View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons

Parallel programming issues and what the compiler can do to help

Sara Royuela*, Xavier Martorell†
Barcelona Supercomputing Center*†, Universitat Politècnica de Catalunya†
{sara.royuela, xavier.martorell}@bsc.es

Abstract- *Twenty-first century parallel programming models are becoming real complex due to the diversity of architectures they need to target (Multi- and Many-cores, GPUs, FPGAs, etc.). What if we could use* one programming model to rule them all, one programming model to find them, one programming model to bring them all and in the darkness bind them, in the land of MareNostrum where the Applications lie. *OmpSs programming model is an attempt to do so, by means of compiler directives.*

Compilers are essential tools to exploit applications and the architectures the run on. In this sense, compiler analysis and optimization techniques have been widely studied, in order to produce better performing and less consuming codes.

In this paper we present two uses of several analyses we have implemented in the Mercurium[3] source-to-source compiler: a) the first use is to help users with correctness hints regarding the usage of the OpenMP and OmpSs tasks; b) the second use is to be able to execute OpenMP in embedded systems, with very little memory, thanks to calculating the Task Dependency Graph of the application at compile time. We also present the next steps of our work: a) extending range analysis for analyzing OpenMP and OmpSs recursive applications, and b) modeling applications using OmpSs and future OpenMP4.1 tasks priorities feature.

Keywords- *Compiler, Static Analysis, Task Dependency Graph, OpenMP, OmpSs, Embedded System*

I. INTRODUCTION

Static and dynamic analysis and optimization techniques are widely used in order to enhance performance and power consumption. Dynamic techniques benefit from a non-restricted knowledge of the application (addresses and values of all variables are known), but they require the execution of the program along with the instrumentation library. This means adding overhead, as well as having results tied to a specific execution. Other data-sets or architectures may change the results of the analysis. On the contrary, static techniques have limited information (values of the variables and pointer aliasing situations), but have the benefit of being effortless from the point of view of the programmer, as well as being valid for any input data. Neither of the options is perfect nor valid for everything, and they can indeed be combined for better results.

II. CORRECTNESS

Although programming models such as OpenMP and OmpSs are tantalizing due to its simplicity and scalability, they also bring forth difficulties when it comes to fully exploit their capabilities. The compiler can be crucial to anticipate bugs that may be very hard to find at runtime. We focus on the OpenMP and OmpSs tasking models to define a set of situations that may cause: a) a runtime failure, b) a loss of performance, or c) a non-deterministic result. We present different cases the compiler is able to detect and the hints it provides to the programmer. To understand these cases some background in the memory model of OpenMP [1] is needed.

A. Automatic storage variables as shared. Automatic storage variables are allocated and deallocated automatically when the program flow enters and leaves the enclosing code block. When the compiler detects that such a variable may be accessed after its scope has been exited, it proposes the following solutions: a) changing the data-sharing attribute from `shared` to `private`, for basic data types; b) adding a `taskwait`, for arrays and structures.

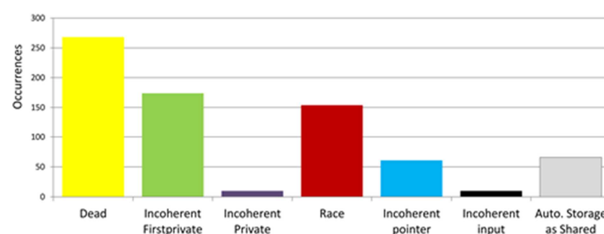
B. Data-race situations. Data-races occur when two or more threads access shared data and at least one of the accesses is a write. When the compiler detects such a situation, it proposes the following solutions: a) protecting the accesses with a `critical` or an `atomic` construct, b) adding a `taskwait` between the uses.

C. Incoherent data-sharing. The data-sharing attribute of a variable must be coherent with its usage. We check three situations, in the following order: a) variables defined within a task and never used in that task, but used after the synchronization of the task, should be `shared`; b) `private` variables in a task should be defined before being read, otherwise they must be `firstprivate`; c) `firstprivate` variables in a task should not be defined before being read, otherwise they must be `private`.

D. Incoherent dependencies. Task dependencies are used to impose an order in the execution of the tasks. Dependable objects must be coherent with the usage of those objects in the task. We check three situations: a) objects accessed via pointer should specify the dependency in the accessed storage, instead of the pointer; b) `input` dependences should be read and never written within the task; c) `output` dependences should be written within the task, and should not be read before being written.

We have tested this work with over 70 students and 5 benchmarks on different courses, and the results are shown in Figure 1. Most errors captured by the compiler are related with the default data-sharing attributes because users forget to explicitly change them. The less common errors are those caused by users explicitly defining incorrect data-sharing attributes. Dependences errors are not common because only one of the five benchmarks includes task dependences.

FIGURE 1
Occurrences of each Correctness Mistake

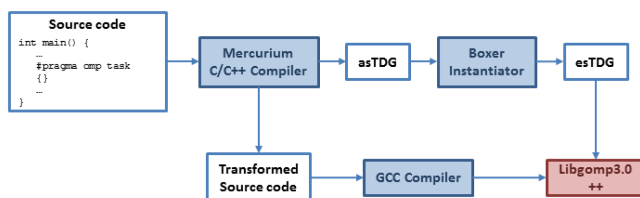


We also have compared our results with those of the Oracle Solaris Studio 12.3 (OCS12.3) compiler for the three first cases, because it does not implement support for task dependences. Since OCS12.3 does not provide hints about performance, Mercurium suggests more accurate solutions in case A. OCS12.3 may report wrong messages for case B, which can be solved by enclosing the analyzed task in a parallel construct. Finally, OCS12.3 does not consider declaring a variable as private when its initial value is not read, so it may result in a loss of performance in case C.

