Efficient methods for kernel trace analysis parallelization

Fabien Reumont-Locke
Under the supervision of Prof. Michel Dagenais

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Presentation outline

I. Introduction and research objectives
II. Parallel Solution
    A. Adapting the tools to parallel processing
    B. Data partitioning
    C. Resolving data dependencies
III. Experimental Results
    A. Parallel memory and I/O operations
    B. Performance and scaling
IV. The road ahead and conclusion

Parallel computing

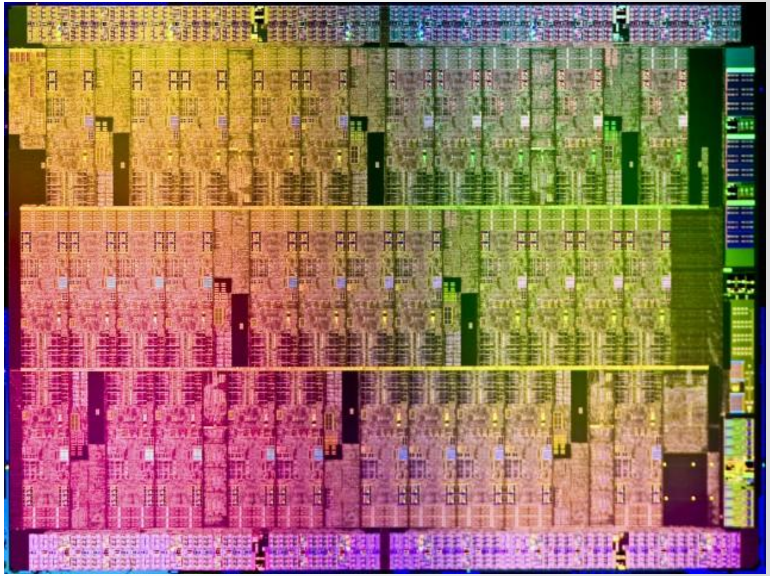
More and more cores being traced

More and more trace data being generated

Trace analysis is still single-threaded

The gap between the amount of traced data and the analysis speed is ever-widening

Intel Xeon Phi - 64 cores



Source: http://www.extremetech.com/wp-content/uploads/2012/04/Aubrey_Isle_die-640x480.jpg

POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Research objectives

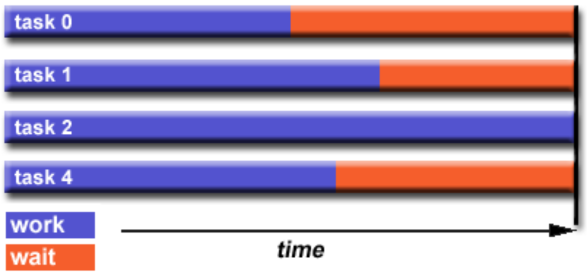
Do parallel computing methods allow for a scalable acceleration of kernel trace analysis?

The goal is to develop trace analysis parallelization methods that will:

a. Work for most existing analyses
b. Be efficient (provide considerable speedup over sequential methods)
c. Be scalable (improved performance as number of parallel units increases)

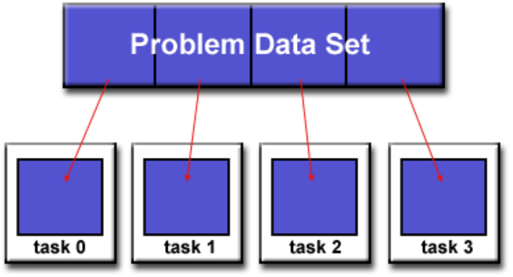
Challenges of parallelization

Load balancing



Source: https://computing.llnl.gov/tutorials/parallel_comp/

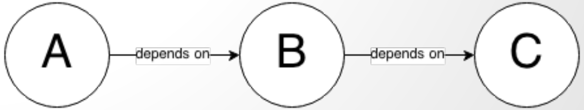
Data partitioning



Source: https://computing.llnl.gov/tutorials/parallel_comp/

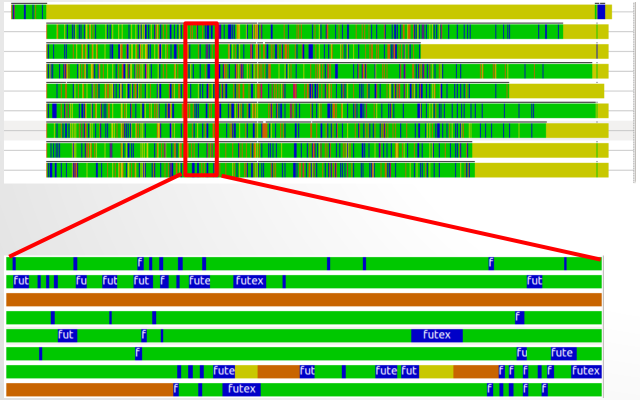
Locking and synchronisation

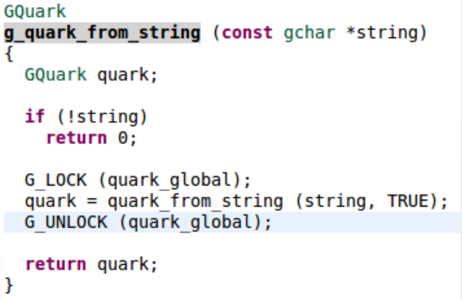
Data dependencies



Adapting babeltrace to parallel analysis

- Added support for multiple iterators per trace by cloning file streams inside each iterator
- Added thread-local quark cache to prevent contention on hash-table access





