Contribution to automatic performance analysis of parallel applications

François Trahay

HDR defense
29th june 2021

Reviewers:
- Brice Goglin, INRIA Bordeaux Sud-Ouest
- Pascal Felber, Université de Neuchatel
- Lionel Seinturier, Université de Lille

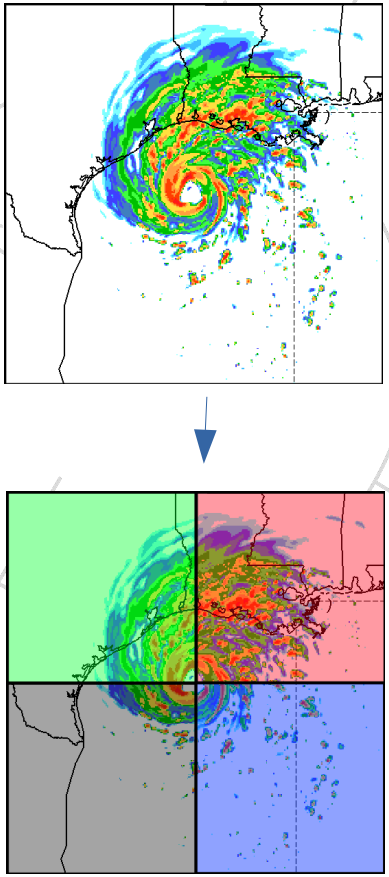
Examiners:
- Raymond Namyst, Université de Bordeaux
- Gaël Thomas, Télécom SudParis

ip-paris.fr

samovar

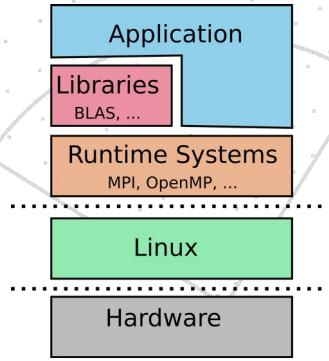
Parallel programming

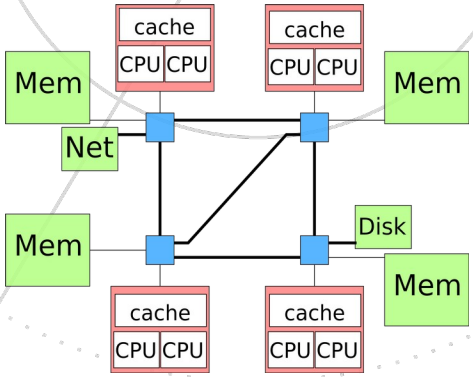
- *High Performance Computing*
  - Extensively used in weather forecasting, molecular modeling, physical simulation, ...
  - Need for a lot of computing power
- *Parallelizing an application*
  - Split a problem into sub problems
  - Distribute over several processors
  - Processors communicate their contribution through a network
- *Performance improvement*
  - 4 processors → 4 times faster
  - 128 processors → 128 times faster ?



Improving parallel applications

- *Many sources of parallel inefficiency*
  - Algorithmic issues
    - Number of synchronization increases at scale
  - Bad usage of hardware ressources
    - Memory access, Disk, ...
- *Improving performance is hard*
  - Need for tools to help developpers

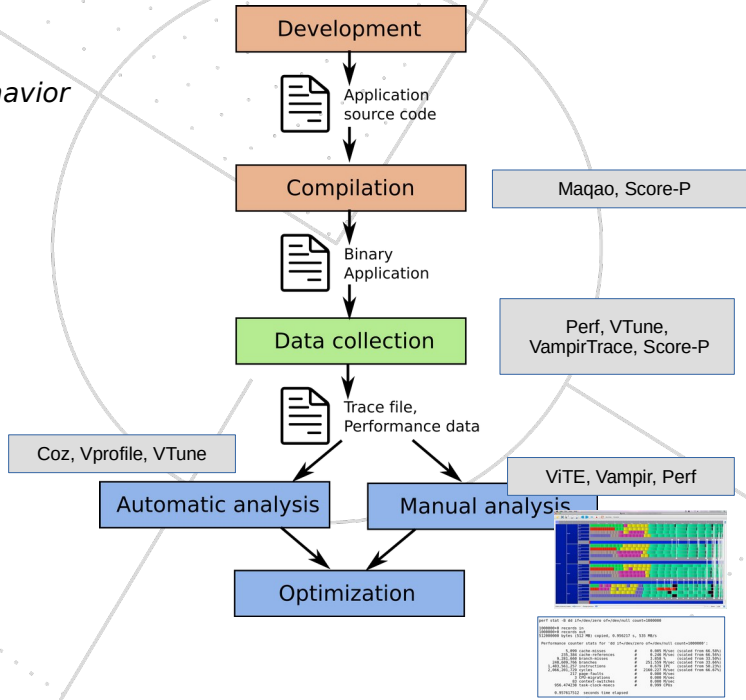




Available resources on a computer

Performance analysis process

- *Collecting performance data*
  - *Challenge: collecting enough data without altering the application behavior*
- *Manual analysis of performance data*
  - *Challenge: depicting performance data*
- *Automatic analysis of performance data*
  - *Challenge: automating the detection of problems*