V. STATIC TASK DEPENDENCY GRAPH

Tasks dependencies impose an order in the execution of the tasks. OpenMP and OmpSs runtimes build a Task Dependency Graph (TDG) ensuring that order. Building the graph at runtime is feasible for HPC systems, where large amounts of memory with a reasonable performance penalty are available. Nonetheless, it may be impossible for embedded systems, where memory can be very limited. The Programming Models and the Real Time teams at BSC have united their forces to build a tool-chain able to derive a TDG statically. The tool-chain is shown in Figure 2.

FIGURE 2
Offline Tool-chain for an Static TDG Creation

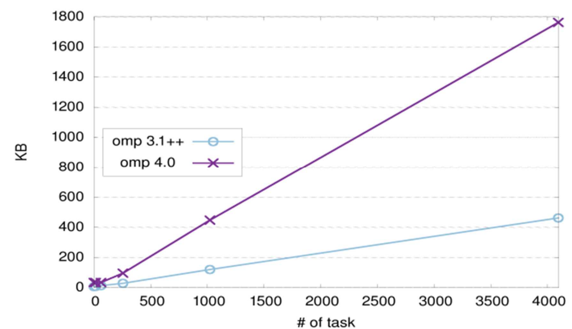


Generating the TDG statically requires a series of steps. First of all, the Mercurium compiler generates a Parallel Control Flow Graph [4] (PCFG) that extends the common CFG with parallelism information. Then induction variables analysis is executed to discover the evolution of each loop. Finally, range analysis [5] extended with support for parallelism is used to calculate the value of the variables at each point of the program. All this information is used to generate an *augmented static TDG* (asTDG) containing: a) one node per each task/taskwait/barrier construct, b) all possible synchronization edges among these nodes, along with predicates defining the conditions for the edges to exist, and c) information about all control flow statements (conditionals and loops) involved in the execution of the previously mentioned constructs.

After that, the Boxer Instantiator expands the asTDG obtaining the complete TDG that will be executed at runtime. This is performed in two steps: 1) expansion of the control flow structures (i.e. decide the number of iterations of each loop statement, and the branches taken in each conditional statement); 2) check of the dependency predicates to decide the task instances that have actual dependences. This process results in an *expanded static TDG* (esTDG).

Finally, the code generated by Mercurium is passed through the GCC back-end compiler to generate the binary that will run in our lightweight libgomp. The tests of our tool-chain are shown in Figure 3. The memory consumption of the runtime using our statically computed TDG scales (in terms of memory consumption) much better than the plain libgomp supporting tasks dependencies, while increasing the number of tasks.

FIGURE 3
MEMORY USAGE (IN KB) OF DIFFERENT RTLS



VI. MODELING TASK PRIORITIES

OpenMP4.1 and OmpSs include support for defining tasks priorities, thus promoting some tasks over others when all of them are ready to be executed. By using our static tool-chain to compute the TDG, we intend to model the applications, and extract patterns in which the priorities of the tasks can be decided at compile time. This is part of our ongoing work and we are looking for applications to exploit this functionality.

ACKNOWLEDGMENT

We acknowledge the Mercurium group, always helping and giving priceless opinions and contributions to this work. We also thank the Real Time group which brought up fresh ideas and push forward this collaboration.

LIST OF PUBLICATIONS

1. (appeared) R. Ferrer, S. Royuela, D. Caballero, A. Duran, X. Martorell, E. Ayguadé. “Mercurium: Design Decisions for a S2S Compiler”. *Cetus Users and Compiler Infrastructure Workshop, PACT*, 2011.
2. (appeared) S. Royuela, A. Duran, C. Liao, D.J. Quinlan. “Auto-scoping for OpenMP tasks”. *IWOMP*, 2012.
3. (appeared) S Royuela, A Duran, X Martorell. “Compiler automatic discovery of ompss task dependencies”. *LCPC*, 2013.
4. (accepted) D. Caballero, S. Royuela, R. Ferrer, A. Duran, X. Martorell. Optimizing Overlapped Memory Accesses in User-directed Vectorization. *ICS*, 2015.
5. (accepted) S. Royuela, R. Ferrer, D. Caballero, X. Martorell. “Compiler Analysis for OpenMP Tasks Correctness”. *CF*, 2015.
6. (submitted) R. Vargas, S. Royuela, MA Serrano, E. Quiñones. “A Lightweight OpenMP Run-time for Embedded Systems”. *ICCAD*, 2015.

REFERENCES

[1] OpenMP ARB. “OpenMP API, v.4.0”, July 2013, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
[2] S. Royuela, R. Ferrer, D. Caballero, X. Martorell, “Compiler Analysis for OpenMP Tasks Correctness”, *CF15*.
[3] BSC. “The Mercurium compiler”, http://pm.bsc.es/mcxx.
[4] S. Royuela, A. Duran, and X. Martorell, “Compiler Analysis and its application to OmpSs”, *Master’s Thesis at UPC*, 2012.
[5] R. Ernani, V. H. Sperle, F. M. Quintão, “A fast and low-overhead technique to secure programs against integer overflows”, *CGO13*.