Automatic Profiling for Climate Modeling

Anja Gerbes[1], Nabeeh Jumah[2], Julian Kunkel[3]

[1]*Center for Scientific Computing (CSC)* [2]*Universität Hamburg* [3]*Deutsches Klimarechenzentrum (DKRZ)*

Abstract

Some applications are time consuming like climate modeling, which include lengthy simulations. Hence, coding is sensitive for performance. Spending more time on optimization of specific code parts can improve total performance. Profiling an application is a well-known technique to do that.

Many tools are available for developers to get performance information about their code. With our provided python package *Performance Analysis and Source-Code Instrumentation Toolsuite (PASCIT)* is a automatic instrumentation of an user's source code possible. Developers mark the parts that they need performance information about.

We present an effort to profile climate modeling codes with two alternative methods.

- usage of GGDML translation tool to mark directly the computational kernels of an application for profiling.
- usage of GGDML translation tool to generate a serial code in a first step and then use LLVM/Clang to instrument some code parts with a profiler's directives.

The resulting codes are profiled with the LIKWID profiler. Alternatively, we use perf and OProfile's ocount & operf to measure hardware characteristics. The performance report with a visualization of the measured hardware performance counters in generating Radar Charts, Latex Tables, Box Plots are interesting for scientist to understand the bottlenecks of their codes.

Introduction

- Code profiling helps in identifying algorithmic bottlenecks, so called hot spots.
- Hot spots are code regions which take the most execution time, the most common source are loops.
- An example of such time consuming loops are computational kernels in climate modeling.
- These stencil computational kernels are in general memory-bound.
- Identifying which parts of the code are responsible for the critical hot spots is important, making a hot spot faster can have a big pay-off.
- In a previous work *Intelligent selection of compiler options to optimize compile time and performance*, we presented an effort in which we explored the compilation process.
  - Different combinations of optimization flags of a compiler including processor-specific flags
  - Provided optimized compilation time and maintained the application performance
  - Automatic detection of compiler flags for performance optimization
- In the current work, we prepare the subject application for profiling with LIKWID.
  - Automatic instrumentation and profiling of scientific code
  - Two different methods are used to prepare the code for profiling
  - A comparison of the profiling results with the two methods is made

Methodology

- A simple test climate model application is written in C language with GGDML extensions.
- GGDML source-to-source translation tool transforms the code into two different code versions:
  - A code that is prepared with OpenMP pragmas & LIKWID markers, which is ready to be profiled.
  - A serial code without profiling markers & multithreading pragmas, which has to be marked with LLVM/Clang.
- The resulting two code versions are profiled with LIKWID, and the results are compared.

GGDML Translation

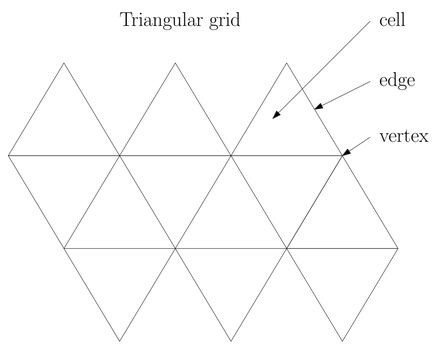
- General Grid Definition & Manipulation Language (GGDML) source-to-source translation tool of AIMES Project is a higher-level set of language extensions to support earth system modeling.
- It allows to develop models from the scientific perspective, not machines perspective.
- Applications developed with GGDML need to be translated with a special translation tool.
- The translation process is highly flexible.
- This allows us
  - to configure the process to parallelize the code with OpenMP.
  - to mark the code for LIKWID to make it ready for profiling.
  - to generate serial code to be processed by Clang for profiling.

