Using CAASCADE and CrayPAT for Analysis of HPC Applications

Reuben D. Budiardja, M. Graham Lopez,
Oscar Hernandez, Jack C. Wells
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
{reubendb, lopezmg, oscar, wellsjc}@ornl.gov

Jisheng Zao, Vivek Sarkar
Georgia Tech, Atlanta, GA, USA

***Abstract*—In this paper we describe our work on integrating** CAASCADE **with CrayPAT to obtain both *static* and *dynamic* information on characteristics of high-performance computing (HPC) applications.** CAASCADE **— *C*ompiler-*A*ssisted *A*pplication *S*ource *C*ode *A*nalysis and *D*atabas*e*—is a system we are developing to extract application parallel programming language features from its source code by utilizing compiler plugins. CrayPAT enable us to add runtime-based information to** CAASCADE**'s feature detection. We present results from analysis of HPC applications.**

I. Introduction

Understanding characteristics of high-performance computing (HPC) applications quantitatively is essential for many reasons, such as informing the co-design efforts for hardware and software ecosystem, developing and implementing future language standards and directives, prioritizing development of compiler and library features, and developing algorithms more amenable to parallelization on future hardware platforms. These characteristics may include traits relatively straightforward to gather, such as the programming languages used by an application and the external libraries it depends on, to traits that are more embedded into the application, and hence harder to characterize from the onset, such as programming motifs, memory-access patterns, and whether the code is amenable to low-level (compiler) optimizations.

However, so far there has not been a comprehensive and systematic way to gather these quantitative data on applications even for characteristics that should be straightforward to capture. The typical ways of data collection—including written user surveys and manual data collection—are often labor intensive, error prone, and easily outdated as applications develop. On the other hand, new tools adoption by application programmers is very hard, if the tool is not production quality or part of their existing application development process.

To address this issue, we are developing a system for analyzing HPC applications called CAASCADE: **C**ompiler-**A**ssisted **A**pplication **S**ource **C**ode **A**nalysis and **D**atabas**e**. CAASCADE is a system that collects information about how applications are constructed directly from the source code and stores this information in a database for further analysis. Programming paradigm, language standards and directives, parallelization method, and external library dependencies are only a few examples of application feature detection that we can do with CAASCADE. Results are stored into a standardized relational database transparently to users, enabling future queries and aggregate analysis across a broad-range of applications. In an earlier work, we demonstrated CAASCADE's ability in

extracting features of several HPC applications [1].

Although *static* (i.e. compile-time) analysis such as this by itself is already useful to obtain quantitative data on applications, a connection to results obtained via a *dynamic*, runtime analysis on typical production jobs for such applications would even be more valuable. One way to do this is by integrating information collected by a runtime-based tool such as CrayPAT.

Within this paper, we show our results in incorporating CrayPAT information to augment CAASCADE capability for analyzing source code of HPC applications. The rest of the paper will be organized as follows. In §II we give a brief overview of CAASCADE and describe how it works. In §III we show how we installed CAASCADE on a Cray XK7 system, the OLCF Titan supercomputer. We present examples of features and statistics we can extract from HPC applications using CAASCADE in §IV. Our concluding remarks and a brief discussion of our current future plans are in §V.

II. CAASCADE Overview

CAASCADE consists of a mechanism for gathering information directly from application source code and a mechanism for storing this information into a database. For maximum accuracy and flexibility, CAASCADE gathers its information from the intermediate representations (IR) of production compilers such as the GNU Compiler Collection (GCC)[1] compiler at their highest level of representation that is close to the source code. Once collected from the compiler, statistics and detailed source code characteristics are assembled at link time and stored in a back-end database in order to most generally support further analysis. This approach requires minimal intervention from the application developers as all they need is to turn on/off the collection via a module installed on the system.

In the following subsections we describe in more details the information gathering and storing mechanisms in CAASCADE.

[1]https://gcc.gnu.org/

A. Compiler Plugins

The reference version of the data-gathering component of CAASCADE is implemented using the plugin facility exposed by recent versions of the GCC. These plugins are loaded as dynamic libraries at the time of compiler invocation, and as they passively gather data from various internal transformation passes, they do not modify the resulting executable instructions, do not require debugging mode support, and work with any level of optimization enabled. By wrapping the plugin load at invocation and producing binaries that are suitable for production runs, CAASCADE gathers information about production applications being used on the system in a completely transparent way to the user.