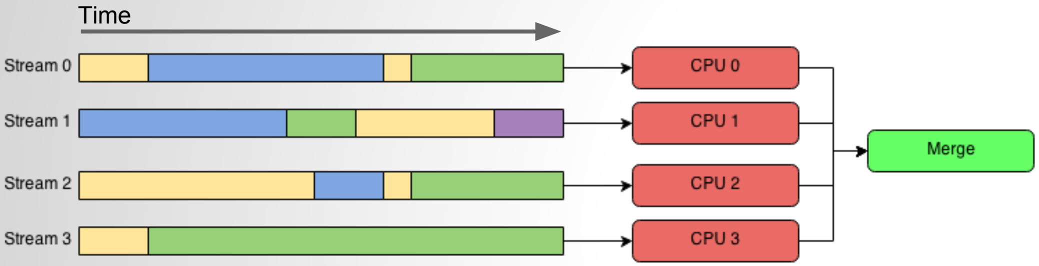
```
GQuark
g_quark_from_string (const gchar *string)
{
  GQuark quark;

  if (!string)
    return 0;

  G_LOCK (quark_global);
  quark = quark_from_string (string, TRUE);
  G_UNLOCK (quark_global);

  return quark;
}
```

POLYTECHNIQUE
MONTRÉAL

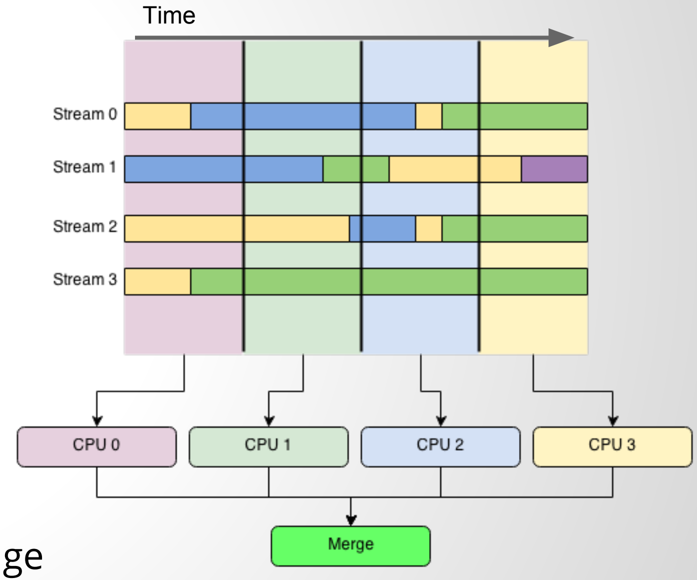
LE GÉNIE
EN PREMIÈRE CLASSE

Data partitioning

Per-stream?



Per-time range?

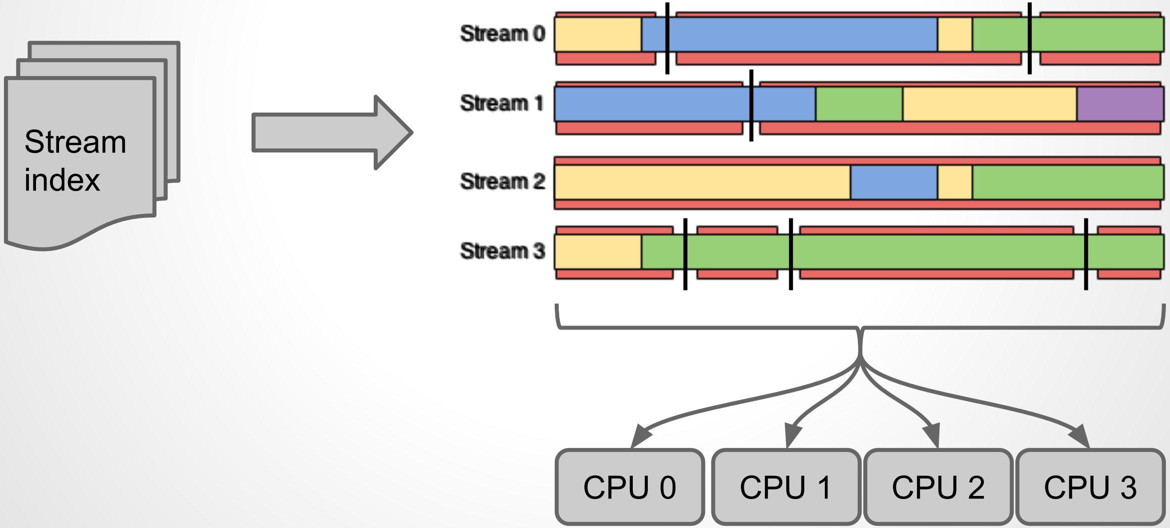


Both suffer from balancing problems!

- **Fewer streams** than available processors
- Some streams contain **more data** than the others
- Trace data **unevenly distributed** within the time range

Hybrid packet-driven partitioning

- CTF traces have a packet index that we can use to balance the load
- We assume that packet size is proportional to the number of events
- Walk packet index, accumulating packets until a certain threshold



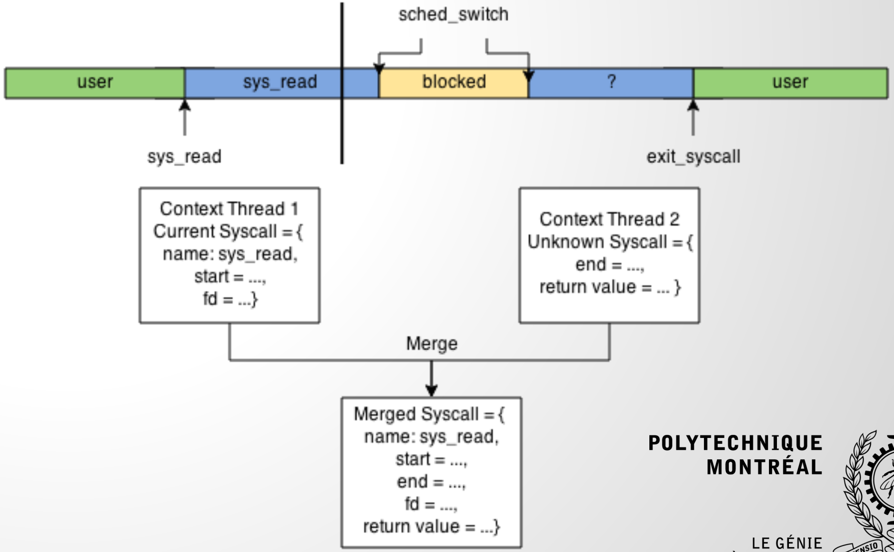
POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Breaking data dependencies

