Using CAASCADE and CrayPAT for Analysis of HPC Applications

Reuben D. Budiardja, M. Graham Lopez, Oscar Hernandez, Jack Wells
Oak Ridge National Laboratory

Jisheng Zao, Vivek Sarkar
Georgia Tech
Motivation

How to answer these questions:

• What (combinations of) numerical libraries, compilers, parallelization methods are used by applications and need to be supported (by vendors, center, …) ?

• What are relative priorities of Fortran, C, C++, and which features of the language standard (e.g. F2003, F2008, C++03, C++11) need better support?

• Which applications use OpenMP and/or OpenACC?

• Which OpenMP and OpenACC features are used most, and are most urgent for implementers to verify and optimize?

• Which applications use mixed language programming (e.g Fortran and C++)? which language “drives” the other?
Motivation

Slightly harder questions:

• How should OpenMP or OpenACC address “deep copy”?  

• How are application using communication libraries (1-sided, bulk transfers, asynchronous task-based, …) ? What communication libraries are used and need better hardware support?

Perfectly reasonable questions, insufficient ways to get quantitative information
Motivation

Answers may direct:

- Standardization efforts
- Compiler and library implementation and optimization efforts
- System and architecture design
Where We Are Now

Source code

- surveys, interviews, "institutional knowledge"
  - staff-time intensive!

Runtime performance

- automated
- on-demand

System monitoring

- automated
- transparent to the user

XALT
CrayPAT
RUR
STAT
Where We Need To Be

Goals:
- Gather data in a reproducible, automated (non-human), and transparent way
- Make it useful to humans

Compilers and linkers know everything “knowable’ about the source code
CAASCADE: Compiler-Assisted Application Source Code Analysis and Database
Compiler (GCC) Intermediate Representation

GCC front ends:
- language-specific code (follow language specifications)
- parser
- genericizer
- optimizer 1
- optimizer 2
- optimizer N
- RTL generator
- code generator

GCC middle end:
- language and machine independent code
- genericizer
- gimplifier
- optimizer 2
- ... optimizer N

GCC back ends:
- machine-dependent (from architecture descriptions)

Executable binary

GCC components
- plugin callback
- application data path
- GCC components

CAASCADE

source code

gcc plugins

app metadata database

app metadata database

gcc plugins
Extracted Program Information

Compile Event and code metadata

- compiler version
- programming language/model (string)
- module/class/typedef
- main program name
- line numbers

Application structure

- subroutine name
- number of exec statements
- loops
- max loop nest
- call statements (int)
- call chain (list)
- total use modules (int)
- module variables (int)
- module variables (list)
- module subroutines (list)
- symbols (int)
- symbols in other namespaces (int)
- subroutines (int)
- namelists (int)
- statements (int)
- statement types
- module usage
- standard usage
- call site arguments (string)
Application data structures

- variables (int)
- array variables (int)
- co-array variables (int)
- pointer variables (int)
- contiguous variables (int)
- target variables (int)
- allocatable variables (int)
- artificial variables (int)
- asynchronous variables (int)
- optional variables (int)
- dummy variables (int)
- protected variables (int)
- volatile variables (int)
- abstract variables (int)
- implicit type variables (int)
- in namelist variables (int)
- external variables (int)
- parameters (int)
- common block variables (int)
- derived types symbols (int)
- derived types with components (int)
- derived types with direct components (int)
- derived types with indirect components (int)
- derived types with array components (int)
- derived types with allocatable components (int)
- derived types with pointer components (int)
- derived types recursive (int)

Parallelization

- OpenMP directives (int)
- statements inside OpenMP (int)
- OpenMP threadprivate variables (int)
- OpenMP UDR variables (int)
- OpenMP declare target variables (int)
- OpenACC directives (int)
- statements inside OpenACC (int)
- contains subroutines (bool)
- OpenACC subroutine (bool)
- OpenACC declare create variables (int)
- OpenACC declare copyin variables (int)
- OpenACC declare deviceptr variables (int)
- OpenACC declare
device_resident variables (int)
- OpenACC declare link variables (int)
CAASCADE: High Level View

Data synthesis (compiler plugins)

Source
Parse
Gimplify
CFG
RTL
ASM

Representation (database)