Clang Tooling

- The interesting thing in using Clang is to modify the Abstract Syntax Tree (AST) on-the-fly to use the toolsuite for performance analysis and source-code instrumentation.
- The structure of the AST is representing the logical structure of a source code, because an AST contains the stored position information of every element in this code.
- By reimplementing the RecursiveASTVisitor template class, we can specify which AST nodes we are interested in by overriding relevant methods.
- The visitor design pattern can be used to reach every node of the tree and perform some action when the process comes to a given type of nodes.

Test Application

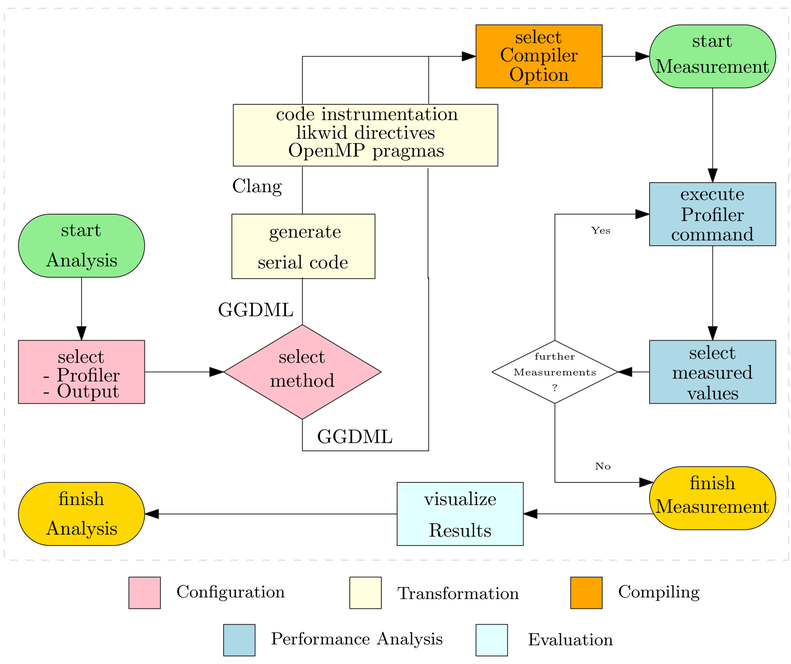
- An Atmospheric/Climate modeling code.
- It runs iteratively in multiple time-steps.
- It executes different model components.
- Components execute a variety of kernels.
- Kernels apply stencil operations to compute a variables' values over the grid.
- Those kernels are compute intensive codes.
- The application's performance is determined by the performance of those kernels.
- Kernels exploit parallel execution capabilities of the processors using OpenMP.
- Test code targets triangular icosahedral grid.
- The model variables are measured/computed at the cell centers or on the edges of the cells.



Profiler

- CPU Profiling is necessary to understand CPU usage quickly and completely
- Perf, Oprofile and LIKWID are a performance measurement and profiling system to identify and analyze the hot spots.
- Perf, Oprofile and LIKWID follows the UNIX design philosophy of „one task, one tool".
- Typical questions during performance analysis:
  - Which parts of the program take the most execution time?
  - Do the number of software or hardware events indicate an actual performance issue to be fixed?
  - How can we fix the performance issue?
- LIKWID is not build on top of the Linux perf interface; this allows LIKWID to provide full support for new architectures quick and independent from special kernel versions.
- LIKWID supports hardware based events and metrics to each hardware counter, this is more than just CPU clock cycles.
- Compared LIKWID to Oprofile
  - Oprofile does not require any special compilation flags, or even recompiled code.
  - Oprofile supports less counters.
  - Oprofile & LIKWID supports multi-threaded code.
  - Oprofile & LIKWID sees all processes and will find bottlenecks in other places.
  - Oprofile & LIKWID requires kernel support.