- Most analyses keep a running “current state” containing all the necessary data
- This current state is also queried to know, for example, which system call was running

- But what if we don’t know some of the current state?
- We rely on the fact that the unknown state lasts only until the next event is read
    - sys_* -> syscall
    - exit_syscall -> user



Context Thread 1
Current Syscall = {
name: sys_read,
start = ...,
fd = ...}

Context Thread 2
Unknown Syscall = {
end = ...,
return value = ... }

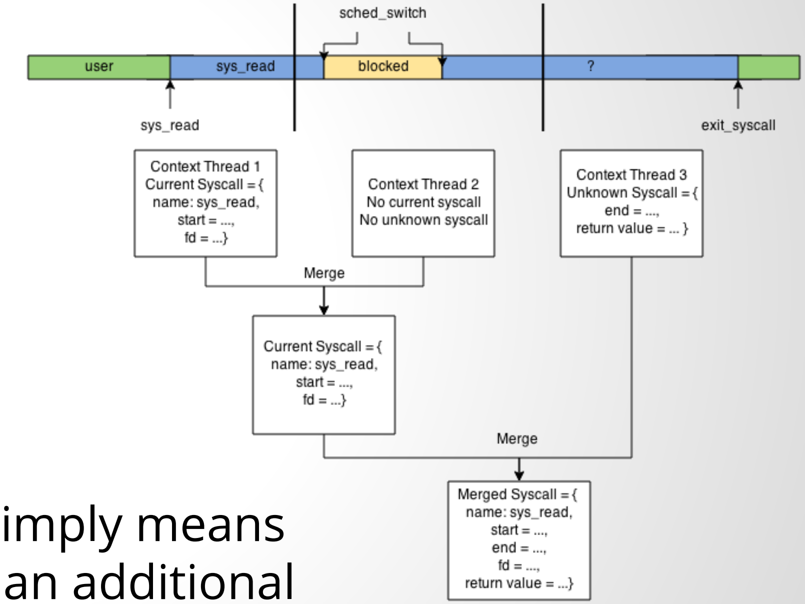
Merged Syscall = {
name: sys_read,
start = ...,
end = ...,
fd = ...,
return value = ...}

POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

State propagation

- Values dependent on unknown state are kept in each chunk’s context
    - e.g. unknown syscall, or syscall in unknown current thread
- State is propagated forward in time at the merge phase
- In terms of implementation, this simply means handling unknown state + adding an additional *merge* method to allow merging the contexts



Context Thread 1
Current Syscall = {
name: sys_read,
start = ...,
fd = ...}

Context Thread 2
No current syscall
No unknown syscall

Context Thread 3
Unknown Syscall = {
end = ...,
return value = ... }

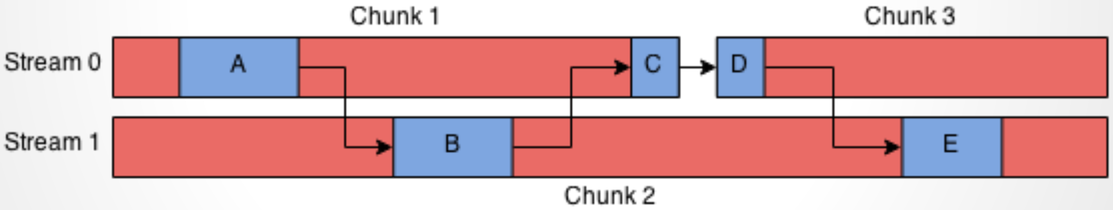
Current Syscall = {
name: sys_read,
start = ...,
fd = ...}

Merged Syscall = {
name: sys_read,
start = ...,
end = ...,
fd = ...,
return value = ...}

Notes on hybrid balancing

- Hybrid balancing adds something else to worry about: ***migrations***
- This is solved by keeping track of process migrations and merging in the same way as described before

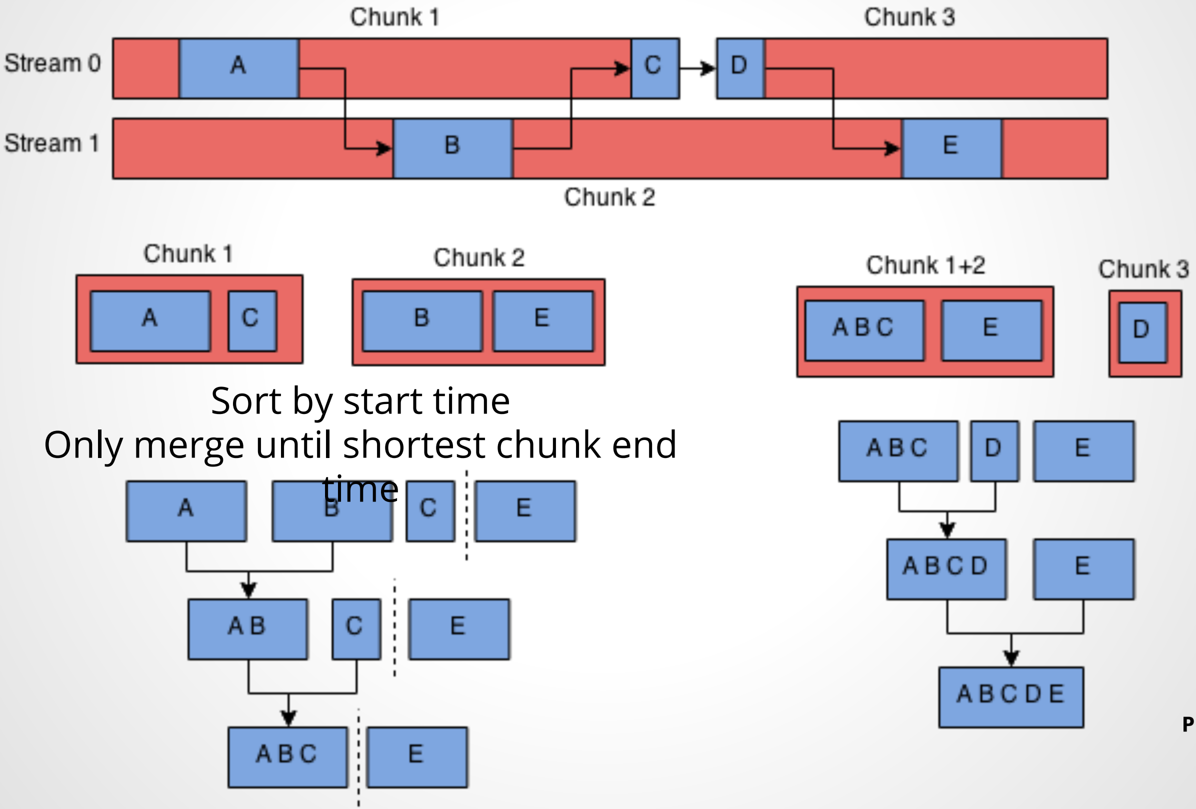
For example:



In blue are continuous executions of a process on a processor

Arrows represent migrations and dependencies

Merging algorithm



Sort by start time
Only merge until shortest chunk end time

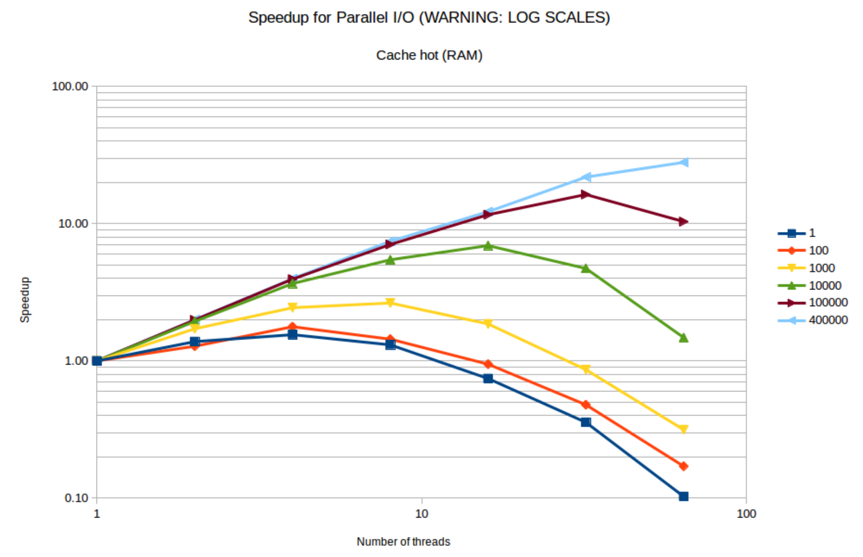
Trace analysis: I/O bound?

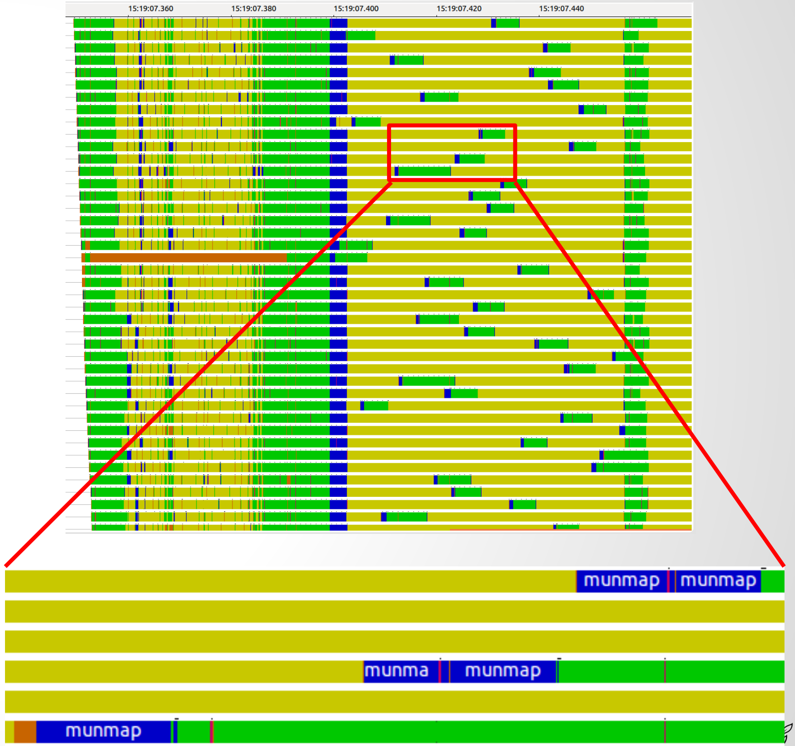
- If trace decoding (i.e. babeltrace) was to be made faster, would trace analysis become I/O bound?
- Simulate execution using simple program with tweakable params
    - Amount of CPU work (“iterations”)
    - Size of mmap’d chunks
    - Prefaulting, etc.
- Allows to simulate with various:
    - Hardware
    - CPU efficiency of analysis
    - I/O efficiency of analysis

```
threadRoutine(chunk_size, chunk_offset, file) {
 buffer = mmap(chunk_size, chunk_offset, file);
 for (i = 0; i < chunk_size; i += PAGE_SIZE) {
   sum += buffer[i];
   /* do some useless work */
   for (j = 0; j < ITERATIONS; j++) {
     sum++;
   }
 }
 munmap(buffer);
 return sum;
}
```

Test CPU : 4 x AMD Opteron 6272
Sixteen-Core Processor

Concurrent memory operations





`mm->mmap_sem` serializes memory operations (`mmap`, `munmap`, page faults)