Scientific approach

- *Experimental science*
  - *Analyze real parallel applications*
  - *Execute applications on real computers*
- *Additional constraints*
  - *Release developped software as open-source*
  - *Avoid application modification when possible*

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

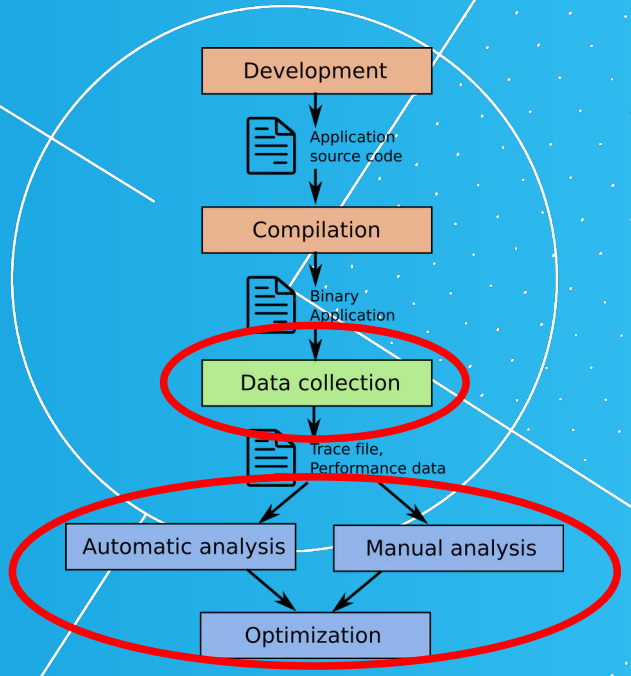
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

C) Detecting scalability issues with ScalOMP



Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

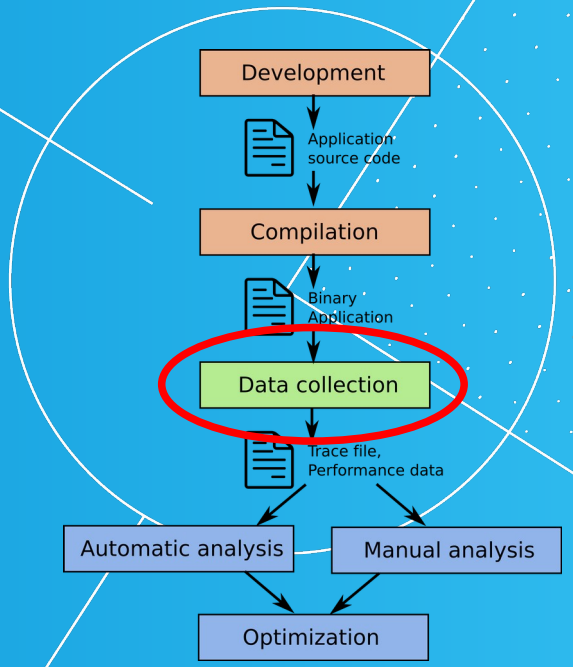
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

C) Detecting scalability issues with ScalOMP



Collecting execution traces with EZTrace

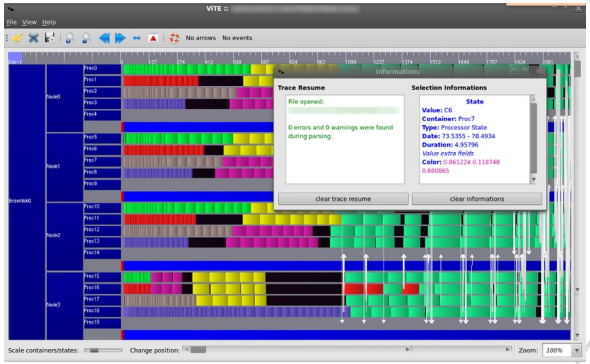
- *Tracing tool for parallel applications*
  - Automatically instrument applications