To understand an application's behaviors and language usage, it is necessary to represent the application as a set of features that can serve as inputs to the analytic tools. The types of program features vary, including syntax level features, control-flow, and data-flow information. From the static analysis point of view, these features lie in different levels of the compiler's intermediate representations (IR). Thus, building the feature collection mechanism inside a compiler and intercepting information from the different levels of IR is a feasible approach.

As stated, we built our feature collector as GCC compiler plugins, which can intercept the information from any level of GCC IR we specify. In this work, we are focusing on the syntax level features, so our plugins work at abstract syntax tree (AST) level; the plugins' feature collector traverses the AST to gather the information that we need.

The extracted information should be rich enough to help post-analysis (e.g. a static analyzer) build the program structure graph that hierarchically presents the major program constructs, including functions, loop nests, and parallel constructs. To establish the program structure graph, the call graph is built and loop nests, parallel constructs, and calls are presented as tree structures in each function or program unit.

The two major programming languages we initially focus on for our developing system are C++ and Fortran. We therefore have built plugins for the following compilers.

1) GNU g++: For C/C++ application, we start at the global namespace level and recursively collect the following features:

- The class information, which includes namespace, class fields, functions, and class hierarchy information; The virtual functions are identified to help build the call graph properly;
- The function features, which includes:
  - The function signatures: function name, return type, argument types;
  - The number of statements;
  - The number of arithmetic expressions;
  - A set of loops, each of which includes the loop type (i.e. `do-while`, `while`, or `for` loop), loop iterators, level of loop nest, the connection to the parent node which can be a loop, a parallel construct or function body;
  - A set of parallel constructs, each of which includes the OpenMP and OpenACC parallel tasks (e.g. `parallel for` loop), each parallel construct should be connected to its parent node which can be a loop, a parallel construct, or function body;
  - A set of call sites, each of them includes the target function if it is available, the receiver class type, arguments' types, and the parent node which can be a loop, a parallel construct, or function body.

2) GNU gfortran: For the Fortran applications we collect the following: modules, submodules, common blocks and procedures symbols, variables, and constants. Additionally, we collect information about executable statements (and their operands and where they belong), directive-based programming models, and call-sites.

Specifically, for the different components of a Fortran program we gather the following information:

- For the program symbols we collect their names, their types and attributes.
- For variables, we collect their scope (i.e. the namespace they belong to) and their type (i.e. primitives, derived-types, classes, or arrays) along with the 'kind' information. Furthermore, we also collect the following variables' attributes:
  - For arrays, we capture their dimensions and sizes (if known statically), and if their types are explicit, deferred, assumed size or rank, or implied shape. We also capture if the arrays are Fortran co-arrays and which are their co-dimensions.
  - For characters, we collect the characters symbols and their lengths
  - Symbol attributes for variables such as: allocatables, artificial, asynchronous, dummy arguments, contiguous, external, optional, pointer, protected, volatile, threadprivate, target of a pointer, result, entry, ISO_BINDING, etc.
- For executable statements, we collect the operand information and classify these operand into the language standard to which they belong. These include type statements. The Table I contains the type of statements we detect and how we classify them into the different Fortran standards. We categorize them by the first time they appeared in the Fortran standards starting from Fortran 77 to Fortran 2008. The table reflects the statement types that *gfortran* can detect as of GCC version 7.2.

Additionally, the plugin is able to detect OpenMP and OpenACC directives, the different types of directives in OpenMP 4.5 and OpenACC 2.5, the number of statements inside the lexical extent of the directive region, and callsites called inside parallel regions.