Solution: single thread does `mmap`/`munmap` in a pipeline

MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Test hardware - I/O

SATA Hard Disk Drive

- ~135 MBps sequential read

SATA Solid State Drive

- ~250 MBps sequential read

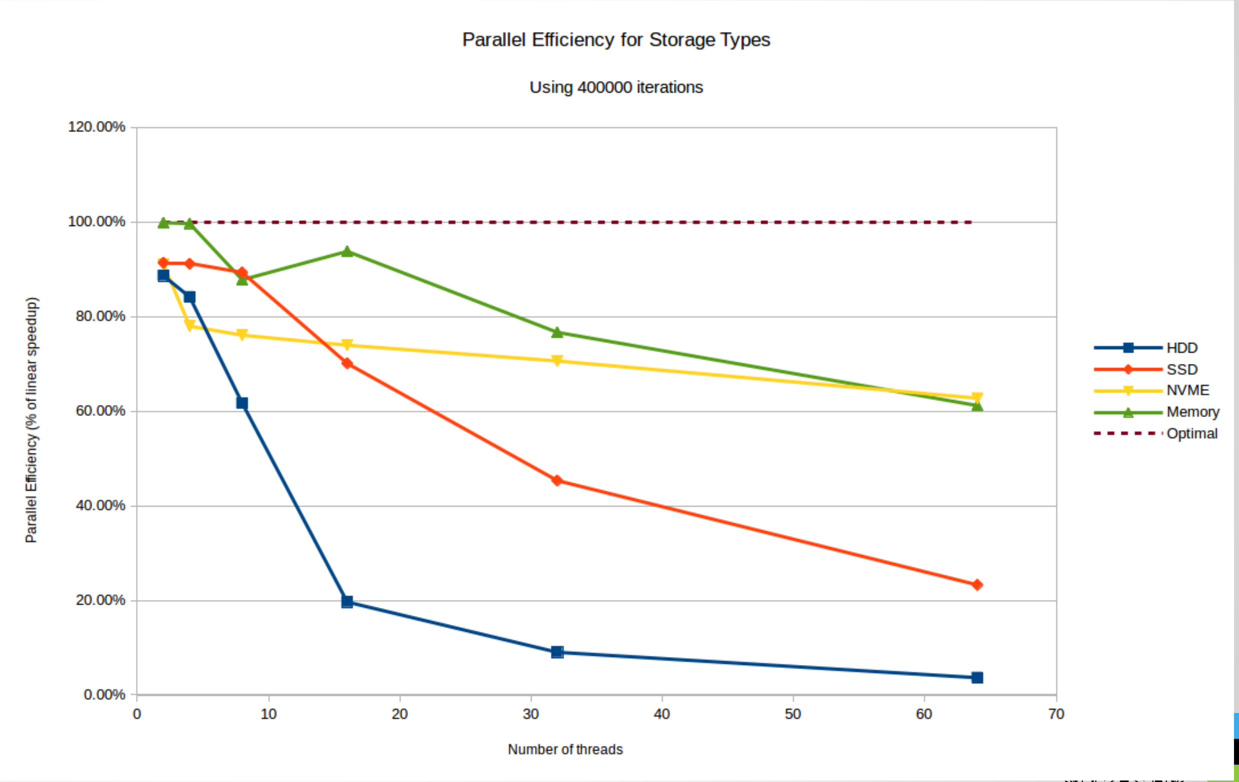
Intel P3700 PCIe SSD

- ~1145 MBps sequential read
- (yes, those are megabytes)

Parallel Efficiency

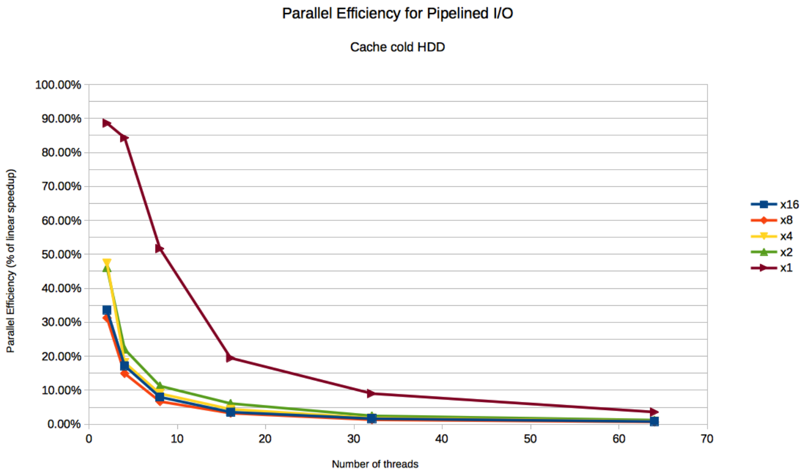
For a program with throughput similar to babeltrace (no analysis):

- 60% linear speedup with 8 threads on HDD (x5 speedup)
- 70% linear speedup with 16 threads on SSD (x11 speedup)
- 63% linear speedup with 64 threads on PCIe SSD (x40 speedup)