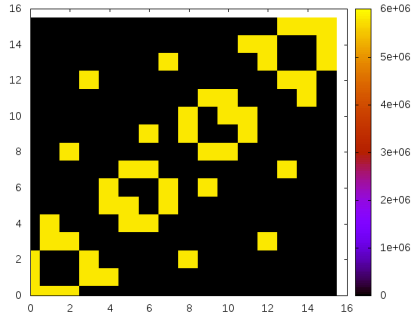
```
$ ./application foo bar
```

→

```
$ eztrace ./application foo bar
```

  - Generate execution trace for post-mortem analysis





https://eztrace.gitlab.io/eztrace

[CCGrid'11]

Plugin-based tracing tool

- *EZTrace uses plugins*
  - Pre-defined plugins (MPI, OpenMP, CUDA, …)
  - User-defined plugins
    - Written in C by hand
    - Generated using a Domain Specific Language

```
BEGIN_MODULE
NAME user_plugin
DESC "module for the example library"
LANGUAGE C

void bar(double* array, int size)
BEGIN
  RECORD_STATE("bar")
END

int foo(int a, double b)
BEGIN
 ADD_VAR("nb_tasks", a)
 CALL_FUNC
 EVENT("task submitted called")
END

END_MODULE
```

```
int foo(int a, double b) {
 do_something() ;
}
```

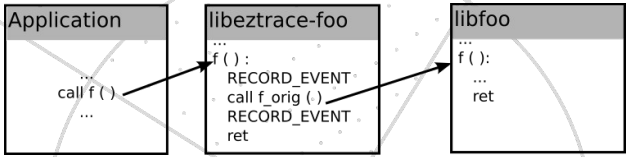
```
int foo(int a, double b) {
 record_event(''foo_entry'', a);
 do_something();
 record_event(''foo_exit'');
}
```

+

```
int process_foo_entry(event *e) {
   ...
}
int process_foo_exit(event *e) {
   ...
}
```

Binary instrumentation of applications

- *LD_PRELOAD interception*
  - Define a wrapper function that intercepts calls
  - Works for shared libraries only

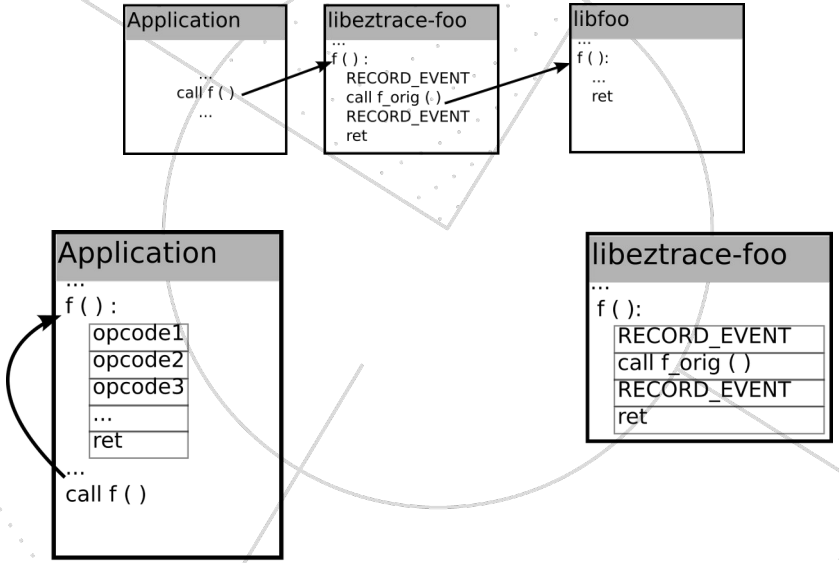


Binary instrumentation of applications

LD_PRELOAD interception

- Define a wrapper function that intercepts calls
- Works for shared libraries only

