runs were taken with 1000 particles, 400 steps and 0 0001 the size for each time step. The purpose behind this program is the heavy computation involved and the optimization opportunities available for the compiler to exploit. We refer to the third test program as MD example.

The three test programs revealed significant differences among the compilers. Providing a performance analysis for each compiler is beyond the scope of this thesis, and we only present the analysis from the model perspective.

V. 2 Experimentation Setup

We assigned the following pseudo names for each compiler to mask its real identity in the analysis (in no particular order): cXlr, cYlr and cZlr. For compilation, we used only the highest optimization option (Intel and MS: Ox, PGI O4) supported by each compiler with the appropriate OpenMP flag. Thus, they are all equivalent as per the options chosen. Windows 7 Pro 32 bit was used on an Intel core 2 duo P7350 @ 2 00GHz with 4 GB's of rams, 32KB L1 D-Cache, 32KB L1 I-Cache (both 8-ways set associative, 64-byte line size) and 3MB L2 Cache (12-way set associative, 64-byte line size). The three programs were compiled as 32-bit binaries Special care has been taken to ensure proper environment setup between each compiler tests to guarantee a fair comparison. Ten runs were taken for each experiment and thread timing, and then averaged using the geometric mean

V. 3 Results Analysis

This section presents the results for the three test programs described in section V.1. Analysis is only provided from the model perspective

Figure 10 shows the compilers performance differences for 8 threads. The relative coherency in the distribution is depicted in Figure 10 as specified in formulation (4). Although cYlr reference timing is not the lowest, it is almost equal to cXlr, therefore by (2), cYlr is uniformly superior. cXlr runtime library registers significant overhead (9 seconds) according to (7), whereas cYlr and cZlr follow (5). The three compilers hold true for (8), (9) and (10). Note that, the scheduler kind in this example is implicitly determined by each compiler at runtime. But we determined that the three compilers selected static scheduler kind via the call to omp_schedule_type OpenMP API.

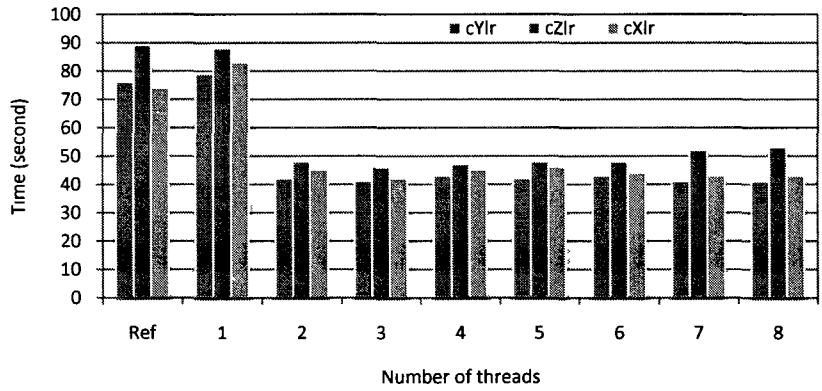


Fig. 10. N-Queens compilers comparison.

cYlr and cXlr support cross linking and compiling of the OpenMP runtime libraries between each other. We compiled and linked N-Queens example with cXlr using cYlr OpenMP runtime library, but the results remained almost the same, with an increase in 1 second of cXlr β state. However, the reference time for cXlr is less than cYlr by 2 seconds. This reveals an important aspect of the compiler optimizer and other factors dependency. Thus, the compiler optimizer and code generator play a significant role in determining the overall performance of the application, and not only the implementation of the runtime library. This helps in revealing interesting aspects of each compiler capabilities.

Formulation (3) holds true for all of the SimpleAdd_s g and d (Figures 11, 12 and 13) examples. The reader should note the large difference between α and β states in the case of SimpleAdd_s and g for cZlr and cXlr, as stated in (7). (11) holds true for SimpleAdd_g and d for cZlr compiler. Formulation (12) strongly holds true for cXlr in SimpleAdd_s and cZlr in SimpleAdd_d. In SimpleAdd_d, we notice an exceptionally abnormal behavior for the three compilers when OpenMP is activated in β and γ states for cYlr and cZlr, while in cXlr this abnormality starts earlier in α state, this is certainly a scheduling problem and most appropriately the impractical chunk size that has been set per iteration, hence (13) holds true.

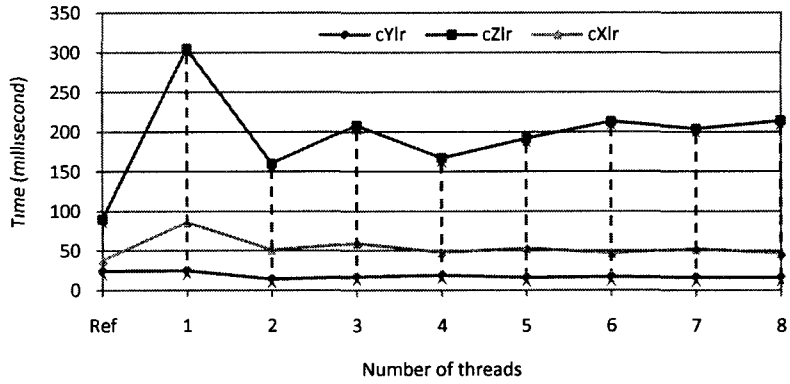


Fig. 11. SimpleAdd_s compilers comparison.

As shown in the examples, the compiler performance varies from application to another, and thus, we cannot assert a deterministic decision on the overall performance of the compiler. Code design has a significant impact on the performance, and at the same time, we notice that the same ranking for each compiler is preserved across different scenarios. Hence, there are fundamental code differences that are inherent in the design of each compiler, in which they propagate constantly to affect the performance of the compiled program. And that's reflected in the ranking of each compiler as reported in both examples where cYlr registers the lowest timing, next cXlr and then comes cZlr. cZlr has a very sporadic behavior when number of threads is greater than the number of cores as

shown in Figures 11, 12 and 13, in which formulation (11) verifies this unbalanced performance Whereas, cYlr and cXlr are highly regular and almost constant with slight variations

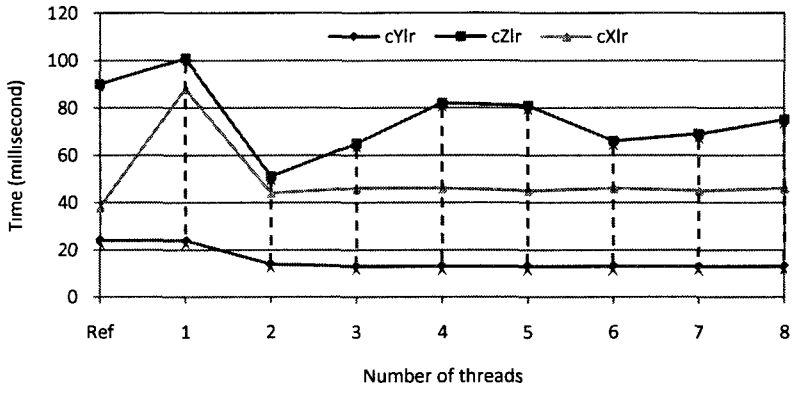


Fig. 12. SimpleAdd_g compilers comparison

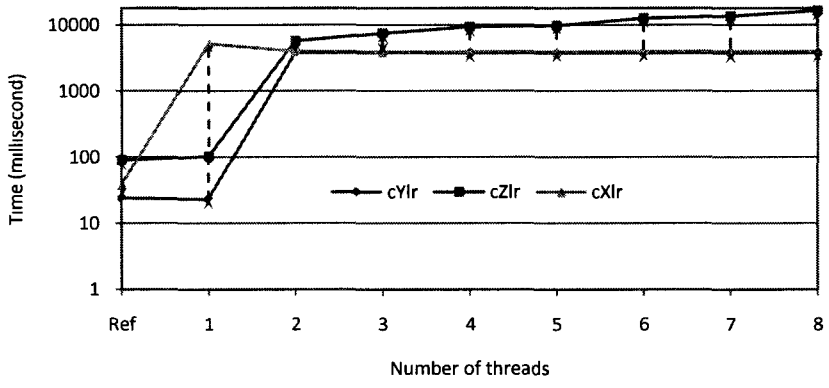


Fig. 13. SimpleAdd_d compilers comparison

The compilers pattern detection algorithm provides an insight into the actual behavior of the compilers from high level perspective As shown in Figure 14, each corner point represents a state, and a line originating from the centre joining any state means that a 1 has been set, otherwise 0 cYlr and cXlr exhibit the same pattern (cYlr is on the same line as cXlr represented with dot inside the

square). We can even map more than one application on the same graph by changing the encoding number, in which a visual parallel demonstration is revealed to reason about the behavior of the compilers performance. This method helps in finding similarities between compilers, from state-to-state, which facilitate the identification of the problematic state.

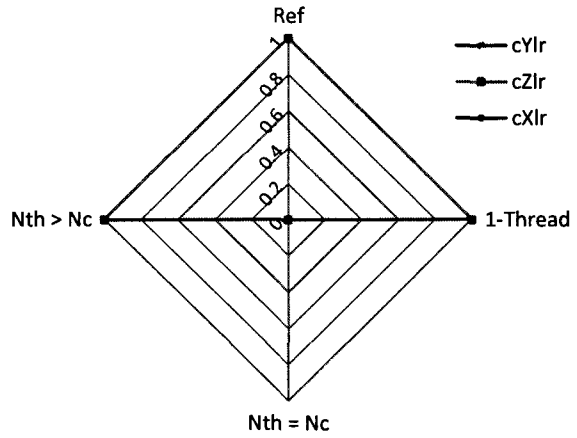


Fig. 14. Compilers pattern recognition (SimpleAdd_s).

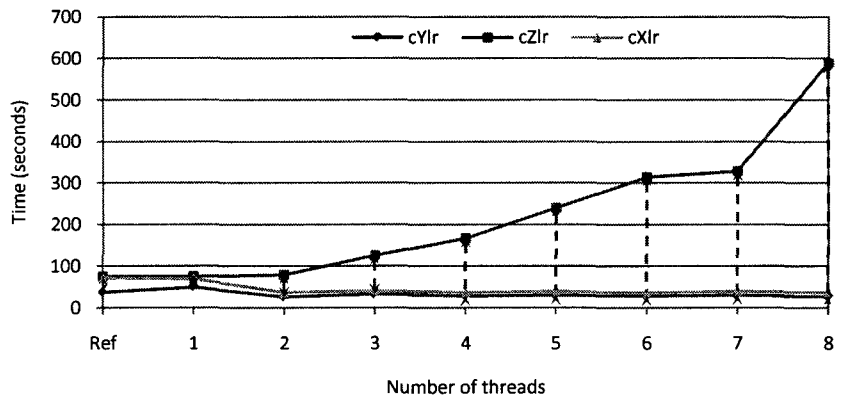


Fig. 15. MD compilers comparison.

Figure 15 shows the performance of MD example across the three compilers. The behavior of this example matches the model formulations. However, we restrict ourselves to point only on a very important aspect of cZlr compiler The α and β states are equal to γ state and certainly this is an out of phase behavior for a parallelizing compiler. For this to happen, a defect in the compilation process must have hindered the compiler parallelizer on this specific program. Therefore, the code design pattern poses a great challenge on the compiler optimizer to produce correct and efficient code. This kind of behavior is part of formulation (11) conditional evaluation.

This shows that it is not enough to provide a performance analysis based only on the evaluation of the overhead incurred by the OpenMP directives as in the case of EPCC microbenchmarks. It is very important that we evaluate each program as a separate case study so that a quantified metrics are used to provide an amalgamated performance analysis about the overall behavior of the compiler. And this is one of the goals of the proposed model.

V. 4 Results Analysis for Different Experimentation Setup

This section contains the results for MD and N-Queens programs on different hardware and software configurations. The compilers used are: PGI Workstation 10 1 (64-bit), Intel C++ 64 Compiler Professional Version 11.1 054 Build 20091130 and Microsoft (R) C/C++ Optimizing Compiler Version 15.00 21022.08 for x64. Microsoft Windows Vista (6 0) 64-bit Home Premium Edition Service Pack 2 (Build 6002) was used on an Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz with 8 GB's of rams, 4 x 32KB L1 D-Cache, 4 x 32KB L1 I-Cache (both 8-ways set associative, 64-byte line size) and 4MB L2 Cache (8-way set associative, 64-byte line size). Both programs were compiled as 64-bit binaries.

The purpose of this section is to examine all the states definitions in the model including the Intermediate state I_{γ}^{β} How the same programs behave on different hardware/software configurations? How the compilers performance and ranking differ with respect to different

hardware, versions and software configurations? Finding the optimum behavior for a given OpenMP program?

It is important to note that it is not possible to literally compare the results of this section to the previous experiments. However, we can extrapolate some interesting differences as in the case of MD example and especially in the case of cZlr compiler when α and β states are equal to γ state (Figure 15 versus Figure 16). The newer version of cZlr compiler solved this out of phase behavior. Also to note the asymmetrical relational differences between the states across the three compilers in the case of N-Queens example (Figure 17) such that, despite the fact that cYlr, cZlr and cXlr compilers register some differences in α and β states, they equal each other in γ and ζ states. We believe that this is a 64-bit-dependent behavior. However, the differences are minimal and the Intermediate state I_{γ}^{β} predicts the time decreasing convergence of each compiler performance as it approaches γ state

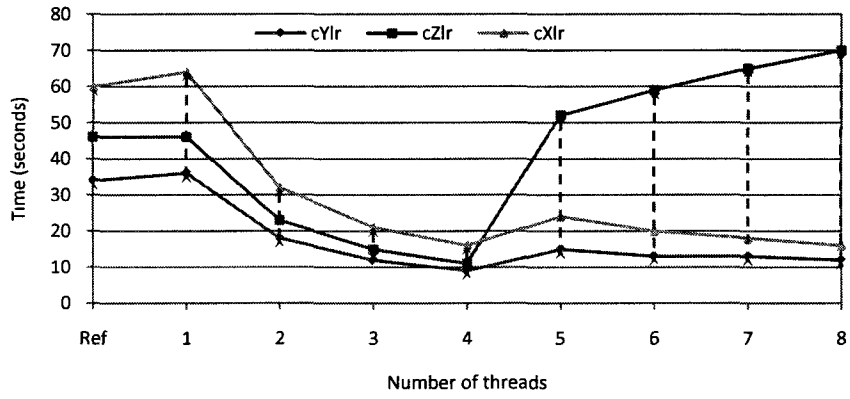


Fig. 16. MD 64-bit compilers comparison on Intel® Core™2 Quad.

Section IV 12 presented the characterization of the optimum performance for a given OpenMP program. Such performance is depicted in Figure 16 and Figure 17 with varying degrees of fluctuations.

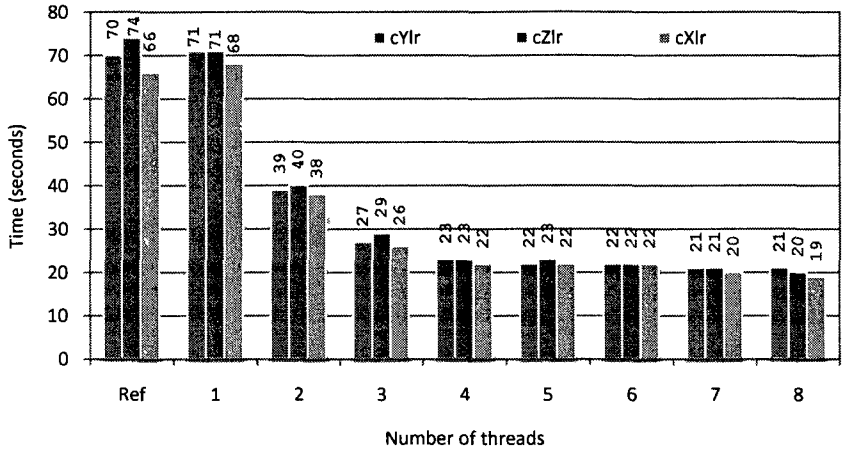


Fig. 17. N-Queens 64-bit compilers comparison on Intel® Core™2 Quad.

CHAPTER VI

"He who controls the present, controls the past He who controls the past, controls the future" - George Orwell

CONCLUSION AND FUTURE WORK

This thesis presented a new high level compiler comparative model to evaluate OpenMP programs on different compilers A model is defined using set theory notations so that it can be improved to encompass any additional formulations while allowing the expression of formulations in two very flexible methods using either strict set notations or mutable ones that can adapt to different situations The thesis presents enough formulations that simplify the classification of OpenMP programs for single and multiple compilers, hence giving the developer the choice to select the best compiler to use in a very short time and effortlessly The model has a high degree of expressiveness allowing it to be completely programmed and automated for generating a report to summarize all the results in an instructive way

The experimentation verified the applicability of the model and aided the discovery of how the performance of one of the three commercial compilers varied severely under three OpenMP programs Another notable result is the compiler optimization and code generation capabilities that influenced the overall performance of the application and not only the OpenMP runtime library as shown in the experimentation Also, the thesis details a compiler pattern detection algorithm which operates on states to find if the compilers are behaving similarly

The improved version of the performance metrics proved to be very flexible and more accurate to consider when analyzing OpenMP programs

Furthermore, the model presented here concludes with a very important finding, that is, the mathematical characterization of the optimum performance helps in classifying the evolutionary behavior for any OpenMP program. This helps in accurately determining the problematic state.

The model works at a higher level of abstraction as compared to other dedicated performance analysis tools. It does not replace the tools which provide low level profile information at the directive level. Hence, using specially designed tools for performance analysis of OpenMP programs is a must to understand the real causes of any performance degradations. However, the difficulty of debugging and analyzing OpenMP parallel programming programs increases tremendously as the program size increases.

Future work includes experimenting with different hardware and software architectures (e g. AMD, Intel Itanium, and different Operating Systems) to investigate the performance of some of the compilers on different non-targeted platforms However, in our case, since Intel compiler takes advantage of its own architecture while PGI does not, we tried all the optimization features/options available with PGI that take advantage of Intel architecture but no differences were noticed.

TBB (Threading Building Blocks) [41] and Intel Cilk++ [42] [43] libraries are two similar parallel programming languages to OpenMP It would be an interesting study to see if the model maps properly to these two languages. However, a deep knowledge is required about the features and implementation of each library in order to examine the validity of the current model formulations in comparison with the two libraries. The initial attempts presented here confirm the possibility of extending the model to work with TBB and Intel Cilk++ libraries. Nonetheless, an extensive experimentation is needed to ensure the generality of any extension on the model (e g. scheduling differences).

Another possibility to extend the model formulations is by building a knowledge base based on the program usage of each OpenMP directive and construct. On the other hand, it is possible to add additional formulations by conducting large amount of experimentations on different compilers and

OpenMP programs. This will enrich the model to provide more detailed analysis on each compiler performance.

Section IV.9 explained how to map the model states to the compiler options selection process that tries to find the best possible set of optimization options that maximize the performance. Currently, we are working on extending Analysis of Compiler Options via Evolutionary Algorithms (ACOVEA) [42] framework to support more than one compiler as well as automating the comparison process across the four states.

In conclusion, the proposed analytical model serves its purpose efficiently as proved in the experimentation section. It is hoped that the model and results will help OpenMP developers in their experimentations to choose the best OpenMP enabled compiler in an easy way.

BIBLIOGRAPHY

[1] OpenMP Application Program Interface, Version 3.0, OpenMP Architecture Review Board, May (2008).

[2] Intel Software Network. (2009) Intel VTune Performance Analyzer with Intel Thread Profiler, http://software.intel.com/en-us/intel-vtune/

[3] Intel Software Network (2009) Intel Parallel Studio, http://software.intel.com/en-us/intel-sdp-home/

[4] Tian, x, Girkar, M., Bik, A. and Saito, H. (2005) Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs. The Computer Journal, **4**(5), 588-601

[5] Reid, F. J. L. and Bull, J. M. (2004) OpenMP Microbenchmarks Version 2.0 Proceedings of the 6th European Workshop on OpenMP (EWOMP'04), Stockholm, Sweden, October 18-22.

[6] Bull, J. M (1999) Measuring Synchronisation and Scheduling Overheads in OpenMP. In Proceedings of the First European Workshop on OpenMP (EWOMP'99), Lund, Sweden, September 30-October 1

[7] Pierattini, S. (2002) Note on OpenMP performance. Corso di Sviluppo e Programmazione SGI Seriale e Parallela, v0 9, http://www.afs.enea.it/grafica/Corsi/SGIHPC/SGIHPC.html

[8] Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W. B. and Parady, B. (2001) SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. Proceedings of the International Workshop on OpenMP Applications and Tools (WOMPAT'01), West Lafayette, IN, USA, July 30-31, *LNCS* **2104**, 15-23, Springer-Verlag, Berlin.

[9] The Portland Group. (2009) PGI Unified Binary, http://www.pgroup.com/resources/unified_binary.htm

[10] Schouten, D., Tian, X., Bik, A. and Girkar, M. (2003) Inside the Intel Compiler. Linux Journal, **2003**(106). p.4, February 2003.

[11] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P. And Zhang G. (2009) The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems, **20**(3), 404-418.

[12] Dimakopoulos, V. V., Leontiadis, E. and Tzoumas, G. (2003) A Portable C Compiler for OpenMP V.2.0. Proceedings of the Fifth European Workshop on OpenMP (EWOMP'03), Aachen, Germany, September 22-26, 5-11.

[13] Chapman, B. and Huang, L. (2009) OpenMP Under The Hood. Proceedings of the Fifth International Workshop on OpenMP (IWOMP'09), Dresden, Germany, June 3-5.

[14] Chapman, B., Jost, G. and Pas, R. V. D. (2008) Using OpenMP Portable Shared Memory Parallel Programming. The MIT Press, Cambridge, Massachusetts.

[15] Tian, X. and Girkar, M. (2004) Effect of Optimizations on Performance of OpenMP Programs. Proceedings of the Eleventh International Conference on High Performance Computing (HiPC'04), Bangalore, India, December 19-22, *LNCS* **3296**, 133-143, Springer-Verlag, Berlin.

[16] Muller, M. (2001) Some Simple OpenMP Optimization Techniques. Proceedings of the Second International Workshop on OpenMP Applications and Tools (WOMPAT'01), West Lafayette, IN, USA, July 30-31, *LNCS* **2104**, 31-39.

[17] Quinlan, D., Schordan, M , Yi, K. and de Supinski, B. R. (2003) A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives Proceedings of the Fourth International Workshop on OpenMP Applications and Tools (WOMPAT'03), Toronto, Canada, June 26-27, *LNCS* **2716**, 13-25, Springer-Verlag, Berlin.

[18] Yonezawa, N., Wada, K., and Aida, T. (2006) Barrier Elimination Based on Access Dependency Analysis for OpenMP. Proceedings of the Fourth International Symposium on Parallel and Distributed Processing and Applications (ISPA'06), Sorrento, Germany, December 4-6, *LNCS* **4330**, 362-373, Springer-Verlag, Berlin.

[19] Brunschen, C. and Brosson, M (2000) OdinMP/CCp - A portable implementation of OpenMP for C. Concurrency: Practice and Experience, **12**, 1193-1203.

[20] Ayguadé, E., Gonzàlez, M , Labarta, J., Martorell, X., Navarro, N. and Oliver, J. (1999) NanosCompiler: A Research Platform for OpenMP Extensions. Proceedings of the First European Workshop on OpenMP (EWOMP'99), Lund, Sweden, September 30-October 1.

[21] Intel Software Network (2009)Intel C++ Compiler Professional Edition, http.//software.intel.com/en-us/intel-compilers/

[22] The Portland Group. (2009) PGI C++ Workstation, http://www pgroup com/products/workpgcc htm

[23] Microsoft Corporation. (2009) Microsoft Visual Studio 2008 Team Edition, http://msdn microsoft.com/en-us/teamsystem/default aspx

[24] Aslot, V. (2001) Performance Characterization of the SPEC OMP Benchmarks. Master's thesis, Purdue University.

[25] Mohr, B , Malony, A. D , Hope, H-C , Schlimbach, F., Grant, H., Hoeflinger, J. and Shah, S (2002) A Performance Monitoring Interface for OpenMP. Proceedings of the Fourth European Workshop on OpenMP (EWOMP'02), September 18-20, Roma, Italy.

[26] Bui, V , Hernandez, O , Chapman, B. and Kufrin, R. (2006) An OpenMP run-time Performance Analyzer. Proceedings of the ACM Conference for Languages, Compilers, and Tools for Embedded Systems (DIGPLAN/SIGBED), Ottawa, Ontario, Canada, 61-64.

[27] Furlinger, K. and Gerndt, M (2005) ompP: A Profiling Tool for OpenMP. In Proceedings of the First International Workshop on OpenMP (IWOMP'05), Eugene, Oregon, USA, June 1-4, *LNCS* **4315**, 15-23, Springer-Verlag, Berlin.

[28] FZJ-ZAM-IB-2001-08 (2001) Knowledge Specification for Automatic Performance Analysis APART Technical Report Julich Supercomputing Centre (JSC), Germany.

[29] Gerndt, M , Mohr, B. And Traff, J L. (2004) Evaluating OpenMP Performance Analysis Tools with the APART Test Suite. Proceedings of the 10th International Euro-Par Parallel Processing Conference (Euro-Par'04), Pisa, Italy, August 31-September 3, *LNCS* **3149**, 155-162, Springer-Verlag, Berlin.

[30] Suleman, A M., Qureshi, K M. and Patt, N. Y. (2008) Feedback-Driven Threading Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs. Proceedings of the 13th International Conference on Architectural Support for

Programming Languages and Operating Systems (ASPLOS'08), Seattle, Washington, USA, March 1-5, *ACM SIGPLAN Notices*, **36**(1), 277-286.

[31] Marr, D T, Binns, F, Hill, D., Hinton, G, Koufaty, D., Miller, J. A. and Upton, M. (2002) Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal, **6**(1), 4-15.

[32] Tian, X, Bik, A., Girkar, M., Grey, P., Saito, H. and Su, E. (2002) Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology· Implementation and Performance. Intel Technology Journal, **6**(1), 36-46

[33] Magro, W., Petersen, P. and Shah, S (2002) Hyper-Threading Technology. Impact on Compute-Intensive Workloads. Intel Technology Journal, **6**(1), 58-66.

[34] Pinkers, R. P J., Knijnenburg, P. M. W., Haneda, M and Wijshoff H A. G (2004) Statistical Selection of Compiler Options. Proceedings of the IEEE 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04), Volendam, The Netherlands, October 4-8, 494–501, IEEE Computer Society.

[35] Haneda, M, Knijnenburg, P. M W. and Wijshoff, H A. G. (2005) Generating New General Compiler Optimization Settings. Proceedings of the 19th annual International Conference on Supercomputing (ICS'05), Cambridge, Massachusetts, USA, June 20-22, 161–168. ACM Press, New York, NY

[36] Cavazos, J., Fursin, G., Agakov, F, Bonilla, E., O'Boyle, M F. P. and Temam, O. (2007) Rapidly Selecting Good Compiler Optimizations using Performance Counters Proceedings of the International Symposium on Code Generation and Optimization (CGO'07), San Jose, California, USA, March 11-14, 185-197, IEEE Computer Society.

[37] Marowka, A. (2008) Performance of OpenMP Benchmarks on Multicores Processors Proceedings of the Eighth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'08), Cyprus, June 9-11, *LNCS* **5022**, 208-219, Springer-Verlag, Berlin.

[38] Bane, M. K. and Riley, G. D (2002) Extended Overhead Analysis for OpenMP. Proceedings of the Eighth International Euro-Par Conference (Euro-Par'02), Paderborn, Germany, August 27-30, *LNCS* **2400**, 1-16, Springer-Verlag, Berlin.

[39] Yongjian, C., Dingxing, W. and Weimin, Z. (2003) Extended Overhead Analysis for OpenMP Performance Tuning. Proceedings of the Fourth International Workshop on OpenMP Applications and Tools (WOMPAT'03), Toronto, Canada, June 26-27, *LNCS* **2716**, 160-169, Springer-Verlag, Berlin.

[40] Burkardt, J. (2010) Molecular Dynamics using OpenMP (MD_OPEN_MP), http://people.sc fsu edu/~burkardt/cpp_src/md_open_mp/md_open_mp.html

[41] Intel Software Network. (2010) Intel Threading Building Blocks 2.2, http://www.intel com/software/products/tbb/

[42] Leiserson, C. E (2009) The Cilk++ concurrency platform. Proceedings of the 49th Automation Design Conference (DAC'09), San Francisco, CA, USA, July 26-31, 522-527. ACM Press, New York, NY.

[43] Blumofe, R D., Joerg, C. F., Kuszmaul, B C., Leiserson, C. E., Randall, K. H. and Zhou, Y. (1995) Cilk· An Efficient Multithreaded Runtime System. Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, Santa Barbara, California, United States, July 19-21, 207-216, *ACM SIGPLAN Notices*, **30**(8), 207-216

[44] Ladd., S. R.. (2010) Analysis of Compiler Options via Evolutionary Algorithm ACOVEA. http://www.coyotegulch.com/products/acovea/index.html

VITA AUCTORIS

NAME: Mohammed F. Mokbel

PLACE OF BIRTH: Kfarhatta, Lebanon

YEAR OF BIRTH: 1984

EDUCATION: University of Windsor

Windsor, Ontario, Canada

2007-2010 M.Sc. Computer Science

Lebanese International University

Majdelyoun, Lebanon

2003-2006 B.Sc. Computer Engineering