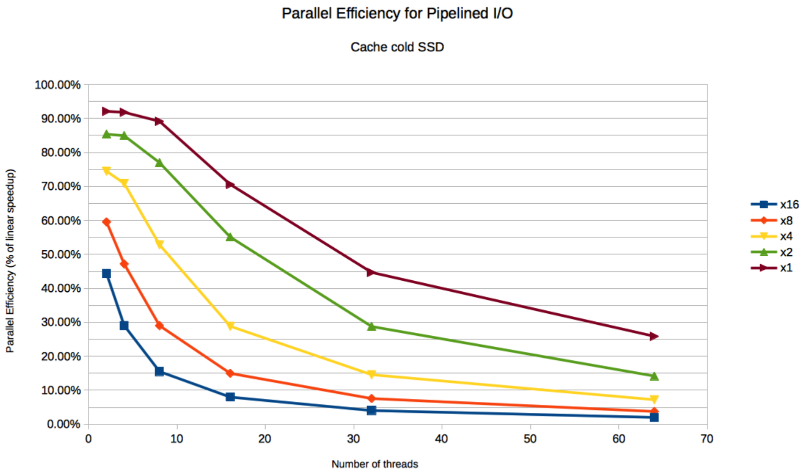

Parallel Efficiency for Storage Types

Using 400000 iterations

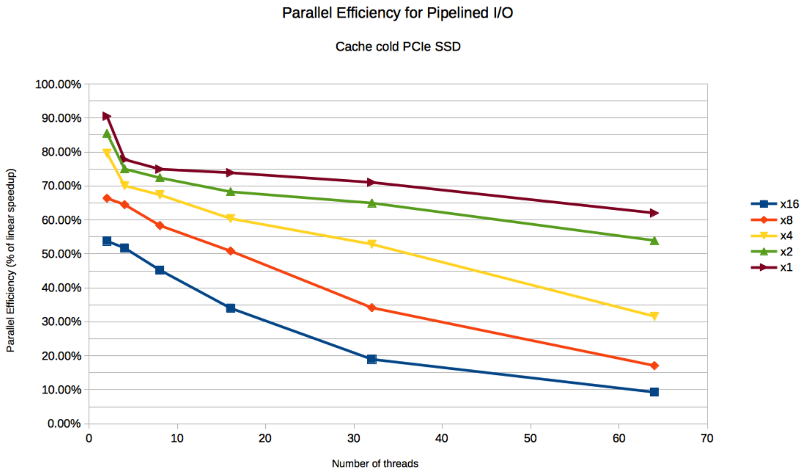
LE GÉNIE
EN PREMIÈRE CLASSE


Parallel Efficiency for Pipelined I/O

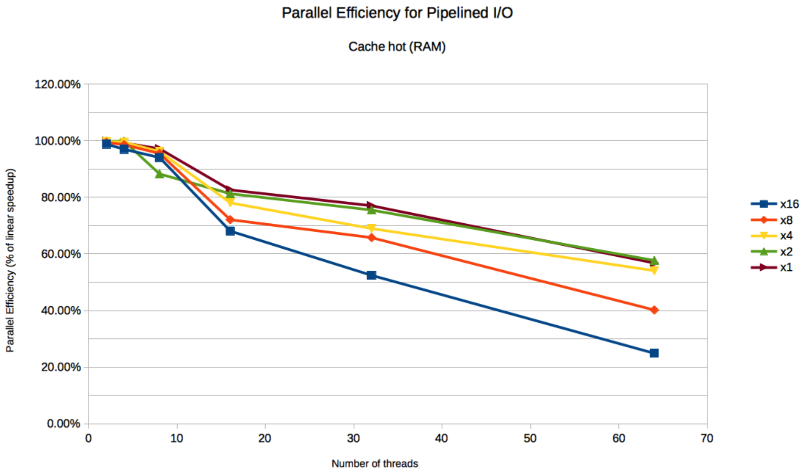
Cache cold HDD


Parallel Efficiency for Pipelined I/O

Cache cold SSD


Parallel Efficiency for Pipelined I/O

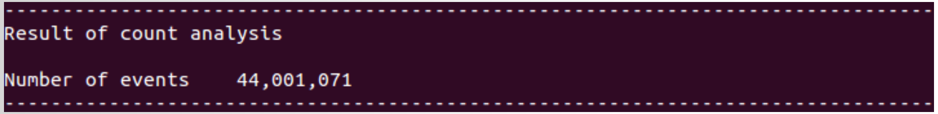
Cache cold PCIe SSD


Parallel Efficiency for Pipelined I/O

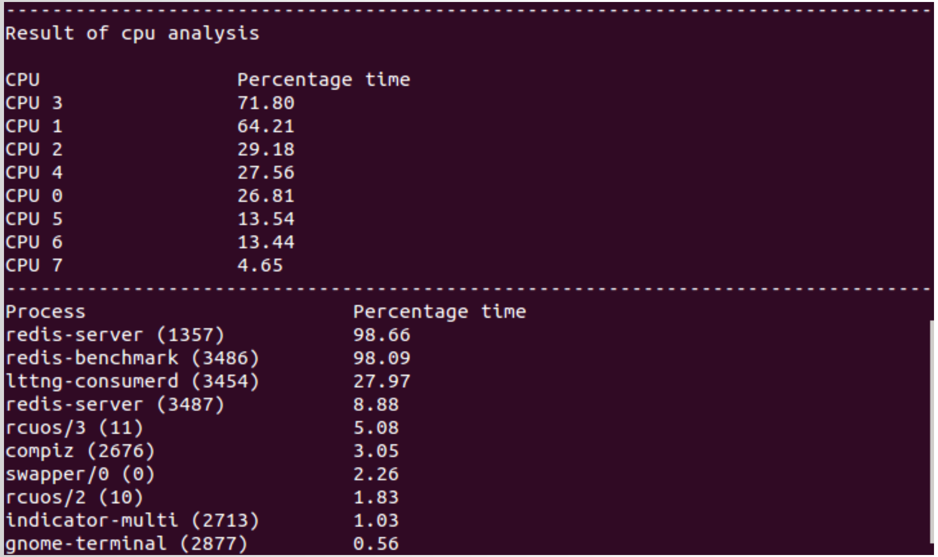
Cache hot (RAM)