- *Binary instrumentation*

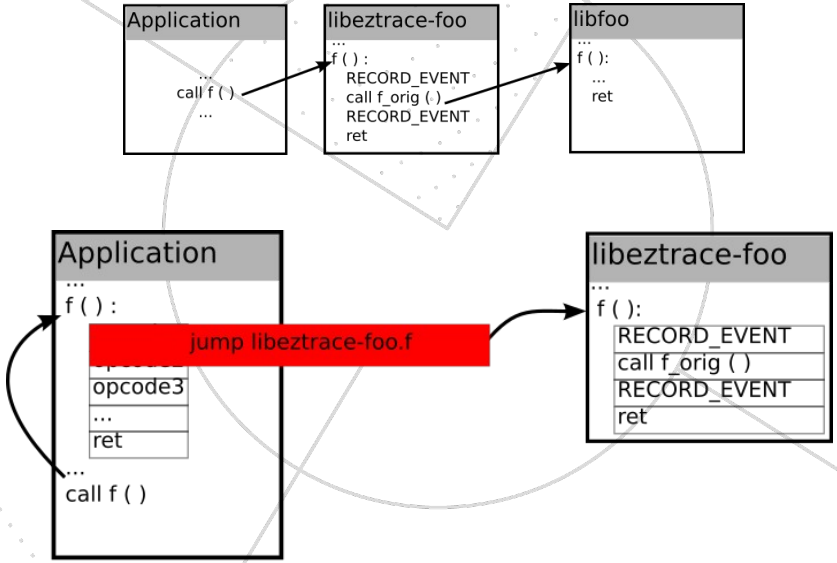


Binary instrumentation of applications

LD_PRELOAD interception

- Define a wrapper function that intercepts calls
- Works for shared libraries only

- *Binary instrumentation*
  - Insert a jump instruction at the function entry
  - Low overhead interposition
  - CPU-specific lines of code < 100

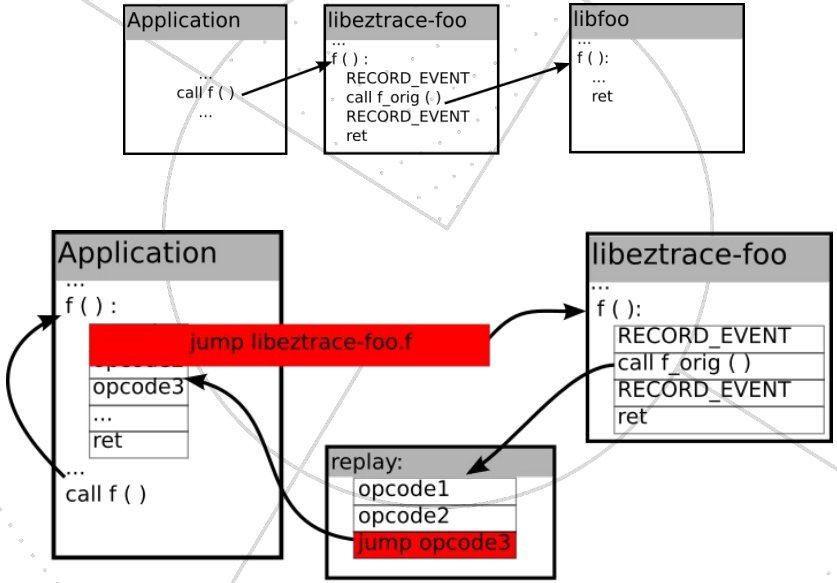


Binary instrumentation of applications

LD_PRELOAD interception

- Define a wrapper function that intercepts calls
- Works for shared libraries only

- *Binary instrumentation*
  - Insert a jump instruction at the function entry
  - Low overhead interposition
  - CPU-specific lines of code < 100



Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

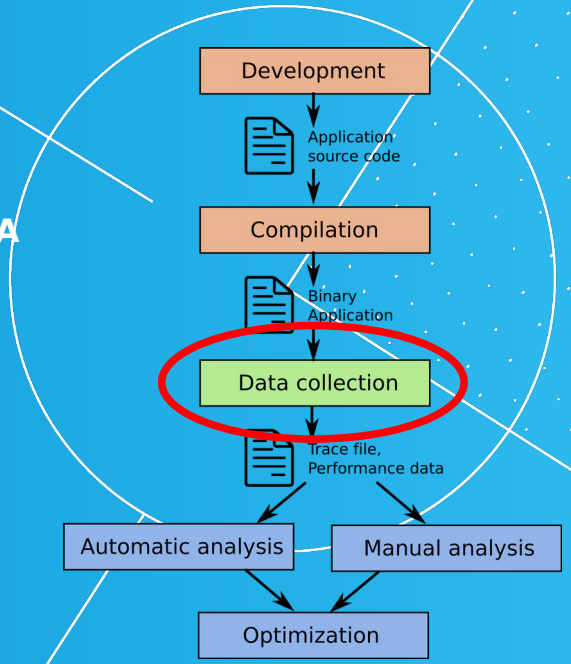
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

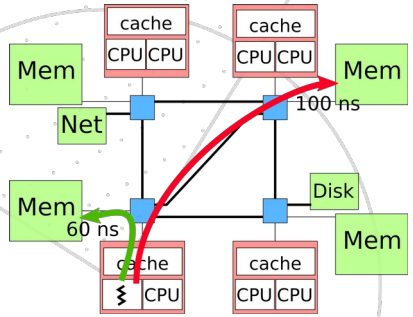
C) Detecting scalability issues with ScalOMP



[ICPP'18]