Statement Type	Fortran Standard	GCC opcode
Event Post	Fortran Ext	EXEC_EVENT_POST
Event Wait	Fortran Ext	EXEC_EVENT_WAIT
Nested Block	Fortran 2008	EXEC_END_NESTED_BLOCK
Critical	Fortran 2008	EXEC_CRITICAL
Error Stop	Fortran 2008	EXEC_ERROR_STOP
Concurrent	Fortran 2008	EXEC_DO_CONCURRENT
Sync All	Fortran 2008	EXEC_SYNC_ALL
Sync memory	Fortran 2008	EXEC_SYNC_MEMORY
Sync Images	Fortran 2008	EXEC_SYNC_IMAGES
Lock	Fortran 2008	EXEC_LOCK_UNLOCK
Form Team	Fortran 2008	EXEC_FORM_TEAM
Change Team	Fortran 2008	EXEC_CHANGE_TEAM
Sync Team	Fortran 2008	EXEC_SYNC_TEAM
Fail Image	Fortran 2008	EXEC_FAIL_IMAGE
FLush	Fortran 2003	EXEC_FLUSH
Wait	Frotran 2003	EXEC_WAIT
Select Type	Fortran 2003	EXEC_SELECT_TYPE
Assign Call	Fortran 2003	EXEC_ASSIGN_CALL
Exit	Fortran 2003	EXEC_EXIT
Type Bound Procedure Call	Fortran 2003	EXEC_CALL_PPC
Computed Call	Fortran 2003	EXEC_COMPCALL
Select	Fortran 95	EXEC_SELECT
Forall	Fortran 95	EXEC_FORALL
Transfer	Fortran 95	EXEC_TRANSFER
Pointer assignment	Fortran 90	EXEC_POINTER_ASSIGN
Init Assignment	Fortran 90	EXEC_INIT_ASSIGN
Where	Fortran 90	EXEC_WHERE
Cycle	Fortran 90	EXEC_CYCLE
Allocate	Fortran 90	EXEC_ALLOCATE
Deallocate	Fortran 90	EXEC_DEALLOCATE
Block	Fortran 77	EXEC_BLOCK
Assignment	Fortran 77	EXEC_ASSIGN
Label Assignment	Fortran 77	EXEC_LABEL_ASSIGN
Goto	Fortran 77	EXEC_GOTO
Call	Fortran 77	EXEC_CALL
Return	Fortran 77	EXEC_RETURN
Entry	Fortran 77	EXEC_ENTRY
Pause	Fortran 77	EXEC_PAUSE
Stop	Fortran 77	EXEC_STOP
Continue	Fortran 77	EXEC_CONTINUE
If	Fortran 77	EXEC_IF
Arithmetic If	Fortran 77	EXEC_ARITHMETIC_IF
Do	Fortran 77	EXEC_DO
Do While	Fortran 77	EXEC_DO_WHILE
End Procedure	Fortran 77	EXEC_END_PROCEDURE
Open	Fortran 77	EXEC_OPEN
Close	Fortran 77	EXEC_CLOSE
Read	Fortran 77	EXEC_READ
Write	Fortran 77	EXEC_WRITE
IO Length	Fortran 77	EXEC_IOLENGTH
Backspace	Fortran 77	EXEC_BACKSPACE
End File	Fortran 77	EXEC_ENDFILE
Inquire	Fortran 77	EXEC_INQUIRE
Rewind	Fortran 77	EXEC_REWIND

Table I: Types of executable statements captured by CAASCADE and classification by Fotran Standards

B. Database Backend

The information gathered by the compiler plugins is formatted as self-describing JSON[2]. The JSON string is then compressed with `zlib`[3], encoded with Base64 algorithm[4], and embedded back into the object file resulting from the compilation itself as an ELF section header. To avoid name collision of the section header when multiple object files are archived into a `.a` library file, the section name is prepended with the object file name, which should be unique in an archive. This technique of embedding the CAASCADE compiler plugin data into the object file ensures that the information is as 'portable' (in the sense of availability) as the object file itself.

An executable is built by linking multiple object files together by the *linker*. It is in this linking phase that we have the opportunity to collect all the data from the compiler plugins that are already embedded into the object files making up the executable. To do this, we leverage the mechanism employed by XALT [2]. XALT collects software and library usage by intercepting the linker and transmitting the information through a choice of transmission mechanism to be stored in a database. We extend XALT's linker-wrapper and transmission method to additionally go through all object files involved in the linking step, extract the previously embedded CAASCADE plugin data, and transmit the data for eventual database storage.