HPC co-design database

code metadata
CAASCADE on Titan

- "module load caascade" with PrgEnv-gnu
- Wrapped "g++" → "g++ -fplugin=caascade_c.so ...", "gfortran" → "gfortran -fplugin caascade_f.so ..."
- Wrapped linker (ld) to collect CAASCADE generated JSON-formatted data for every object file
- Leverage XALT transmission mechanism to store data (e.g. directly to DB, via syslog, HTTP broker, or file)
- **Works transparently** (no changes in application build process)
<table>
<thead>
<tr>
<th>Executable</th>
<th>Link Program</th>
<th>Build User</th>
<th>Build Date</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>RayleighTaylor_Cray_GNU</td>
<td>gfortran</td>
<td>reubenbd</td>
<td>2018-04-09 22:28:45</td>
<td>titan</td>
</tr>
<tr>
<td>RayleighTaylor_Cray_GNU</td>
<td>gfortran</td>
<td>reubenbd</td>
<td>2017-08-10 14:13:17</td>
<td>titan</td>
</tr>
<tr>
<td>RayleighTaylor_Cray_GNU</td>
<td>gfortran</td>
<td>reubenbd</td>
<td>2017-08-10 14:11:42</td>
<td>titan</td>
</tr>
</tbody>
</table>

```
program_reactor_longation_simmer
  preparation
  reaction

measurement_form:initial->from_m
measurement_form:initial->m

show_command:showinteger.m
  copy_command:copyread.m
  show_command:showcharacter.m
```

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Source Object</th>
<th>Source Name</th>
<th>Compile Command</th>
<th>Build User</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basics.a</td>
<td>Runtime.o</td>
<td>Runtime.o</td>
<td></td>
<td>reubenbd</td>
<td>titan</td>
</tr>
<tr>
<td>Basics.a</td>
<td>PROGRAM_HEADER_longation.o</td>
<td>PROGRAM_HEADER_longation.o</td>
<td></td>
<td>reubenbd</td>
<td>titan</td>
</tr>
<tr>
<td>Basics.a</td>
<td>GetMemoryUsage_Command.o</td>
<td>GetMemoryUsage_Command.o</td>
<td></td>
<td>reubenbd</td>
<td>titan</td>
</tr>
</tbody>
</table>

```bash
make -j8
make -j8
make -j8
```
Results: GenASiS
An astrophysics simulation framework written in Fortran
Distribution of Fortran Language Standard

**Static**
- Fortran 95: 9.7%
- Fortran 90: 57.5%
- Fortran 2003: 32.8%

**Dynamic**
- Considers runtime information
- Runs production job with `perftools-lite`, generate profile with `pat_report`
- Uses profile to re-weight compiler plugin output → get a new distribution

Program units (modules, subroutines) requiring the specified minimum language standard to compile
Distribution of Fortran Language Standard

What (Fortran) language standard gets used the most?

**Static**

- Fortran 95: 9.7%
- Fortran 90: 32.8%
- Fortran 2003: 57.5%

**Dynamic**

- Fortran 95: 95.7%
- Fortran 90: 4.3%
- Fortran 2003: 0%

Program units (modules, subroutines) requiring the specified minimum language standard to compile
Program units (modules, subroutines) requiring the specified minimum language standard to compile
Derived types gets written the most into the code, yet scalars and arrays contribute the most during execution.
Classification of Executable Statements

**Static**

- ASSIGN
- CALL
- IF
- DO
- END_PROCEDURE
- DEALLOCATE
- BLOCK
- POINTER_ASSIGN
- ALLOCATE
- RETURN
- SELECT_TYPE
- SELECT
- INIT_ASSIGN
- OMP_PARALLEL_DO
- WRITE
- IOLENGTH
- EXIT
- READ
- DO_WHILE
- PPC

**Dynamic**

- ASSIGN
- DO
- OMP_PARALLEL_DO
- END_PROCEDURE
- IF
- WHERE
- BLOCK
- CALL
- SELECT_TYPE
- CYCLE
- POINTER_ASSIGN
- SELECT
- ALLOCATE
- DEALLOCATE
- RETURN
- IOLENGTH
- INIT_ASSIGN
- WRITE
- EXIT
- DO_WHILE
QMCPACK: Data Types Distribution
Many-body Quantum Monte Carlo code (C++)

Static

Dynamic

- arrays: 60.1%
- pointers: 29.6%
- allocatables: 9.3%
- derived types: 11.9%
- scalars: 11.3%
- 51.6%
- 25.1%
E3SM: OpenMP Statements

- OMP_DO
- OMP_PARALLEL_DO
- OMP_BARRIER
- OMP_MASTER
- OMP_PARALLEL
- OMP_CRITICAL
- OMP_SINGLE
- OMP_SIMD

The chart shows the frequency of different OpenMP statements in the E3SM codebase.
Work in Progress

• Systematically answering driving questions (see “Motivation” slides)
  • Support for CUDA code
    – Using LLVM (and also for for C, C++, Fortran)
  • Tackling the “a.out problem”
    – Information from IR can be used as code signatures
  • Integrate more runtime information
    – supporting other runtime-based tools / profilers in agnostic way
  • Support from and integration with other compilers
    – PGI started with -Msummary
    – would love similar feature from Cray (CCE)
  • Motifs detection (dense LA, sparse LA, spectral, structure or unstructured grids, …)