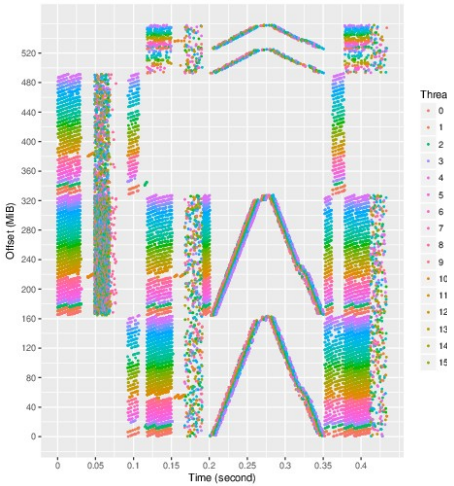
NumaMMA memory profiler

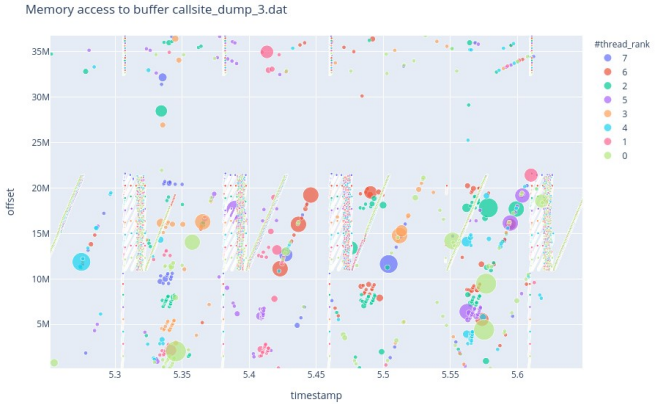
- *Goal: running threads close to their memory*
  - Access latency depends on the placement



- *NumaMMA memory profiler*
  - Low overhead collection of memory access through hardware sampling
  - Detect the access pattern of threads on memory objects
  - Reports the most accessed objects

- *Allows to improve performance*
  - Select the best NUMA allocation policy
  - Improve the thread binding
  - Up to 28% performance improvement





https://numamma.github.io/numamma/

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

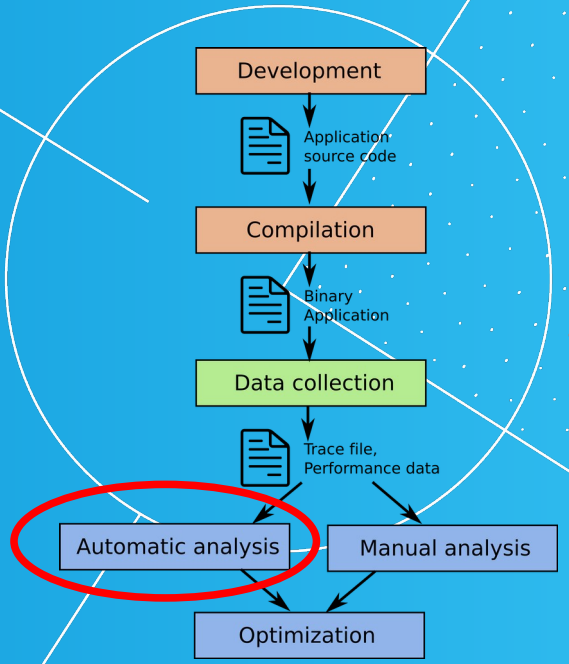
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

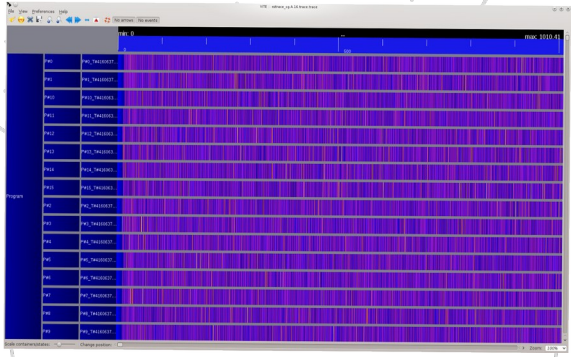
B) Differential execution analysis

C) Detecting scalability issues with ScalOMP



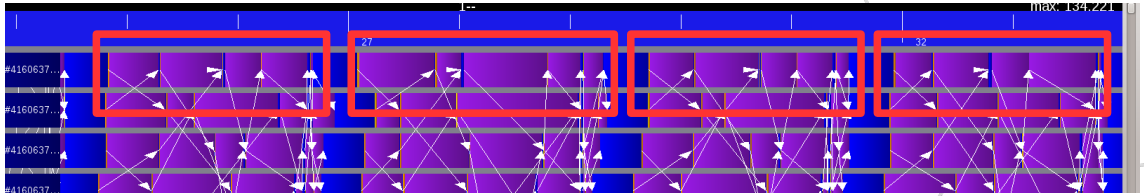
A case for automatic performance analysis

- *Execution traces contain lots of information*



NPB CG class A 16 MPI Processes – 426 000 events

- *Most traces have repetitive patterns of events*

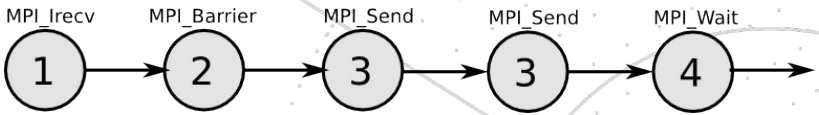


→ Need for tools that
- process large quantities of data
- detect performance problems automatically
- provide optimization hints

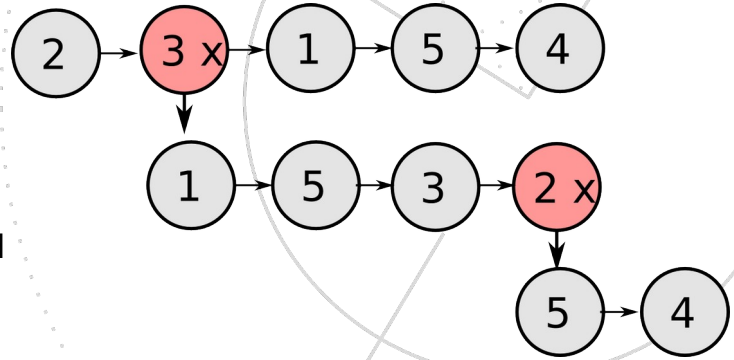
[PDP'15]

Detecting the structure of traces

- *Trace = sequence of events*



- *A trace factorization algorithm*
  - Find sequences that repeat
  - Aggregate sequences into loops
  - Expand sequences
  - Evaluation: ~ 1M events per second processed



- *Filtering traces*
  - Remove similar sequences with similar duration
  - Evaluation: up to 99 % of events can be filtered out

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

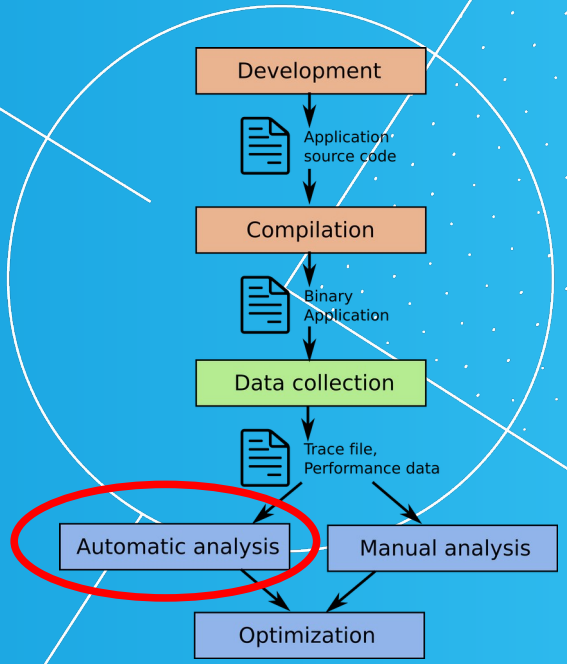
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

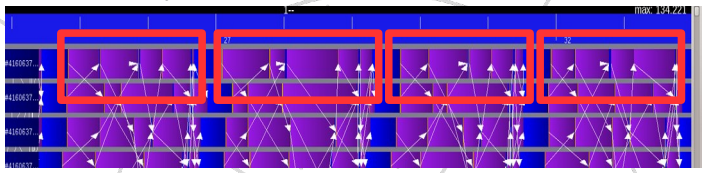
C) Detecting scalability issues with ScalOMP

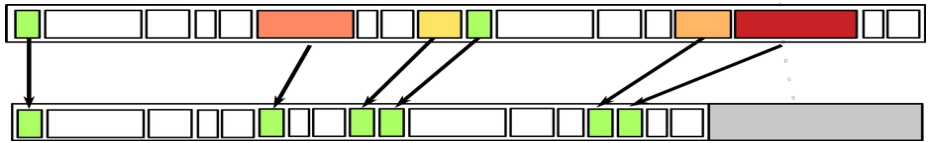


[TPDS'19]

Detecting thread contention

- Differential execution analysis
  - Detect sequences of event that repeat
  - Compare the duration of sequences
  - Simulate the execution time without contention





→ Universal indicator for contention

- Evaluation on 27 applications
  - Detect 12 problems (lock contention, disk IO, network, memory placement, …)
  - Up to 900% speedup once (manually) fixed

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

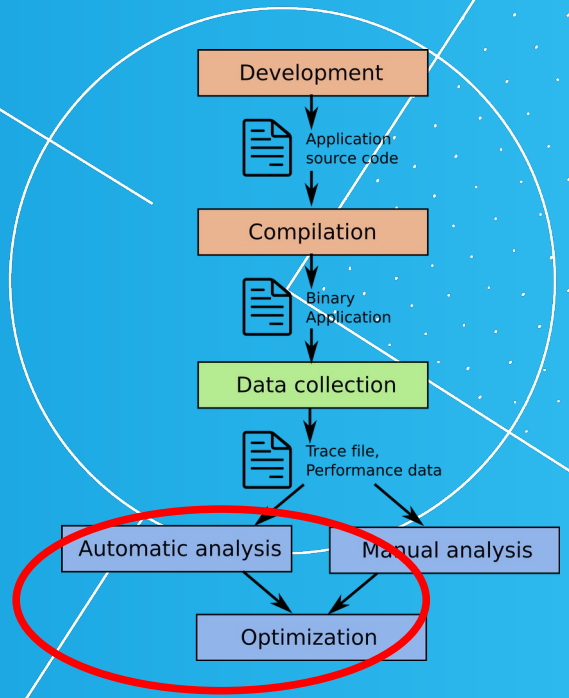
B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

C) Detecting scalability issues with ScalOMP



[IWOMP'19]

Detecting scalability issues with ScalOMP

- OpenMP: pragma-based parallelisation scheme
  - Which parallel region degrades performance ?
  - Why does it degrade performance ?
- Collecting performance data
  - Instrument OpenMP applications using OMPT
  - Adaptive sampling to mitigate the overhead of ScalOMP
- Detecting scalability issues
  - Scalability analysis: how do regions scale with the number of threads ?
  - Detect sources of performance problems (imbalance, synchronization)
  - Provide optimization hints to guide developpers

```
#pragma omp parallel for
    for (int i = 0; i < size; i++) {
        a[i] = b[i] * c[i];
    }
```

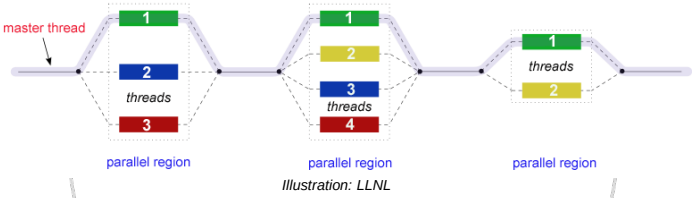


Illustration: LLNL

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

B) Differential execution analysis

C) Detecting scalability issues with ScalOMP

- ***Impact & dissemination***

Impact & dissemination

- Scientific production:
  - Research papers: [CCGrid'11], [PSTI'11], [PROPER'12], [CC'13], [PDP'15], [ICPP'18], [TPDS'19], [IWOMP'19], [CHEOPS'21], [OSR'21]
  - Open-source software: EZTrace, LiTL, NumaMMA

- Supervision
  - 3 PhD students
    - Mohamed Saïd Mosli Bouksiaa (2014-2018)
    - Anton Daumen (2018-2021)
    - Alexis Colin (2019-2022)
  - 1 post-doc researcher (Roman Iakymchuk, 2013)
  - 20+ master students

- Research projects
  - Carnot EZTrace (2012)
  - INRIA ADT EZPerf (2013)
  - DIM RFSI NumaMMA (2018)
  - ANR JCJC Pythia (2019)
  - FUI IDIOM (2019)

Other contributions

- Optimization of the I/O stack
  - [ICPP'21], [TCAD'20], [Systems'18], [Access'18], [TAAS'18], [TCC'17], [TACO'16], [TPDS'15]
- Runtime systems for
  - MPI [HPCC'20], [IJPP'16], [ICPP'16], [EuroMPI'12], [Cluster'11]
  - Transactional memory [Coord'18]
  - Heterogeneous computing [ICPP'15]