Figure 1 shows the database schema of the relevant XALT tables with a new table containing source code information gathered by CAASCADE compiler plugins. When a new executable is produced by a linker, a new entry is added to the table `xalt_link` with a new `link_id`, which contains information about the executable such as the builder, the system on which it was built, and the absolute path of the executable. The table `xalt_object` stores information about an object file as uniquely identified by the object path and its SHA-1 hashes[5]. If an object file does not exist in this table during linking, a new entry is created. Since many object files are needed to build an executable, the one-to-many relationship between the executable in `xalt_link` table and `xalt_object` table is represented by the table `join_link_object`.

As a new entry is created in `xalt_object` table, if that object file contains CAASCADE plugin information, a new entry is also created in table `caascade_source`. In particular, the JSON string containing the information from the compiler plugin is stored in the column `source_proginfo`, while other meta-data information about the source code are stored in other columns.

C. Web-based Query Interface

A prototype web-based query interface has been developed within the CAASCADE system to easily extract information from our database. Within the interface, one can search for a particular application and generate prepared plots on-the-fly for the selected application. All plots used in §IV were generated from this web-based interface. Additionally, one can drill down to the individual source file from which the executable was generated to look at the JSON output generated by CAASCADE compiler plugins. Figure 2 shows a screenshot of our web-based query interface.

III. Installation

We aim to make CAASCADE work as transparently as possible to the users. Users should be able to just load the appropriate modulefile for CAASCADE to collect data seamlessly without further changes to the application's build system or workflow. Although CAASCADE is not currently deployed in production yet on our systems, this 'just-works' principle guides our development practice.

The GCC compilers require that the command-line parameter `-fplugin=file` to be specified to load a compiler plugin. To avoid exposing users to this requirement, we provide scripts to wrap

[2]https://www.json.org/

[3]http://zlib.net/

[4]https://tools.ietf.org/html/rfc3548.html

[5]http://www.faqs.org/rfcs/rfc3174.html

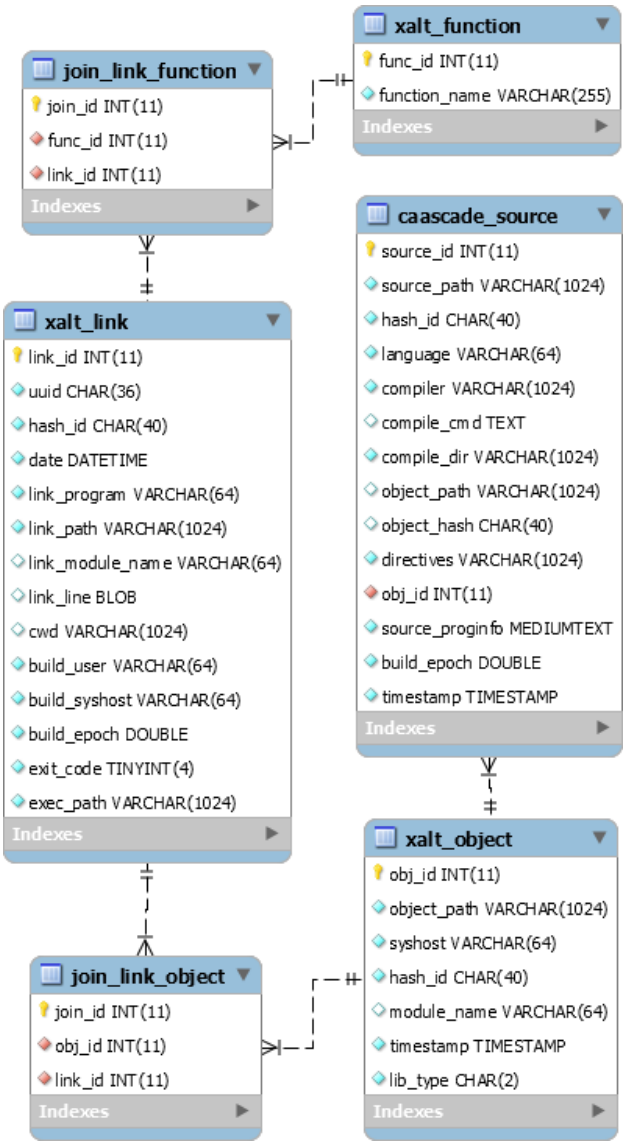

Figure 1: Database schema of a subset of XALT tables with CAASCADE source table.

the calls to the real compiler commands (e.g. `gfortran`, `g++`). The wrapper scripts add the required command-line option. Listing 1 shows our wrapper script for the `gfortran` compiler. The compiler plugins have to be built for each GCC versions we want to use. The wrappers provide the facility to select the correct plugin version.

On OLCF Titan, we provide modulefile that prepend environmental variable `$PATH` with the location of the wrapper script. The modulefile also

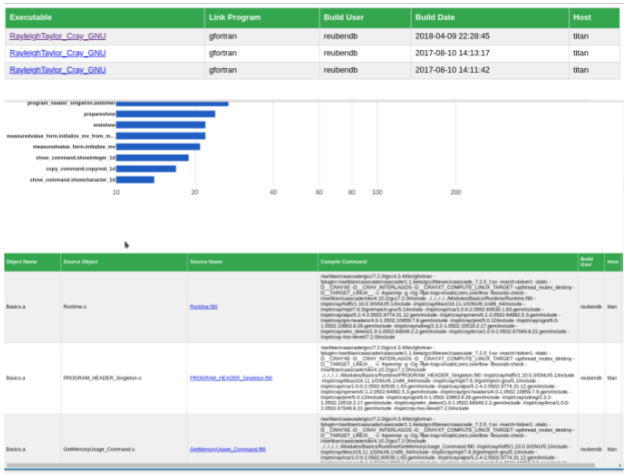

Figure 2: Screenshot of CAASCADE web-based query interface.

Listing 1: CAASCADE compiler wrapper for `gfortran`

```
my_path=$0
my_name=$(basename $my_path)
for exe in $(type -p -a $my_name); do
  if [ $exe == $my_path ]; then
    continue
  else
    realcomp=$exe
    break
  fi
done

#-- Get compiler version
version=$($realcomp --version \
  | head -n 1 | awk {'print $4'})
case "$version" in
  '6.3.0')
    plugin=caascade_6.3.0_f.so
    ;;
  '7.2.0')
    plugin=caascade_7.2.0_f.so
    ;;
  '8.0.1')
    plugin=caascade_8.0.1_f.so
    ;;
  *)
    $realcomp "$@"
    exit $?
esac
$realcomp \
  -fplugin=${CAASCADE_DIR}/libexec/$plugin "$@"
```

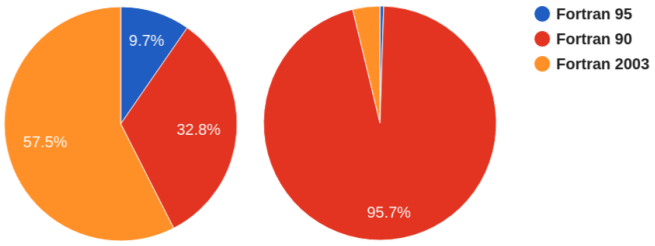

Figure 3: Distribution of Fortran language standard in GENASIS code from the static (left) and dynamic (right) analysis.

sets the environmental variable `$CAASCADE_DIR`. This setup works well even with Cray's provided compiler wrappers (e.g. `CC` for C++ and `ftn` for Fortran); a Cray's compiler wrapper calls CAASCADE's compiler wrapper which in turns calls the real compiler driver (e.g. `gfortran` or `g++`).

IV. RESULTS

In this section we show results from analyzing applications with CAASCADE as augmented by performance information from CrayPAT.

A. GENASIS

The first application we look at is GENASIS, an astrophysics simulation framework targeted at large-scale computing [3] written in Fortran.

A feature of the CAASCADE compiler plugin for `gfortran` is in identifying the minimum Fortran language standard needed to compile a program unit. Summing this up across all of the source files of an application, we can then get a distribution of the Fortran language standard used by that application. This tells us, for example, how much of the application only requires Fortran 90 standard, and how much of the application requires Fortran 2003 standard statement wise. Figure 3 shows the Fortran standard distribution for the code GENASIS.

The left plot of Figure 3 is simply the aggregate of the number of program units required the specified Fortran standard. This result comes directly from our compiler plugin which we label as *static* result.

The right plot of Figure 3 is the same aggregation except in this case we weighted the count with sampling results from CrayPAT profile. To obtain this result, we built GENASIS with `perftools-lite` and ran a test job representative of a production job. A profile output from this job can then be obtained using `pat_report`, which gives us the number of sampling for each subroutine, typically ordered by sample count. The number of sampling becomes the *weight* factor for that program unit. The *dynamic* result simply comes from multiplying the weight by the static value. To avoid completely masking program units that do not show up in CrayPAT profile, we set a floor value for the weight to 0.01.

Figure 3 shows complementary data answering slightly different facets of the question "what (Fortran) language standard gets used the most?" In this case, Fortran 2003 is required the most by the application developers and gets written into the code the most frequently. However, the most time consuming program units (i.e. what gets executed the most) are subroutines only requiring Fortran 90 standard. This is perhaps not surprising, since the major addition of Fortran 2003 standard concerns with data structure and object orientation features. These features likely increase developer's productivity in building a complex application, yet they are less useful to write computational intensive kernels.

Information on the different data types used by an application is also collected by our Fortran compiler plugin. Figure 4 shows the distribution of data types—scalars, arrays, allocatables, pointers, and derived types—in GENASIS. The contrast of data type usage for the static code (left) and at runtime (right) is apparent. Derived types gets written the most in the code yet scalars and arrays contribute the most during execution. This is in line with GENASIS development consensus that compute intensive kernels operate on mostly arrays. Derived types are also very useful for managing data structure in a complex application, however arrays and scalars are better in providing compiler the most optimization opportunity for compute intensive kernels.

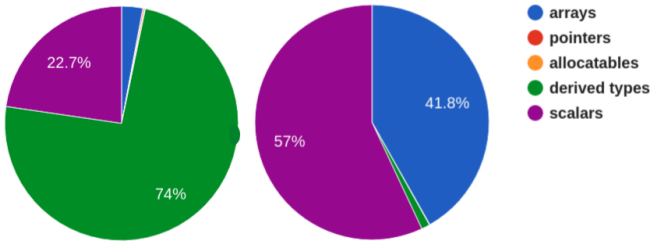

Figure 4: Distribution of data types in GENASIS code from the static (left) and dynamic (right) analysis. Derived types are used the most in the code, yet contribute the least during code execution.

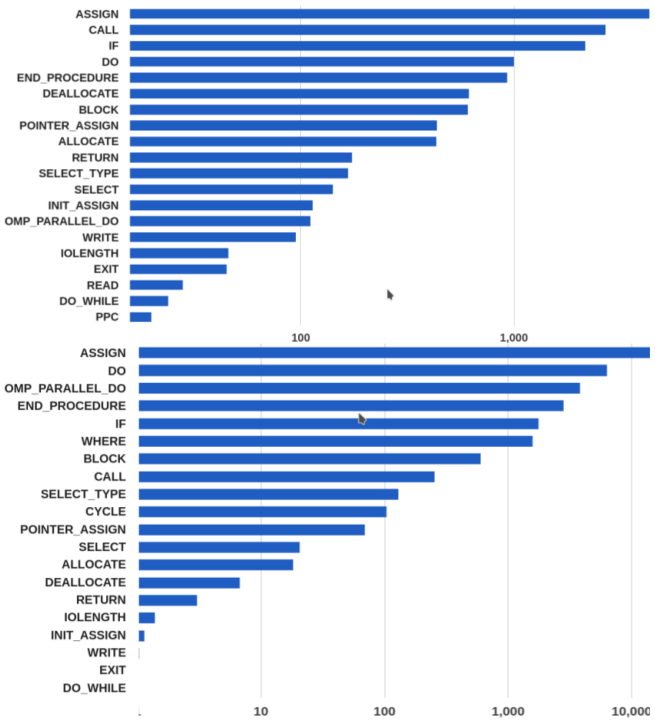

Figure 5: Frequency of classified executable statements in GENASIS code from the static (top) and dynamic (bottom) results.

In Figure 5 we compare the executable statements from static and dynamic results. Although the distribution of statements is mostly similar, dynamic result shows that loop-related operations dominates.

One way to quantify the parallelization methods employed by an application is by counting the number of calls to MPI subroutines and the

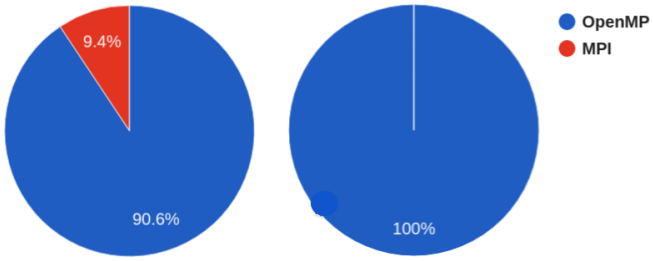

Figure 6: MPI and OpenMP parallelization method in GENASIS as written in the code (left) and as executed during runtime (right).

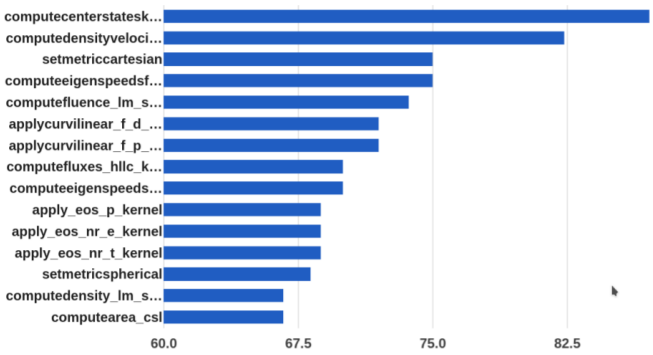

Figure 7: OpenMP coverage for subroutines in GENASIS. Only the top fifteen subroutines with the most OpenMP coverage are shown for readability.

number of executable statements inside OpenMP directives. Figure 6 shows the results from our plugin for GENASIS code. On the left plot we see that OpenMP makes up about 90 percent of the parallelization paradigm of the static code. During execution most of the time is spent on code path with on-node parallelization with OpenMP (right). Figure 7 shows the percentage of OpenMP coverage in each subroutine, although here we only show the top fifteen subroutines with most coverage. A hundred-percent coverage would mean that every statement in that program unit is within OpenMP pragma.

B. *QMCPACK*

QMCPACK is a many-body quantum Quantum Monte Carlo code for high-performance computing, written primarily in C++. Building this application with CAASCADE allows us to ex-

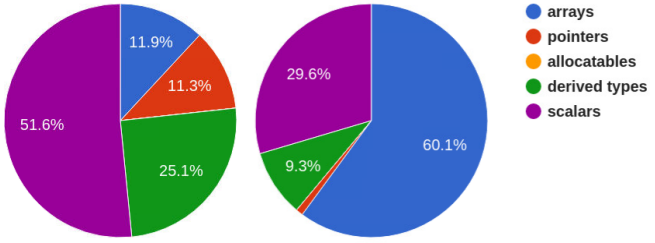

Figure 8: Distribution of data types in QMCPACK from the static (left) and dynamic (right) analysis.

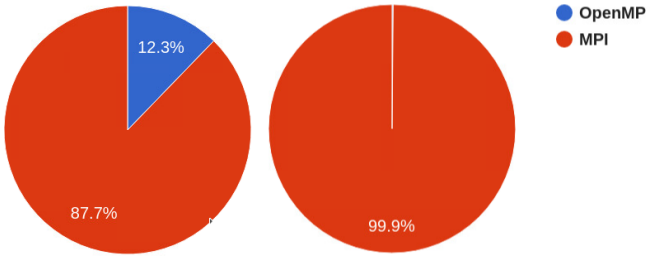

Figure 9: MPI and OpenMP parallelization method in QMCPACK as written in the code (left) and as executed during runtime (right).

ercise our C++ plugin implementation. For the runtime dynamic results, we ran the Nickel-Oxide performance test from QMCPACK with CrayPAT, which represent typical production jobs for this application.

Figure 8 shows the data types in QMCPACK as captured by CAASCADE C++ plugin. On the left we see the data types distribution as they are written in the code. On the right we see how the distribution changes as weighted by the number of sampling from CrayPAT profile data.

On Figure 9 we see the distribution of usage frequency from static and dynamic results for OpenMP and MPI. Figure 10 shows the OpenMP coverage for subroutines in QMCPACK. Note that from Figure 9 and 10 only we cannot infer the time spent in either parallelization method. Rather, they are indicators of how much of each parallelization method are used by the developer to write the application.

Figure 11 shows the different MPI calls and their usage frequency in the application.

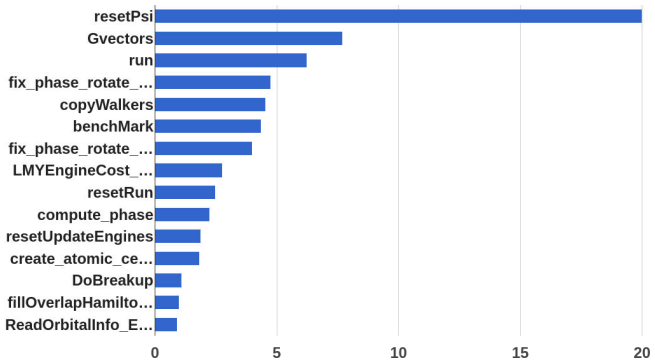

Figure 10: OpenMP coverage for subroutines in QMCPACK. Only the top fifteen subroutines with the most OpenMP coverage are shown for readability.

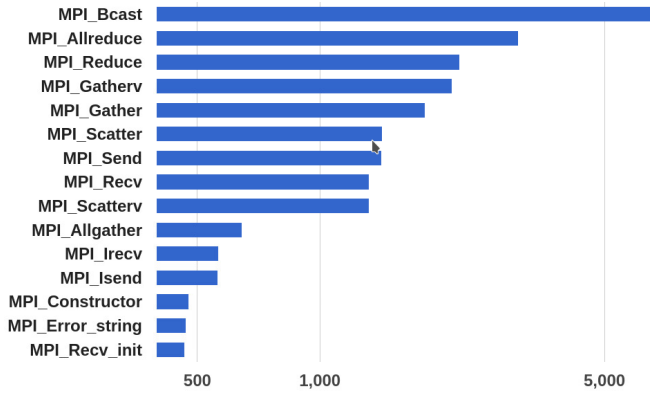

Figure 11: MPI subroutines and their usage frequency in QMCPACK.

V. Conclusion

In this paper we present a new method for analyzing characteristics of HPC application using CAASCADE, a new system we are developing. We have also shown how we use CrayPAT to augment results from CAASCADE.

The CAASCADE system allows us to store the information collected by its compiler plugins transparently to users. Storing these information into the database enables us to do further analysis as needed, including, for example, performing inter-procedural / inter-source-files analysis of an executable, aggregating data from multiple executables on a system, and searching for common programming patterns across different executables. Although we have not done all of these

analyses, we plan to do so in the near future.

Our reference implementation of the data-gathering component of CAASCADE is implemented using the GCC plugin facility. However other compilers, including the Cray Compiling Environment (CCE), are widely used by the user community. By way of this presentation, we hope to engage both vendors and the community in collaborative work such that compiler-based data-gathering capability may be available in other compilers as well.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This project is sponsored by the Laboratory Directed Research and Development (LDRD) Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy via the LDRD project 8277: "Understanding HPC Applications for Evidence-based Co-design".

REFERENCES

[1] M. Graham Lopez, Oscar Hernandez, Reuben D. Budiardja, and Jack Wells. Caascade: A system for static analysis of hpc software portfolios. *to be published, Proceedings of 6TH Workshop on Extreme-Scale Programming Tools*, 2018.

[2] Reuben Budiardja, Mark Fahey, Robert McLay, Prasad Maddumage Don, Bilel Hadri, and Doug James. Community use of xalt in its first year in production. In *Proceedings of the Second International Workshop on HPC User Support Tools*, HUST '15, pages 4:1–4:10, New York, NY, USA, 2015. ACM.

[3] Christian Y. Cardall and Reuben D. Budiardja. Genasis mathematics : Object-oriented manifolds, operations, and solvers for large-scale physics simulations. *Computer Physics Communications*, 222:384 – 412, 2018.