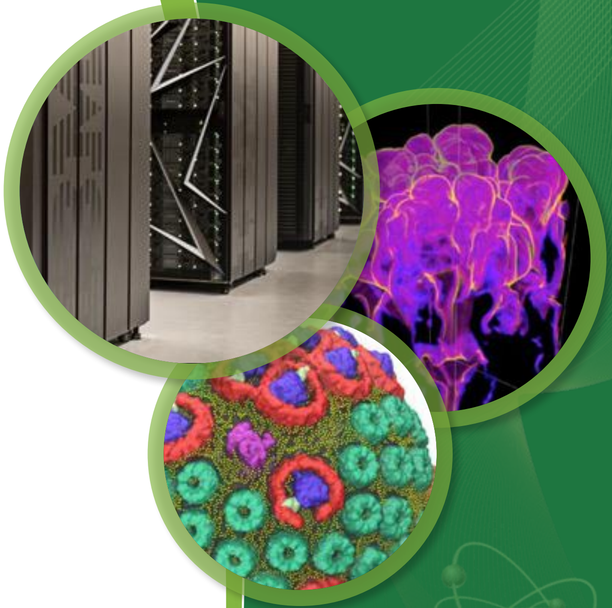
Using CAASCADE and CrayPAT for Analysis of HPC Applications

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

Jisheng Zao, Vivek Sarkar
Georgia Tech



ORNL is managed by UT-Battelle for the US Department of Energy

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.

Oak Ridge National Laboratory | Leadership Computing Facility

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?

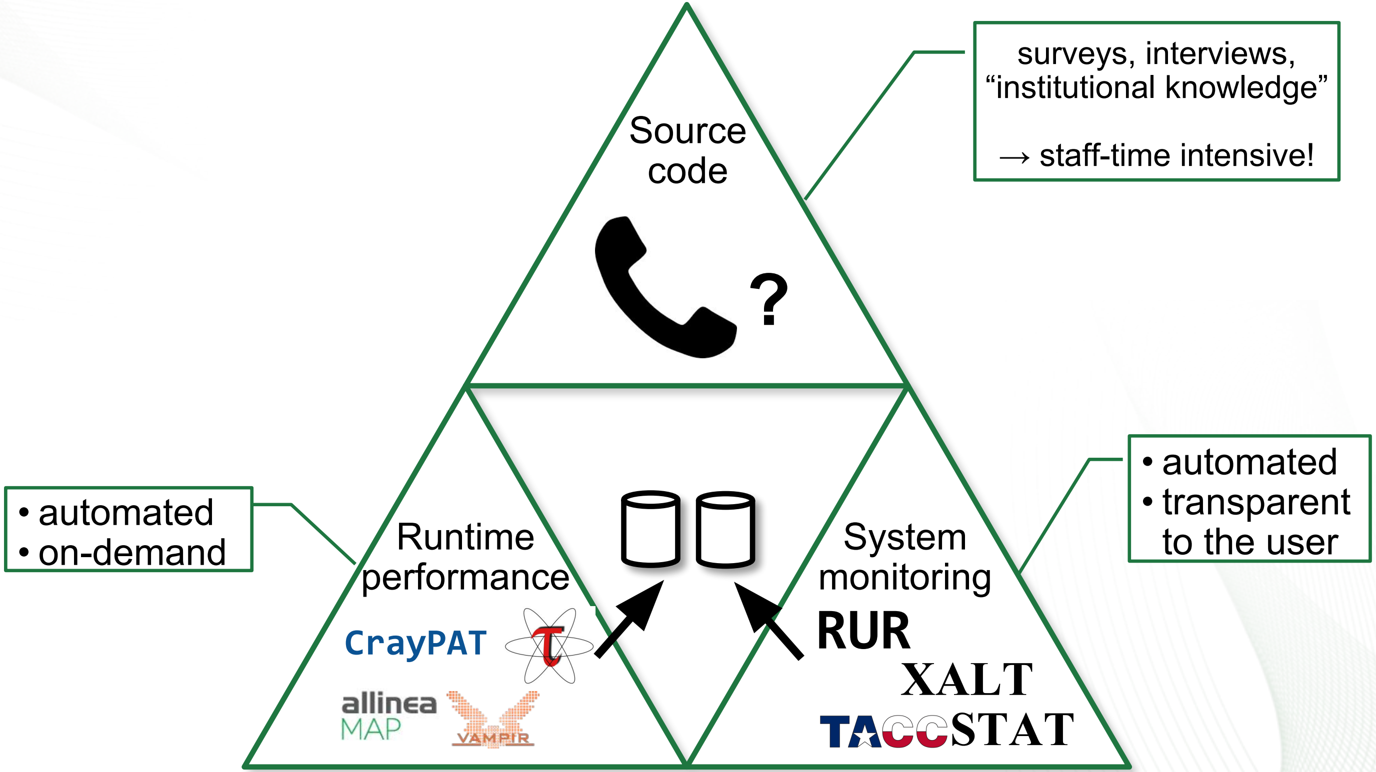
Perfectly reasonable questions, insufficient ways to get quantitative information

Motivation

Answers may direct:

- Standardization efforts
- Compiler and library implementation and optimization efforts
- System and architecture design

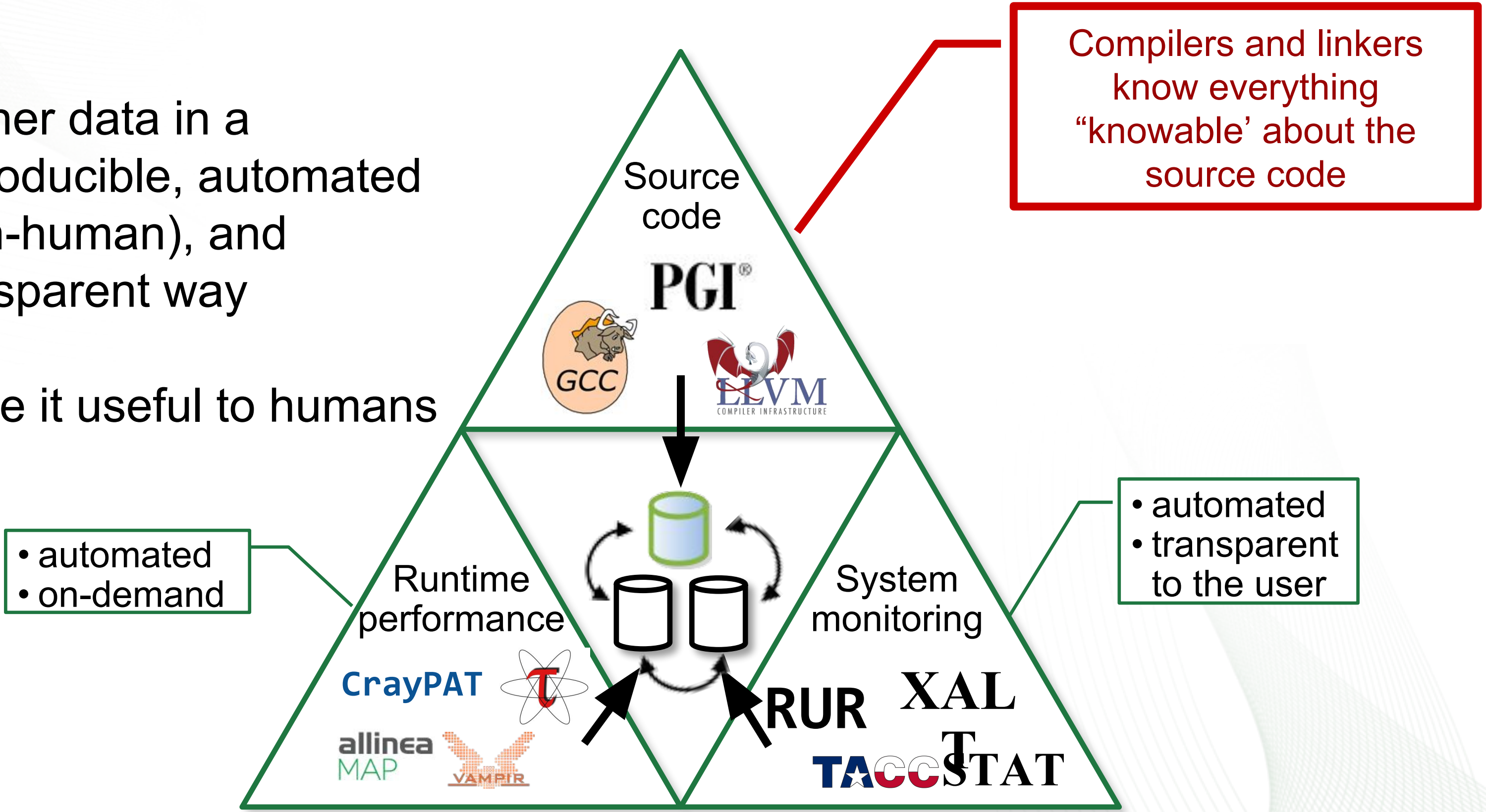
Where We Are Now



Where We Need To Be

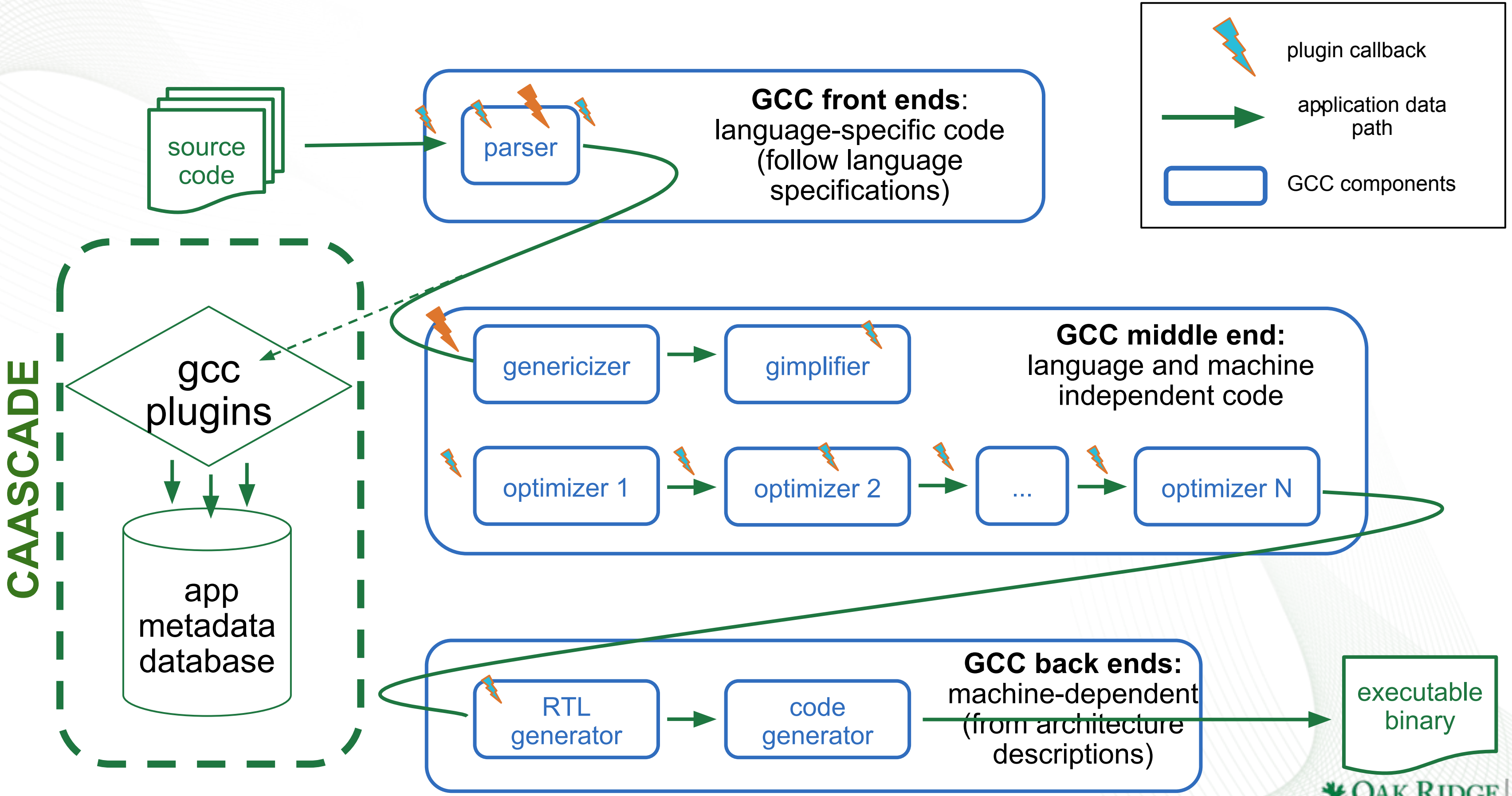
Goals:

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



CAASCADE: *C*ompiler-*A*ssisted *A*pplication *S*ource *C*ode *A*nalysis and *D*atabase

Compiler (GCC) Intermediate Representation



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)

Extracted Program Information – continued

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

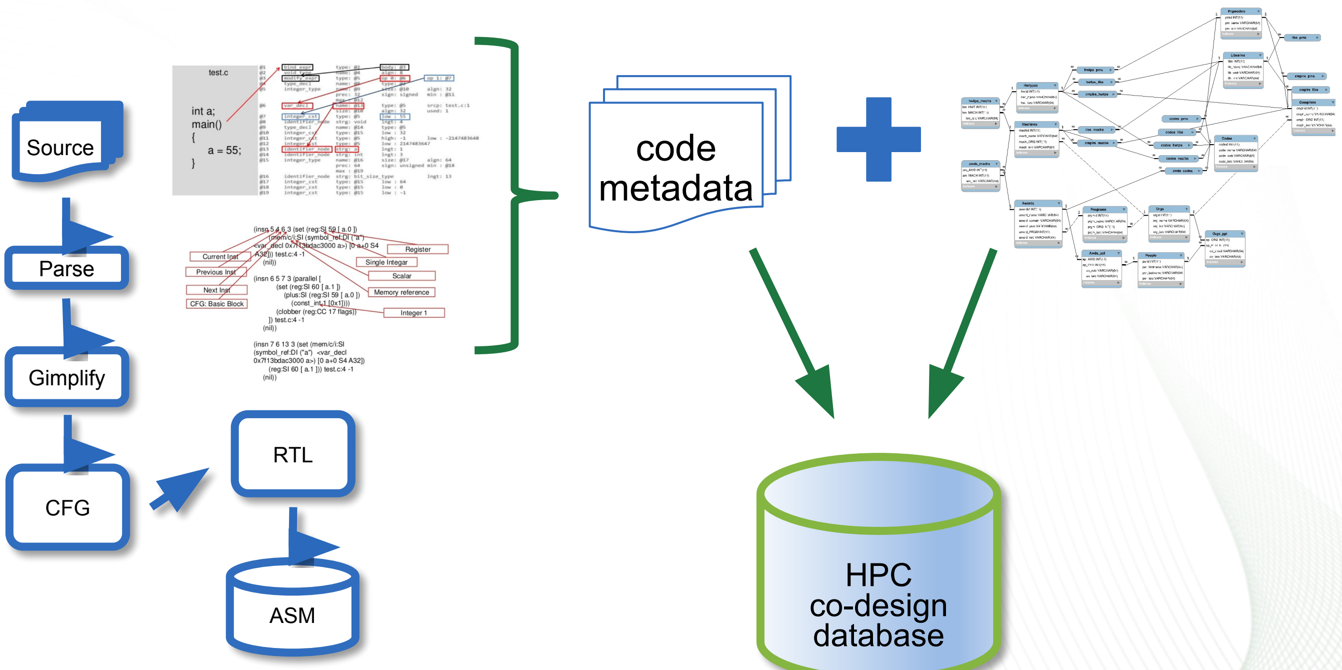
OpenMP directives (int)
statements inside OpenMP (int)
OpenMP threadprivate variables (int)
OpenMP UDR variables (int)
OpenMP declare target variables (int)
OpenACC directives (int)
statements inside OpenACC (int)
contains subroutines (bool)
OpenACC subroutine (bool)
OpenACC declare create variables (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)

+

Representation
(database)



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)

Executable	Link Program	Build User	Build Date	Host
RayleighTaylor_Cray_GNU	gfortran	reubendb	2018-04-09 22:28:45	titan
RayleighTaylor_Cray_GNU	gfortran	reubendb	2017-08-10 14:13:17	titan
RayleighTaylor_Cray_GNU	gfortran	reubendb	2017-08-10 14:11:42	titan

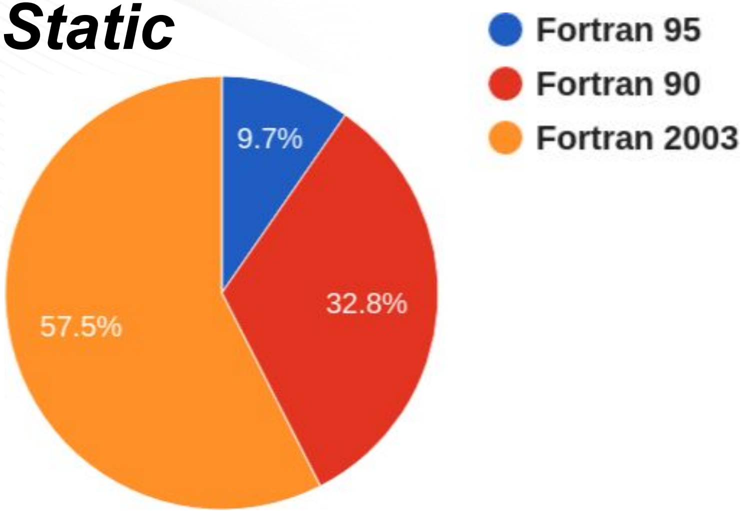
Object Name	Source Object	Source Name	Compile Command	Build User	Host
Basics.a	Runtime.o	Runtime.f90	[illegible]	reubendb	titan
Basics.a	PROGRAM_HEADER_Singleton.o	PROGRAM_HEADER_Singleton.f90	[illegible]	reubendb	titan
Basics.a	GetMemoryUsage_Command.o	GetMemoryUsage_Command.f90	[illegible]	reubendb	titan

Results: GenASiS

An astrophysics simulation framework
written in Fortran

Distribution of Fortran Language Standard

Static



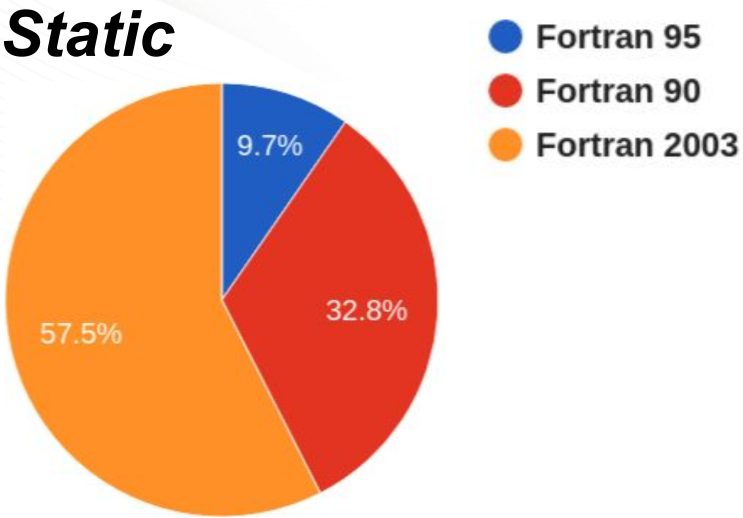
Program units (modules, subroutines) requiring the specified minimum language standard to compile

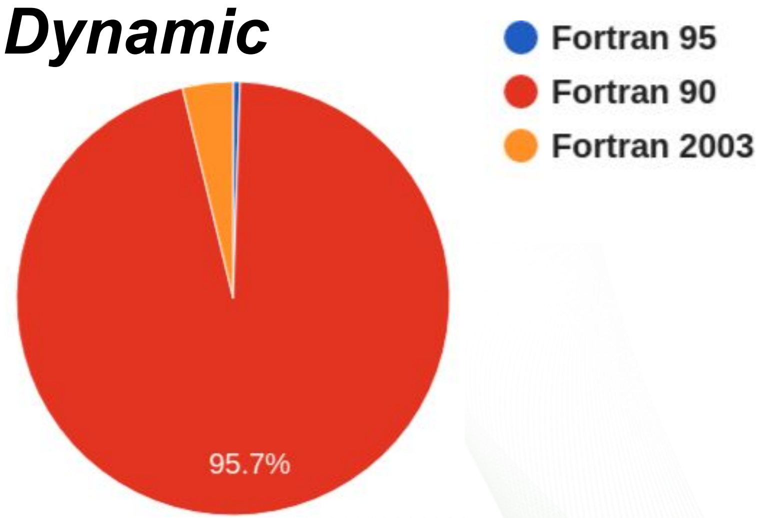
Dynamic

- 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

Distribution of Fortran Language Standard

what (Fortran) language standard gets used the most?



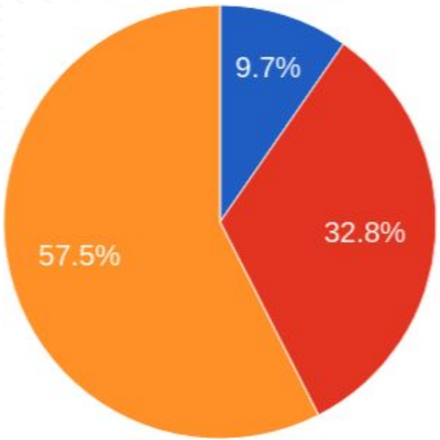


Program units (modules, subroutines) requiring the specified minimum language standard to compile

Distribution of Fortran Language Standard

what (Fortran) language standard gets used the most?

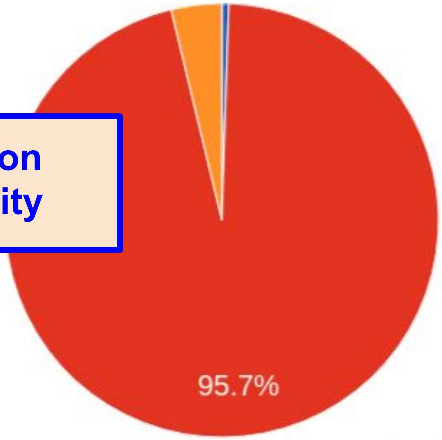
Static



Fortran 95
Fortran 90
Fortran 2003

Required for application developer's productivity

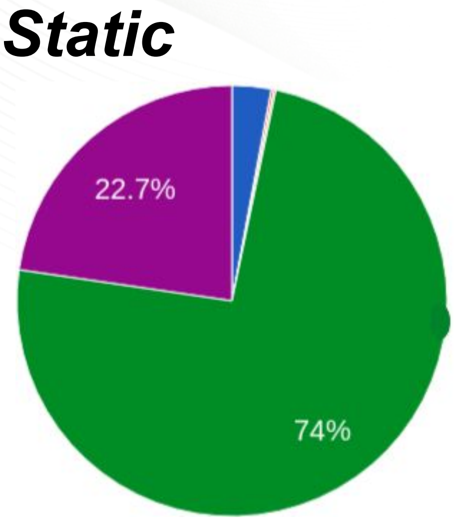
Dynamic

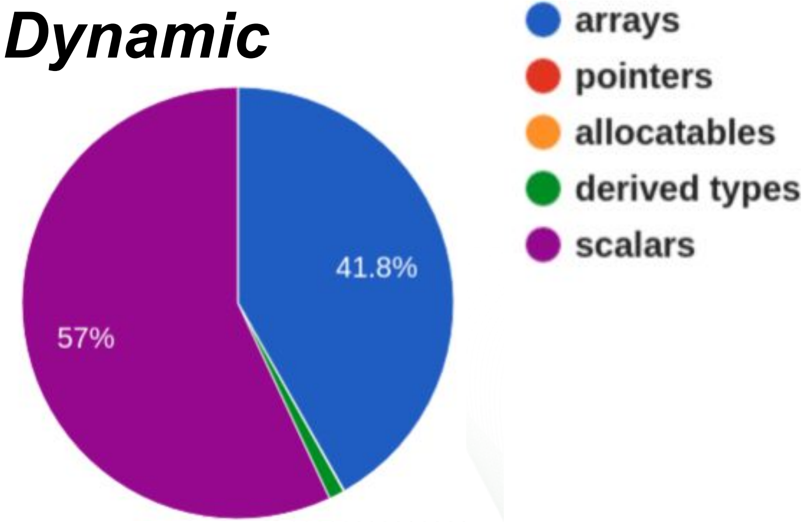


Fortran 95
Fortran 90
Fortran 2003

Program units (modules, subroutines) requiring the specified minimum language standard to compile

Distribution of Data Types

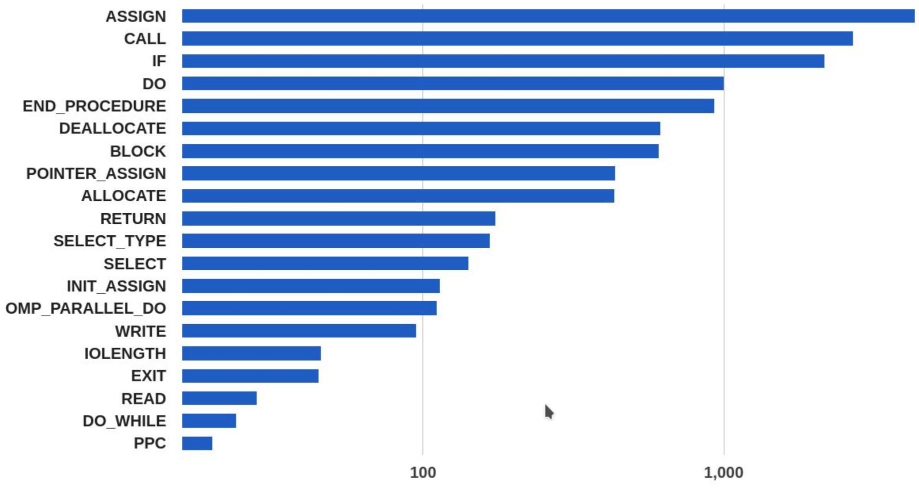




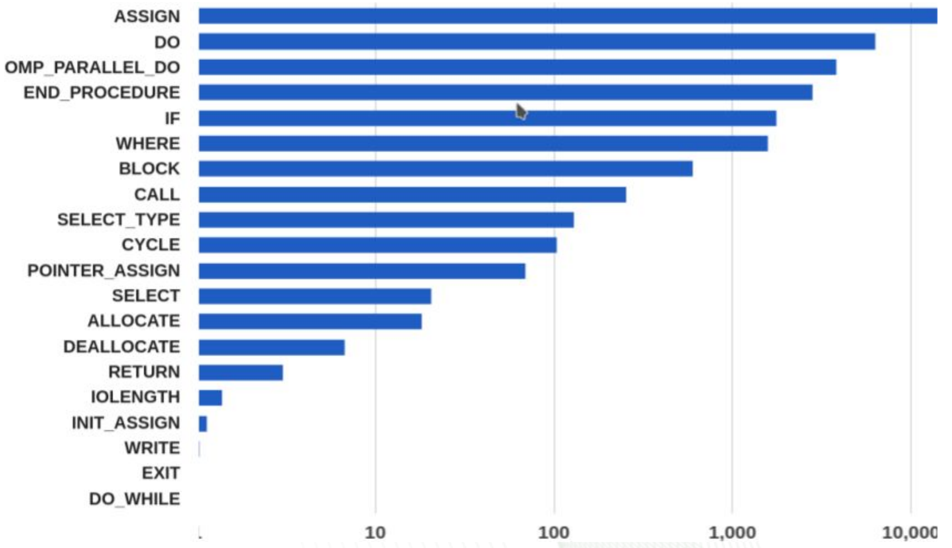
Derived types gets written the most into the code, yet *scalars* and *arrays* contribute the most during execution

Classification of Executable Statements

Static

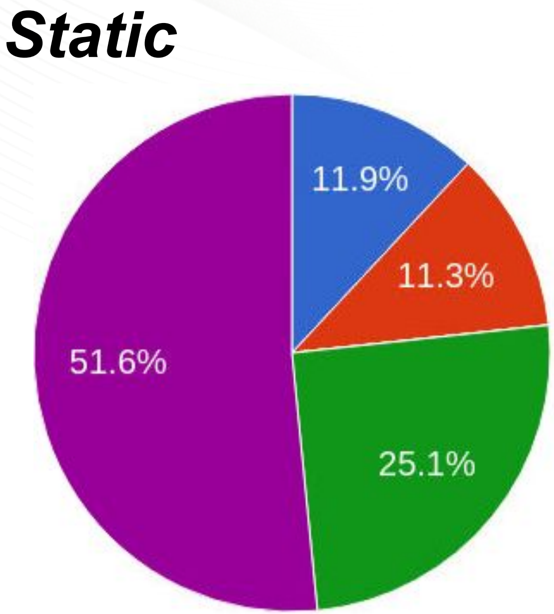


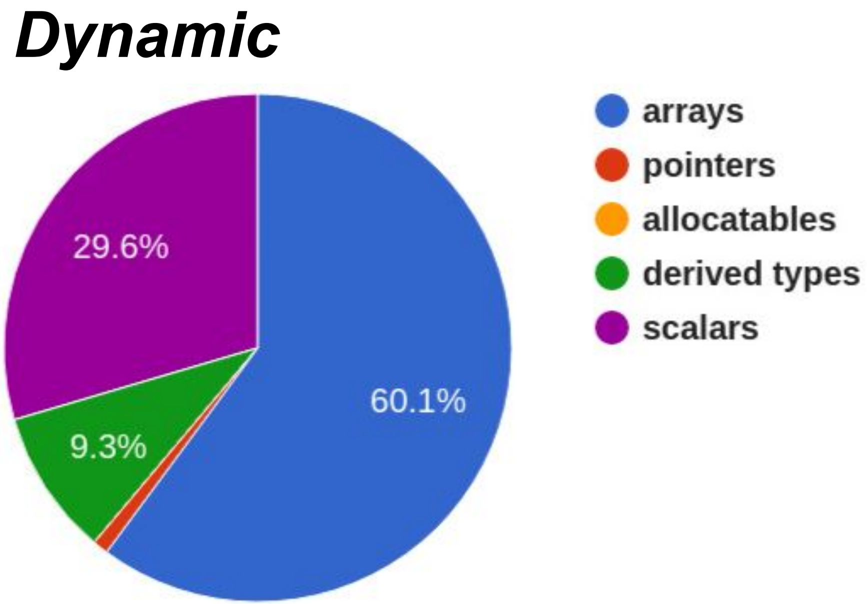
Dynamic



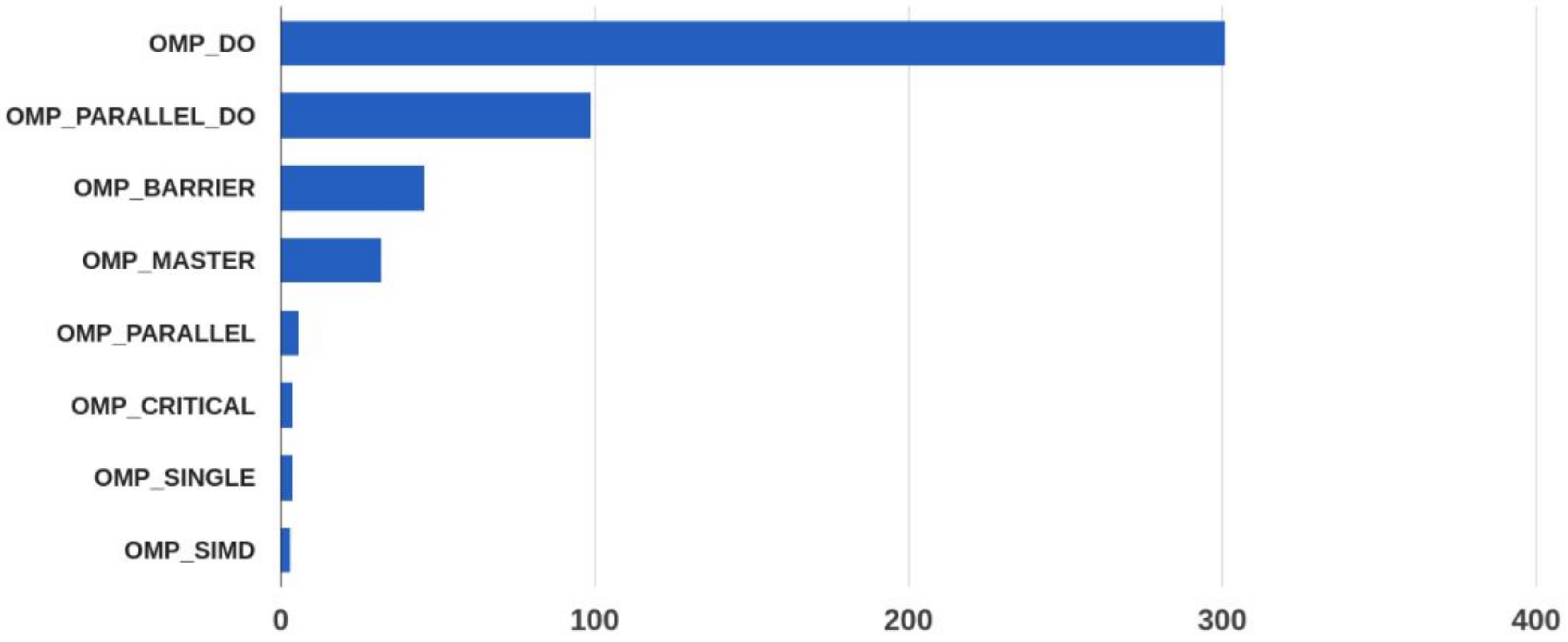
QMCPACK: Data Types Distribution

Many-body Quantum Monte Carlo code (C++)





E3SM: OpenMP Statements



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, …)