Contributions

I. *Collecting performance data*

A) Collecting execution traces with EZTrace

B) Collecting memory access patterns with NumaMMA

II. *Analyzing performance data*

A) Detecting the structure of an execution trace

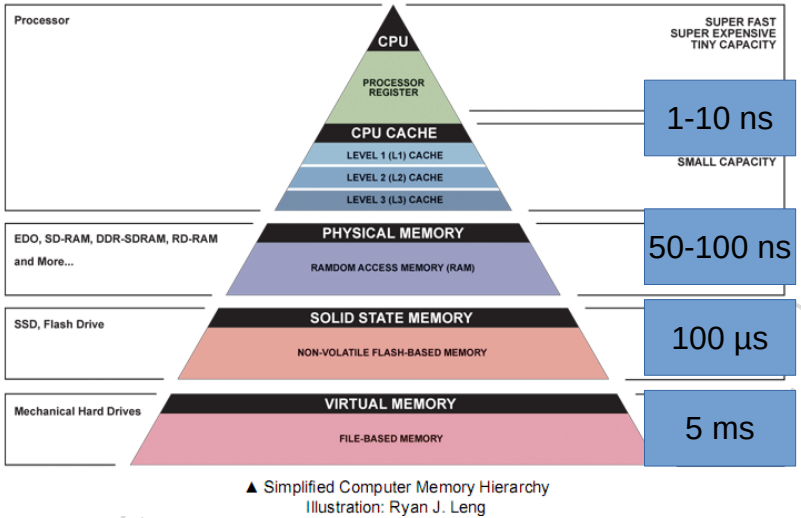
B) Differential execution analysis

C) Detecting scalability issues with ScalOMP

- ***Future work***

Data management systems

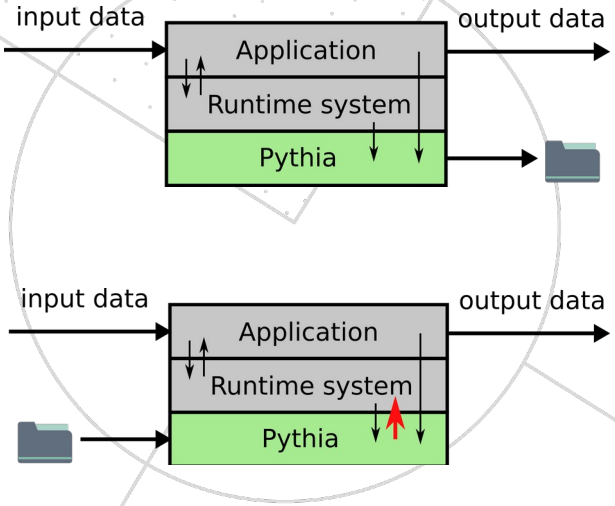
- Convergence of storage and memory
- Challenges
  - IO overhead is now due to software
  - Memory subsystem becomes heterogeneous
- Ongoing/future projects
  - Performance analysis of the whole I/O stack [FUI IDIOM]
  - Memory placement strategies for heterogeneous memory
  - Modeling application memory performance



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

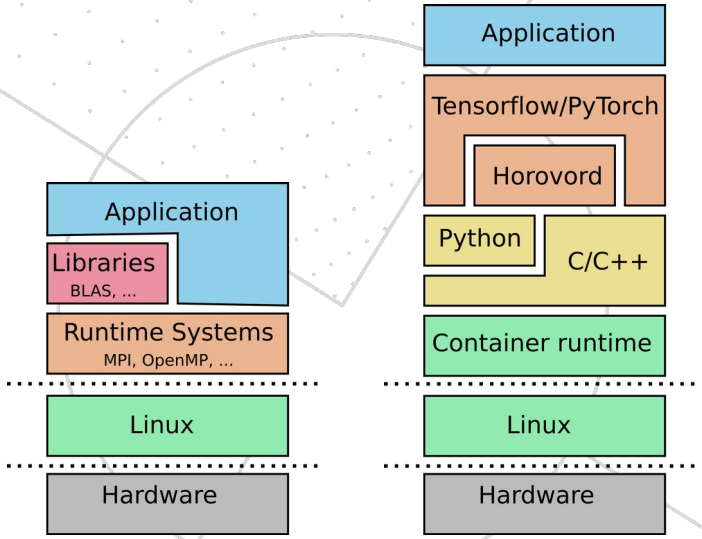
Automatic optimization of applications

- Many applications are regular
  - Same execution regardless of the input data
- Record one run, and guide the following ones
  - Trace-based oracle for runtime systems [ANR JCJC Pythia]
  - Trace-based compiler optimizations



Convergence of HPC and AI

- Supercomputers now run HPDA applications
  - New software stack
    - Need for performance analysis tools
  - New type of workload
    - Need to adapt runtime systems



Questions ?

Acknowledgements

- The G2 / HP2 / PDS group
- PhD students
- Colleagues from INF
- Colleagues from other departments & support services
- Collaborators throughout the world