Analysis Toolsuite



Our provided analysis toolsuite is named *PASCIT* and run the following steps, depending on which method was chosen:

Steps of Clang Method:

1. Generate serial code using GGDML method
2. Global search & replace on a bunch of C++ files
3. Renaming the file, which is using for Source-to-Source-Instrumentation
4. Modify the makefile
5. Build the makefile
6. Instrument the C++ code

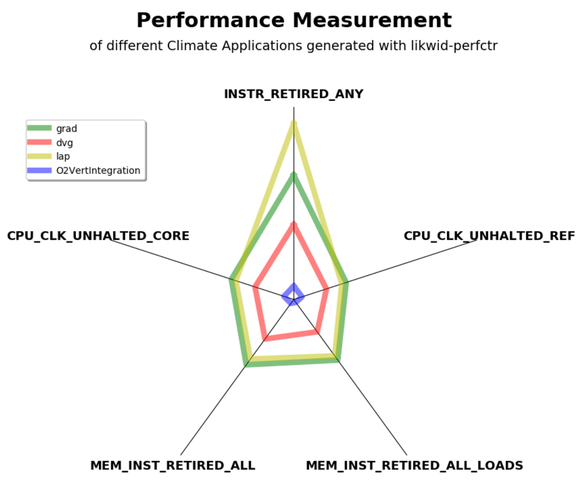
Steps of GGDML Method:

1. Parse GGDML-based kernels into an AST
2. Apply some optimization procedures
3. Generate serial code for Clang method
4. Instrument the kernels
5. Translation of DSL enriched code to pure host language

Performance Analysis & Evaluation Step:

1. Compile instrumented code
2. Profile code
3. Generating output file
   - Profiler's Output ⇒ CSV ⇒ Slices ⇒ Radar Chart | Latex Table | Box Plots | Rose Diagram | CSV | JSON

Results



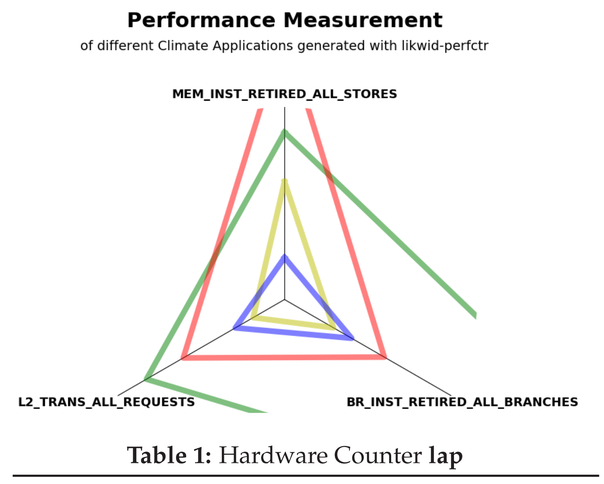


Fig. 1: Radar Charts

Experimental Setup
Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
Ubuntu 17.10, Compiler (gcc-6, clang-3.9)
Profiler (Likwid-4.3.1, Oprofile-1.1.0, Perf-4.4.98)

Successful automatic instrumentation & profiling

Table 1: Hardware Counter **lap**

INSTR_RETIRED_ANY	18266880
CPU_CLK_UNHALTED_CORE	6323082
CPU_CLK_UNHALTED_REF	5245592
MEM_INST_RETIRED_ALL	7647261
MEM_INST_RETIRED_ALL_LOADS	7247121
MEM_INST_RETIRED_ALL_STORES	428434
L2_TRANS_ALL_REQUESTS	131287
BR_INST_RETIRED_ALL_BRANCHES	205629

Conclusion

- Both tools were successful in marking the code for profiling.
  - GGDML source-to-source translation tool based on higher semantics of the language extensions.
  - Clang source-to-source instrumentation tool based on the clang node type.
- The two methods could be used for collecting profiling information.

Future Work

- Automatic identification of bottlenecks
- Identification of HPC code patterns suitable for optimization
- Optimization of LLVM translation for HPC relevant patterns
- Compiler-level optimization of typical HPC code structures
- *PASCIT's* functionality as a black box for other scientific application domains
- Extension of python package *PASCIT*
- Support more types of AST nodes

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) (GZ: LU 1353/11-1).