EN PREMIERE CLASSE

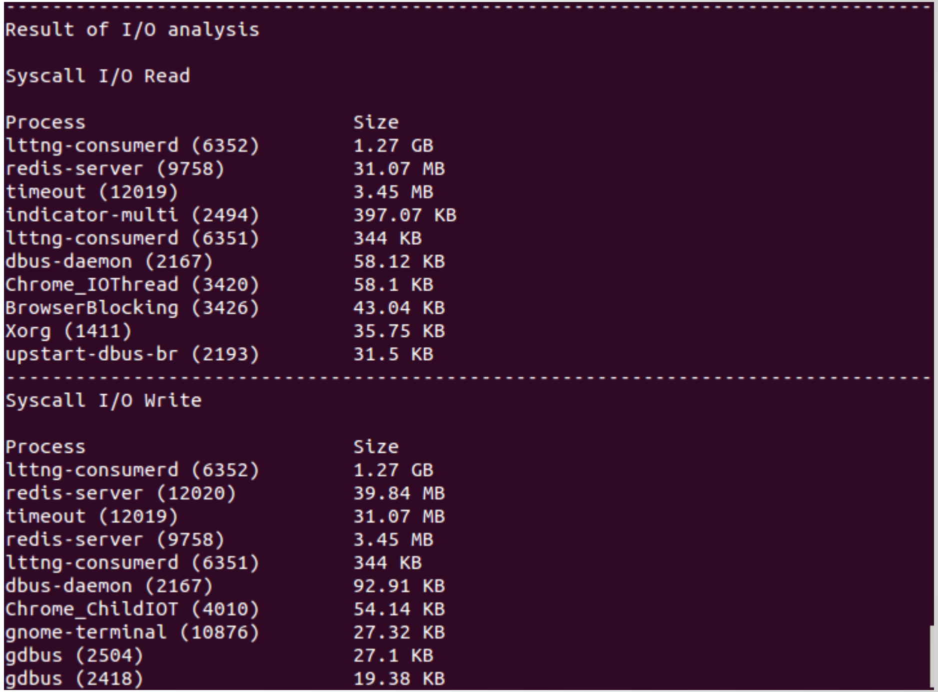
Test analyses



```
--------------------------------------------------------------------------------
Result of count analysis

Number of events    44,001,071
--------------------------------------------------------------------------------
```



```
--------------------------------------------------------------------------------
Result of cpu analysis

CPU                 Percentage time
CPU 3               71.80
CPU 1               64.21
CPU 2               29.18
CPU 4               27.56
CPU 0               26.81
CPU 5               13.54
CPU 6               13.44
CPU 7               4.65
--------------------------------------------------------------------------------
Process                       Percentage time
redis-server (1357)           98.66
redis-benchmark (3486)        98.09
lttng-consumerd (3454)        27.97
redis-server (3487)           8.88
rcuos/3 (11)                  5.08
compiz (2676)                 3.05
swapper/0 (0)                 2.26
rcuos/2 (10)                  1.83
indicator-multi (2713)        1.03
gnome-terminal (2877)         0.56
```

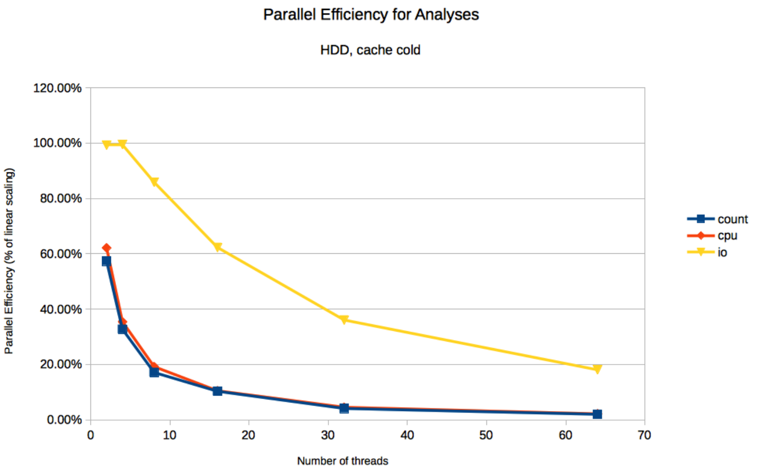


```
--------------------------------------------------------------------------------
Result of I/O analysis

Syscall I/O Read

Process                       Size
lttng-consumerd (6352)        1.27 GB
redis-server (9758)           31.07 MB
timeout (12019)               3.45 MB
indicator-multi (2494)        397.07 KB
lttng-consumerd (6351)        344 KB
dbus-daemon (2167)            58.12 KB
Chrome_IOThread (3420)        58.1 KB
BrowserBlocking (3426)        43.04 KB
Xorg (1411)                   35.75 KB
upstart-dbus-br (2193)        31.5 KB
--------------------------------------------------------------------------------
Syscall I/O Write

Process                       Size
lttng-consumerd (6352)        1.27 GB
redis-server (12020)          39.84 MB
timeout (12019)               31.07 MB
redis-server (9758)           3.45 MB
lttng-consumerd (6351)        344 KB
dbus-daemon (2167)            92.91 KB
Chrome_ChildIOT (4010)        54.14 KB
gnome-terminal (10876)        27.32 KB
gdbus (2504)                  27.1 KB
gdbus (2418)                  19.38 KB
```

Implemented some of the Python analyses made by Julien Desfossez

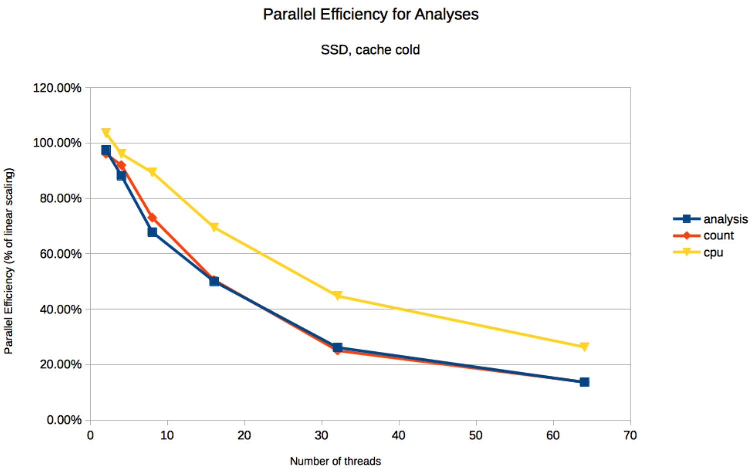
POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE



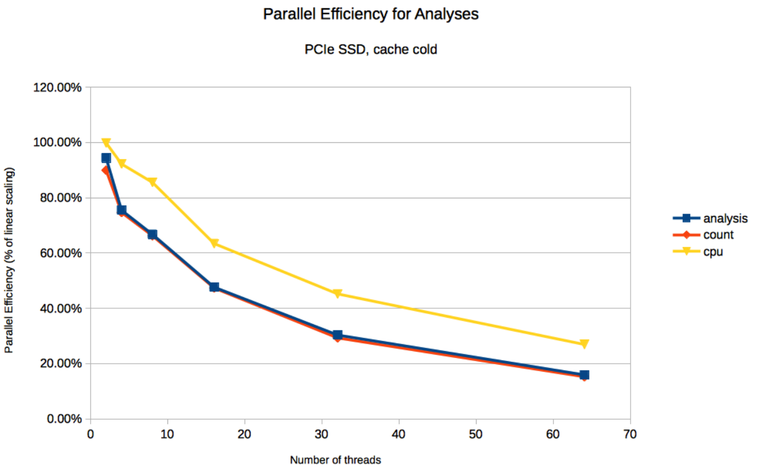
Parallel Efficiency for Analyses

HDD, cache cold



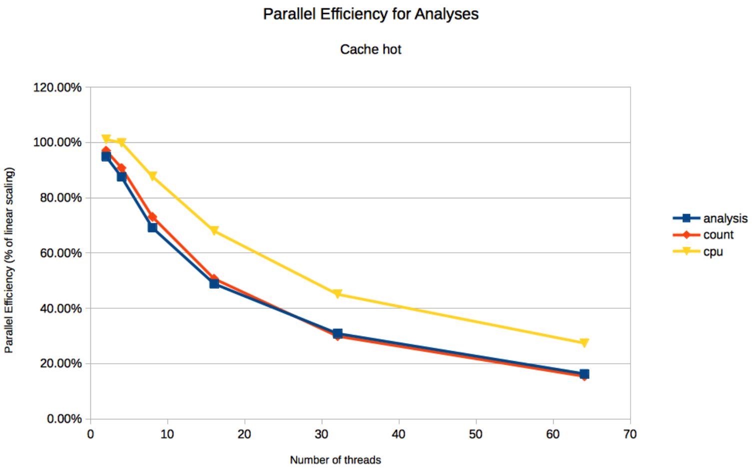
Parallel Efficiency for Analyses

SSD, cache cold



Parallel Efficiency for Analyses

PCIe SSD, cache cold



Parallel Efficiency for Analyses

Cache hot

Speedup for analyses

Trace info: execution of Redis benchmark on 8-core machine
Trace size: 267MB
Trace events: 6,915,790

Analysis	Serial time in ms	Parallel time in ms for 64 threads	Speedup
count	15990	1534	10.42
cpu	17622	1790	9.85
io	68584	3912	17.53

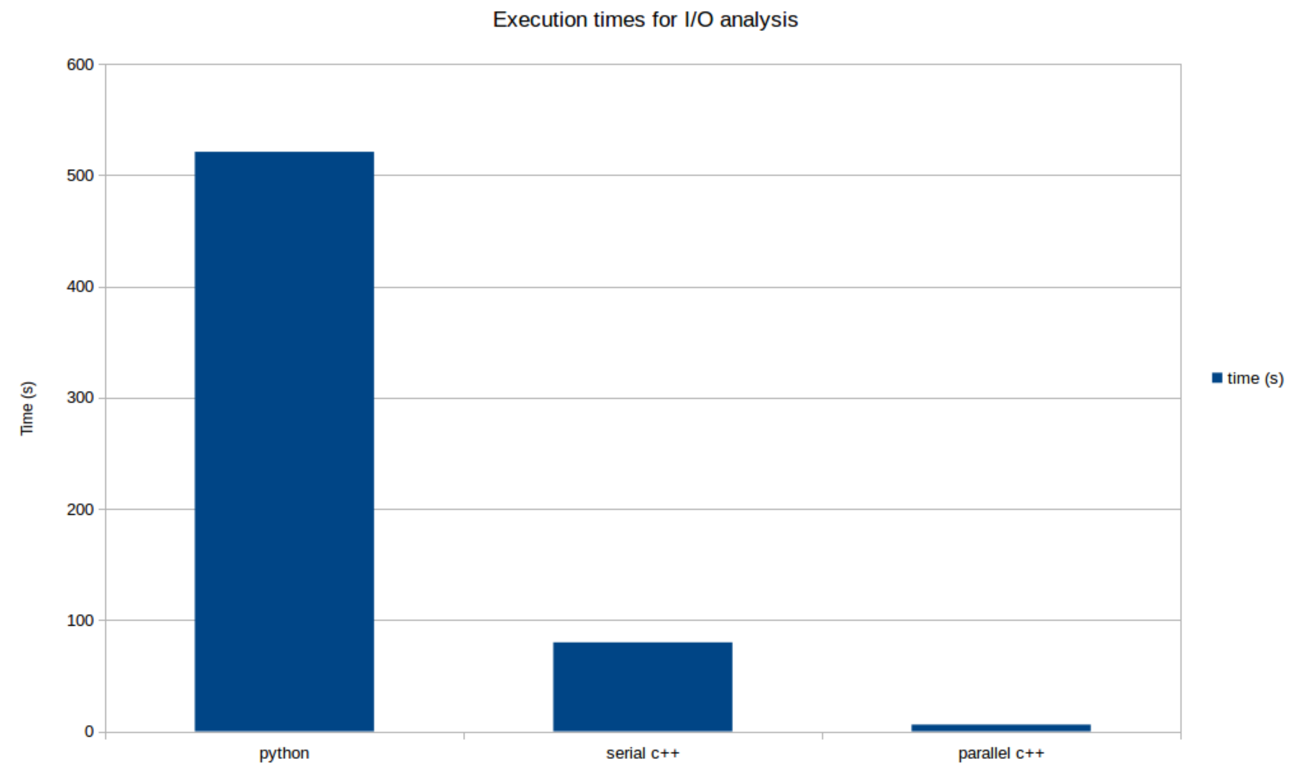
Conclusion

- Parallel processing is a viable way to achieve better, more scalable performance for the analysis of large traces.
- Parallelization will remain relevant as trace decoding improves, especially with recent high-performance disk hardware.
- Parallelizing for 64 cores is very different from parallelizing for 8 cores!

The road ahead

- Short-term goals
    - Pipeline babeltrace I/O
    - Implement other analyses, such as current state, memory
- Medium-term goals
    - Add support for parallelizing the XML state system analysis
    - Output into State History Tree
- Long-term goals
    - Distributed analysis
    - Live tracing analysis

One more thing...



Thank you!

Questions?

POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE