Efficient methods for kernel trace analysis parallelization

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

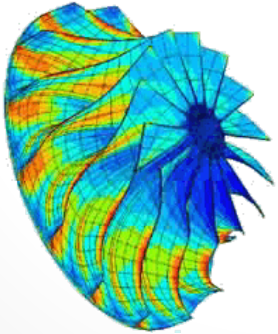
POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Presentation outline

Parallel computing

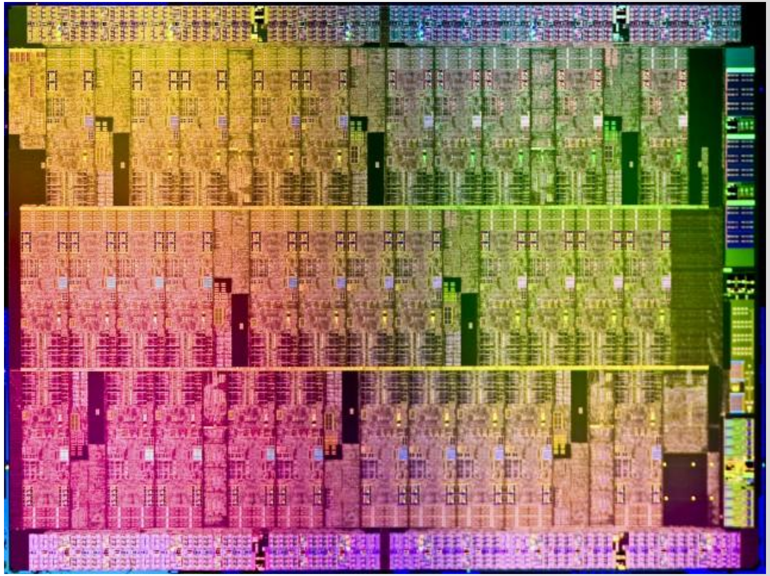
- Modern architectures allow higher and higher levels of parallelism
- Already used in a lot of areas: physics, mechanical engineering, rendering

For example, finite element analysis



Source: http://www.sti-tech.com/images/impell-a.gif

Intel Xeon Phi - 64 cores



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

Can we apply it to trace analysis?

Research objectives

Does the use of parallel computing methods allow for an acceleration of the analysis of kernel traces, which is both efficient and scalable?

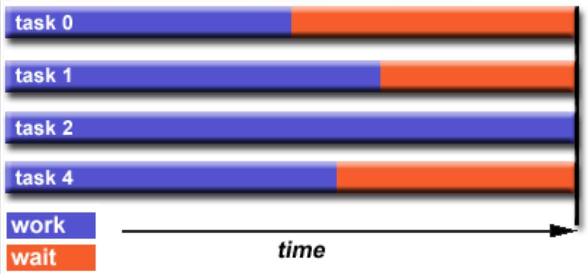
The goal is to develop trace analysis parallelization methods that will:

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)

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

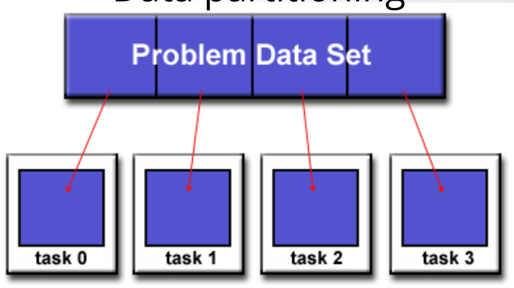
Challenges of parallelization

Load balancing



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

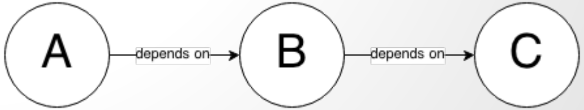
Data partitioning



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

Locking and synchronisation

Data dependencies



POLYTECHNIQUE
MONTRÉAL
LE GÉNIE
EN PREMIÈRE CLASSE

Choice of tools

- Trace Compass (formerly TMF) is the Eclipse-based tool for trace analyses and visualizations
  - Very complete framework, lots of infrastructure for reading traces, analysing them and displaying the results
  - Unfortunately, this also means lots of complexity, making it very hard to experiment with parallelization
- babeltrace is a C library that allows reading CTF traces
  - “Only” provides reading events from traces
  - The simpler design lends itself better to parallelization

Adapting babeltrace to parallel analysis

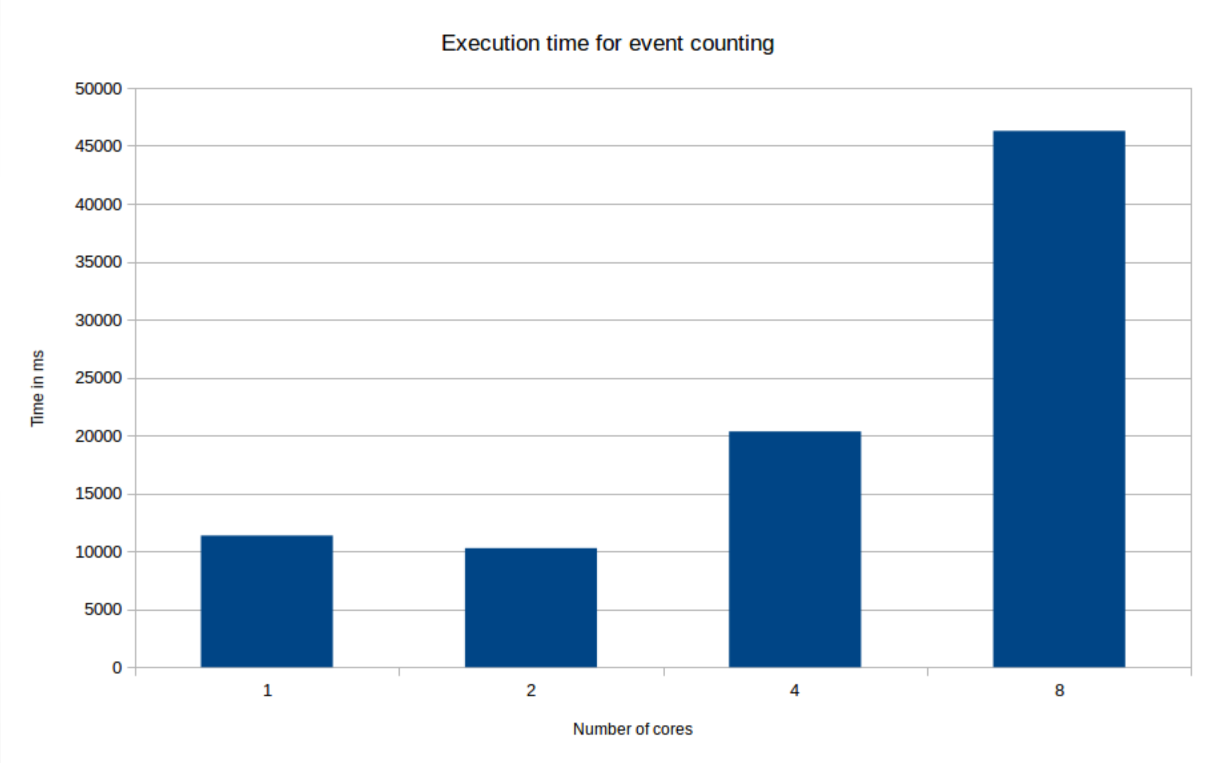
- Babeltrace allows only one iterator per trace
  - Temporary solution : create one context per thread, add trace to each context
  - Very long startup cost, since we have to parse metadata, create structures, etc., *for each thread*
  - Not viable when working with up to 64 cores
- Added support for multiple iterators per trace by cloning file streams inside each iterator
- Then there was also the problem of performance...

Use parallel processing to slow your application down!

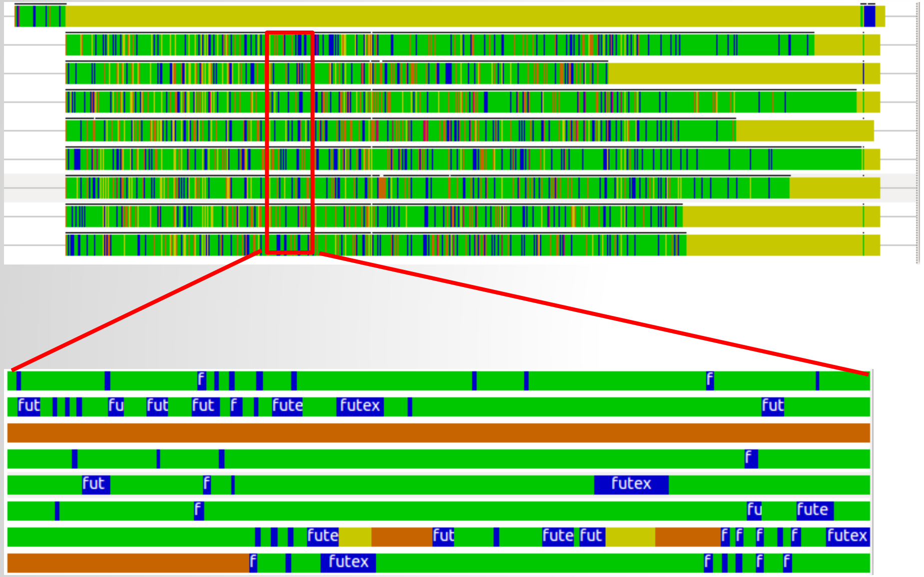
Trace size	5,114,625 events
CPU on analysis machine	AMD FX 9370 Eight-Core Processor

Performance gradually worsens as we add more threads

Something is definitely wrong...



What is happening?



perf record --call-graph

```
Samples: 50K of event 'cycles', Event count (approx.): 47402328831
-  49.16%  lttng-parallel-  libpthread-2.19.so          [.] pthread_mutex_lock
   + pthread_mutex_lock
   - g_mutex_lock
      - 99.93% g_quark_from_string
         - 79.43% bt_lookup_definition
            + 65.39% bt_lookup_integer
            + 17.72% bt_lookup_enum
            + 16.89% bt_lookup_variant
         + 12.19% bt_new_definition_path
         + 4.90% bt_append_scope_path
         + 3.46% _array_definition_new
+   8.82%  lttng-parallel-  libpthread-2.19.so          [.] pthread_mutex_unlock
+   4.48%  lttng-parallel-  [kernel.kallsyms]           [k] _raw_spin_lock
+   2.45%  lttng-parallel-  libglib-2.0.so.0.4002.0     [.] g_mutex_get_impl
+   1.62%  lttng-parallel-  [kernel.kallsyms]           [k] memcpy
+   1.59%  lttng-parallel-  libpthread-2.19.so          [.] __lll_lock_wait
+   1.56%  lttng-parallel-  [kernel.kallsyms]           [k] lttng_event_reserve
+   1.09%  lttng-parallel-  libbabeltrace-ctf.so.1.0.0  [.] ctf_pos_access_ok
+   1.05%  lttng-parallel-  libbabeltrace-ctf.so.1.0.0  [.] _aligned_integer_read
+   0.89%  lttng-parallel-  libglib-2.0.so.0.4002.0     [.] g_private_get_impl
+   0.89%  lttng-parallel-  libc-2.19.so                [.] __GI___strcmp_ssse3
+   0.81%  lttng-parallel-  libglib-2.0.so.0.4002.0     [.] g_hash_table_lookup
+   0.80%  lttng-parallel-  [kernel.kallsyms]           [k] lttng_event_commit
+   0.76%  lttng-parallel-  [kernel.kallsyms]           [k] lttng_event_write
+   0.70%  lttng-parallel-  libbabeltrace.so.1.0.0      [.] generic_rw
+   0.68%  lttng-parallel-  libc-2.19.so                [.] _int_malloc
```

In glib/gquark.c ->

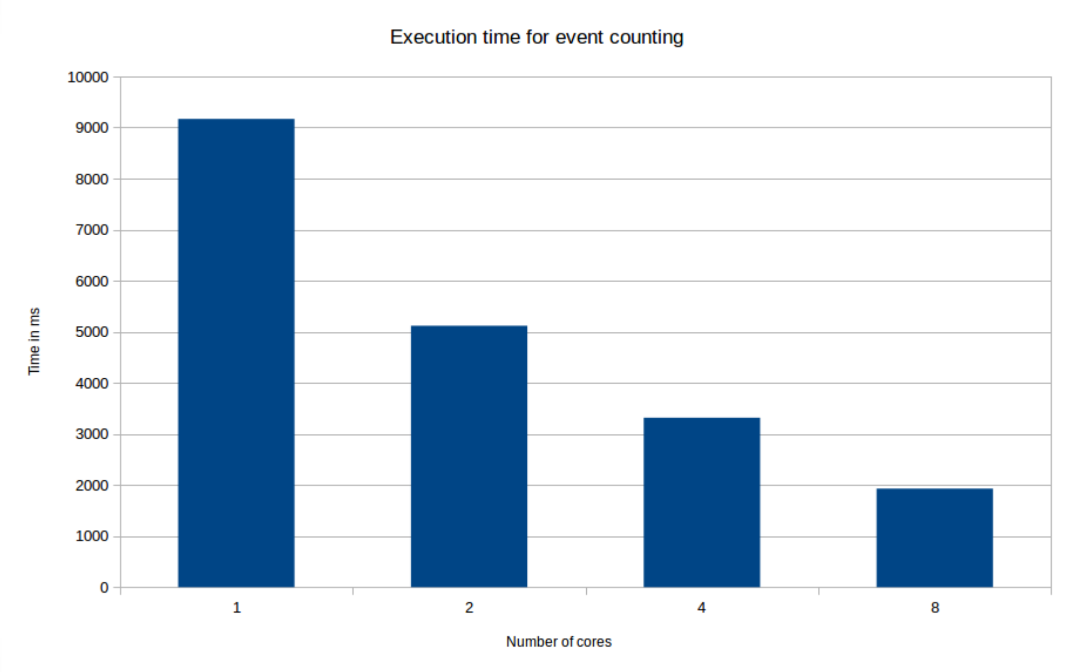
```
GQuark
g_quark_from_string (const gchar *string)
{
  GQuark quark;

  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

After patching things up

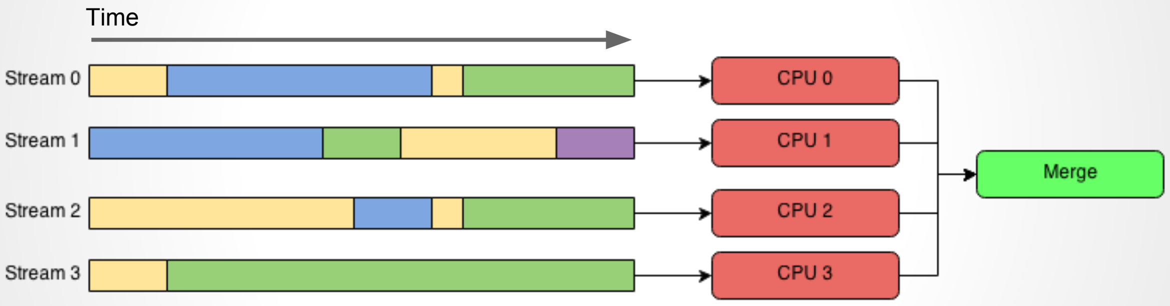
- Added a thread-local quark cache to prevent global lock contention in the glib
- Each thread only queries the global glib quark hash table if it does not have the quark in its cache
- Duplicates data, but the tradeoff is worth it



POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Per-stream data partitioning

- One stream per CPU on the **analyzed** machine
- Each CPU on the **analyzing** machine treats one stream

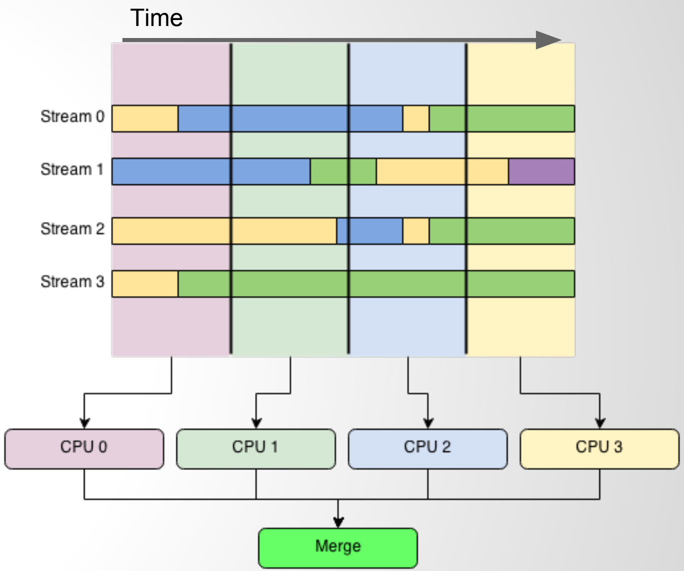


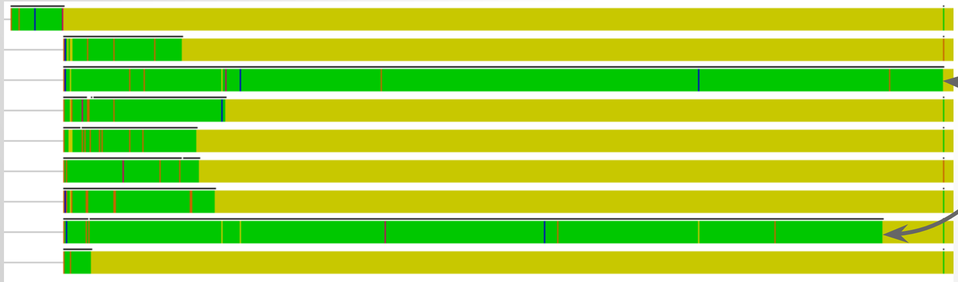
- What about if there are less streams than CPUs?
- Synchronization problems when causality between events (e.g. migrated process)

We need to split “vertically”, i.e. by time

Per-time range partitioning

- Split the trace evenly across streams by **timestamp**
- Each CPU analyzes all the events between two timestamps
- Load balancing: uneven event density
- Data dependency: some events are dependent on prior events
    - E.g. which system call just exited?





These two threads slow down the analysis

POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

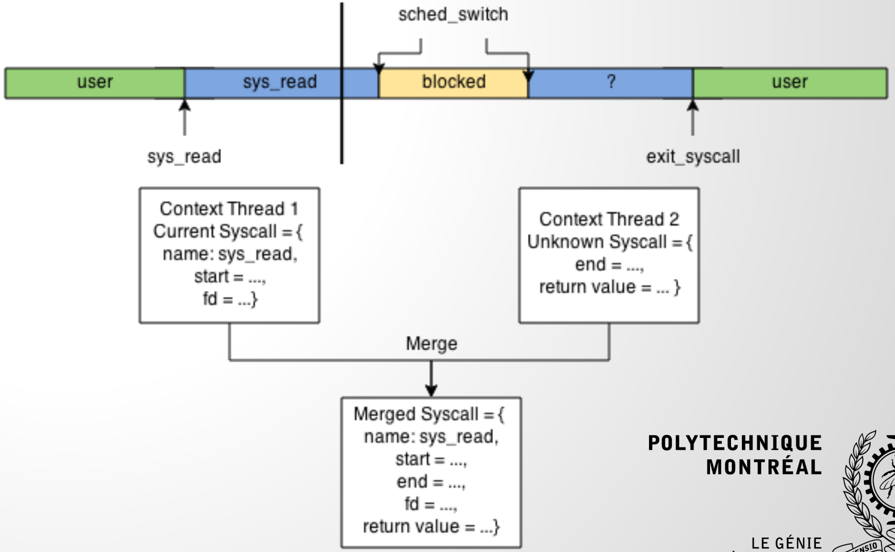
Using CTF packet index to balance load

- 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
- Other advantages:
  - Seeks on packet frontiers are cheaper than within packets
- Disadvantages:
  - Works best only if trace has a lot of packets (not always the case)
  - At the moment only works on single-stream traces

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

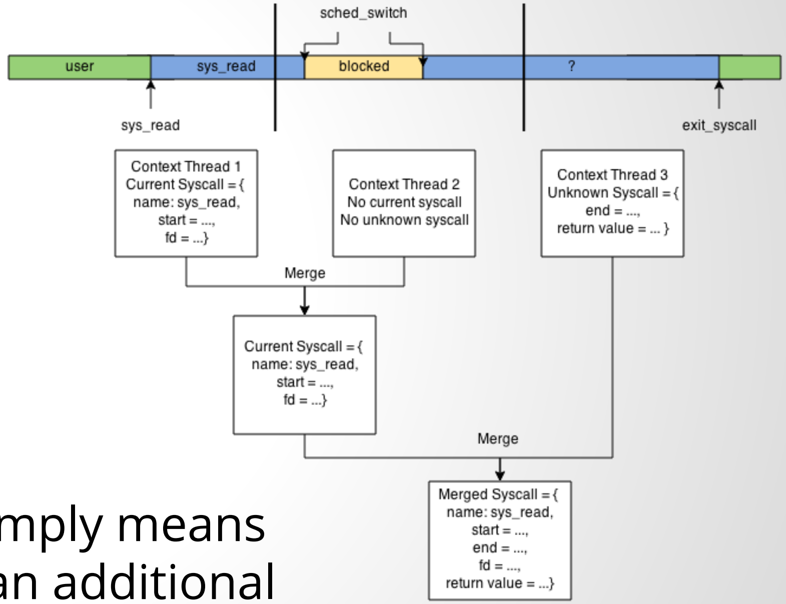


POLYTECHNIQUE MONTRÉAL

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



POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE 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

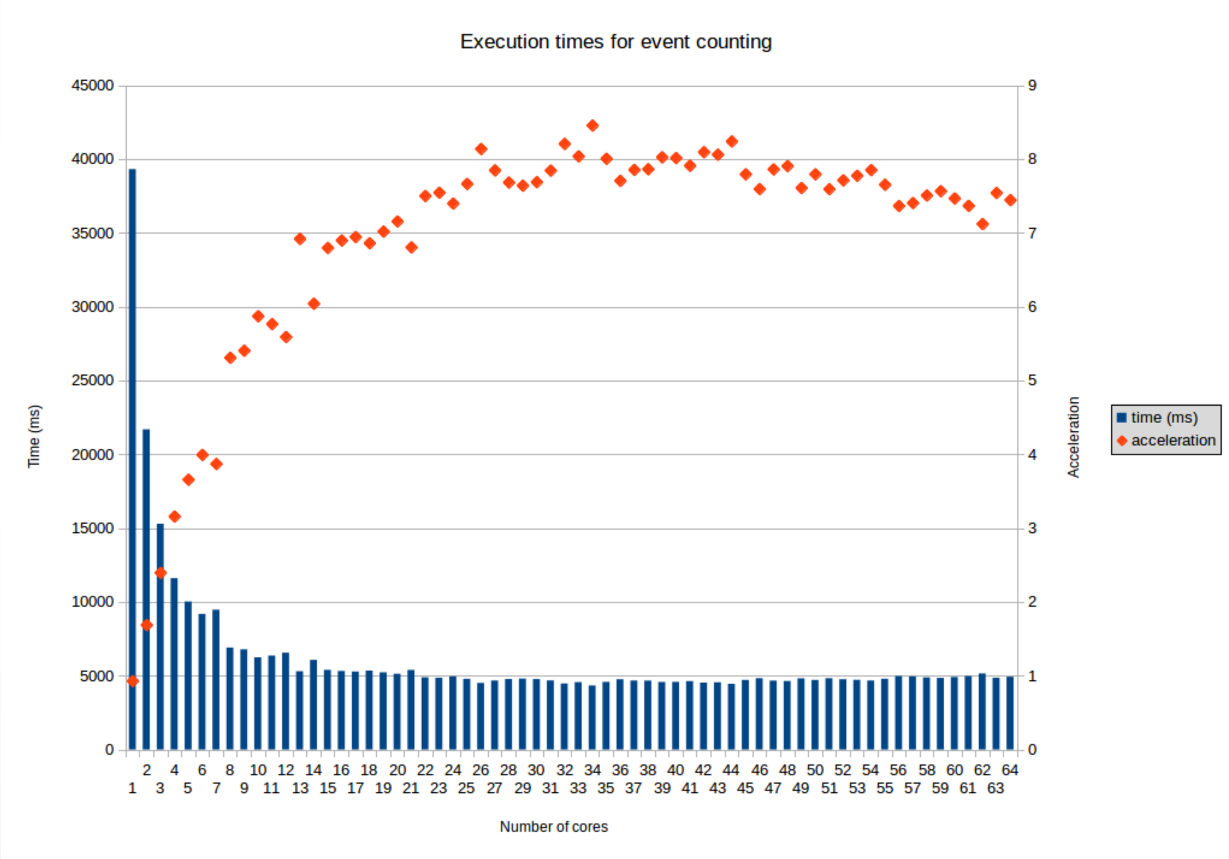
Results - Event counting

Trace size	17,358,022 events
CPU on analysis machine	4 x AMD Opteron 6272 Sixteen-Core Processor
Serial time	36.7 s
Parallel time (best)	**4.5 s** w/ 26 threads

Count the number of events in a trace

Maximal acceleration: ~8.2x

Worst-case scenario:
Lots of I/O for little CPU work



Results - CPU analysis

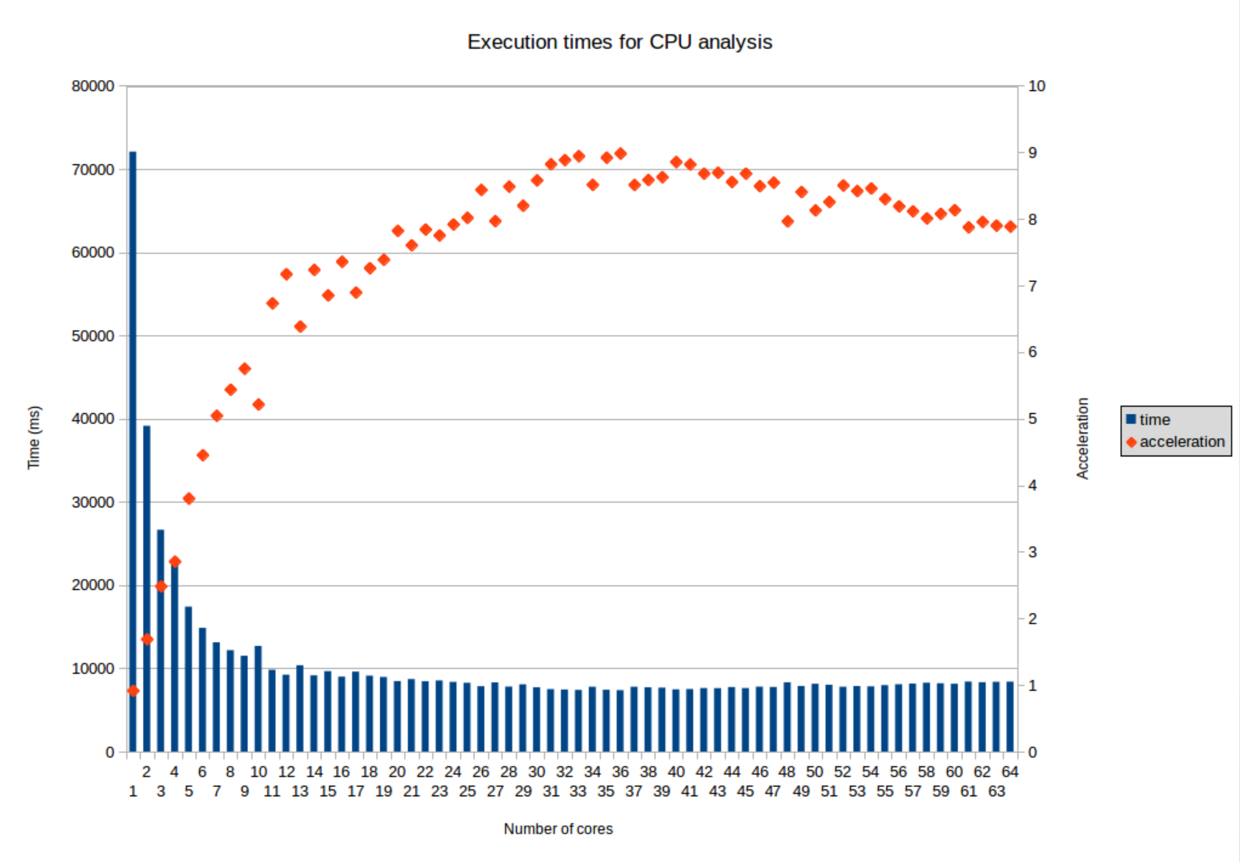
Trace size	44,001,071 events
CPU on analysis machine	4 x AMD Opteron 6272 Sixteen-Core Processor
Serial time	66.3 s
Parallel time (best)	**7.4 s** w/ 32 cores

Measure the proportion of CPU active vs idle + CPU usage per thread

Maximal acceleration: ~9x

Only uses scheduling events (*sched_switch*)

Very simple analysis



Results - IO analysis

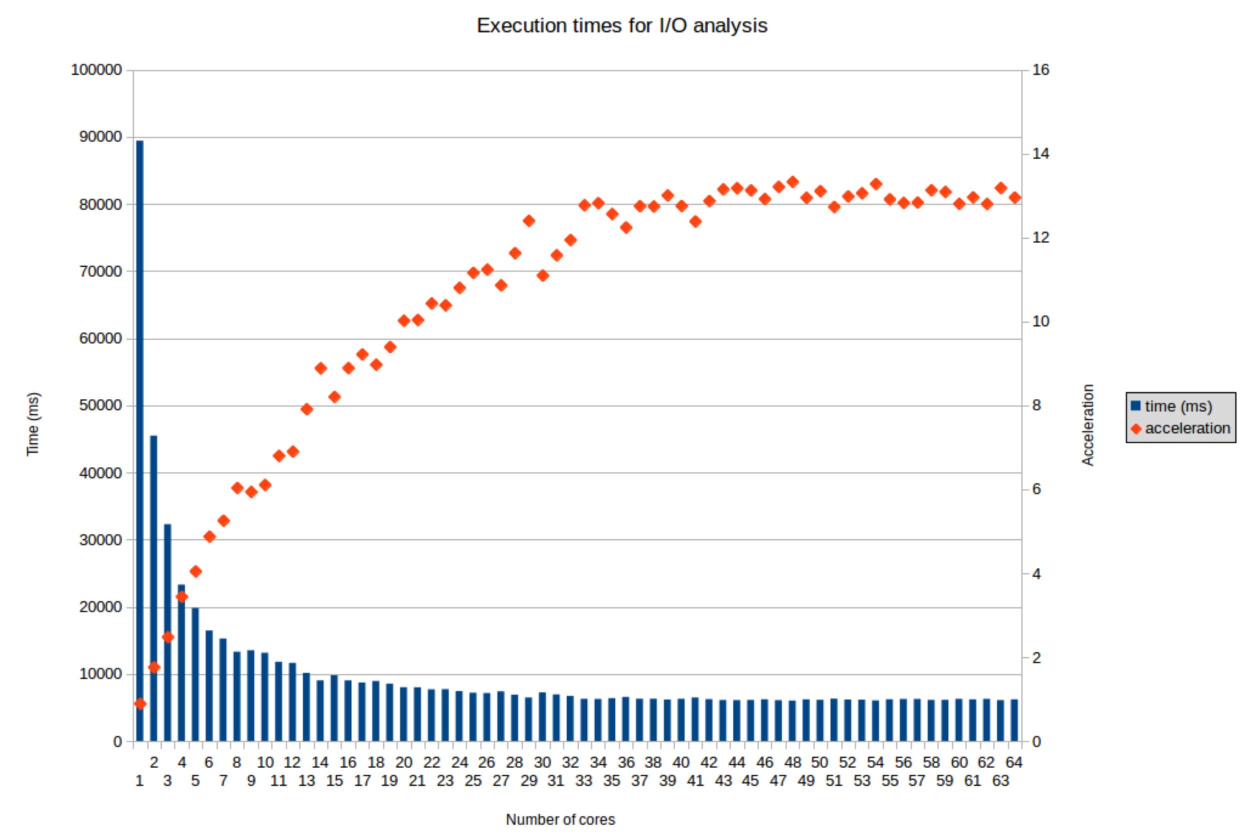
Trace size	17,358,022 events
CPU on analysis machine	4 x AMD Opteron 6272 Sixteen-Core Processor
Serial time	80.3 s
Parallel time (best)	**6.0 s** w/ 48 cores

Measure syscall read/write I/O per thread

Maximal acceleration: ~13.3x

Uses I/O syscalls (*sys_read*, *sys_write*, *sys_splice, etc.*)

Slight increase in complexity brings much better scaling



The road ahead

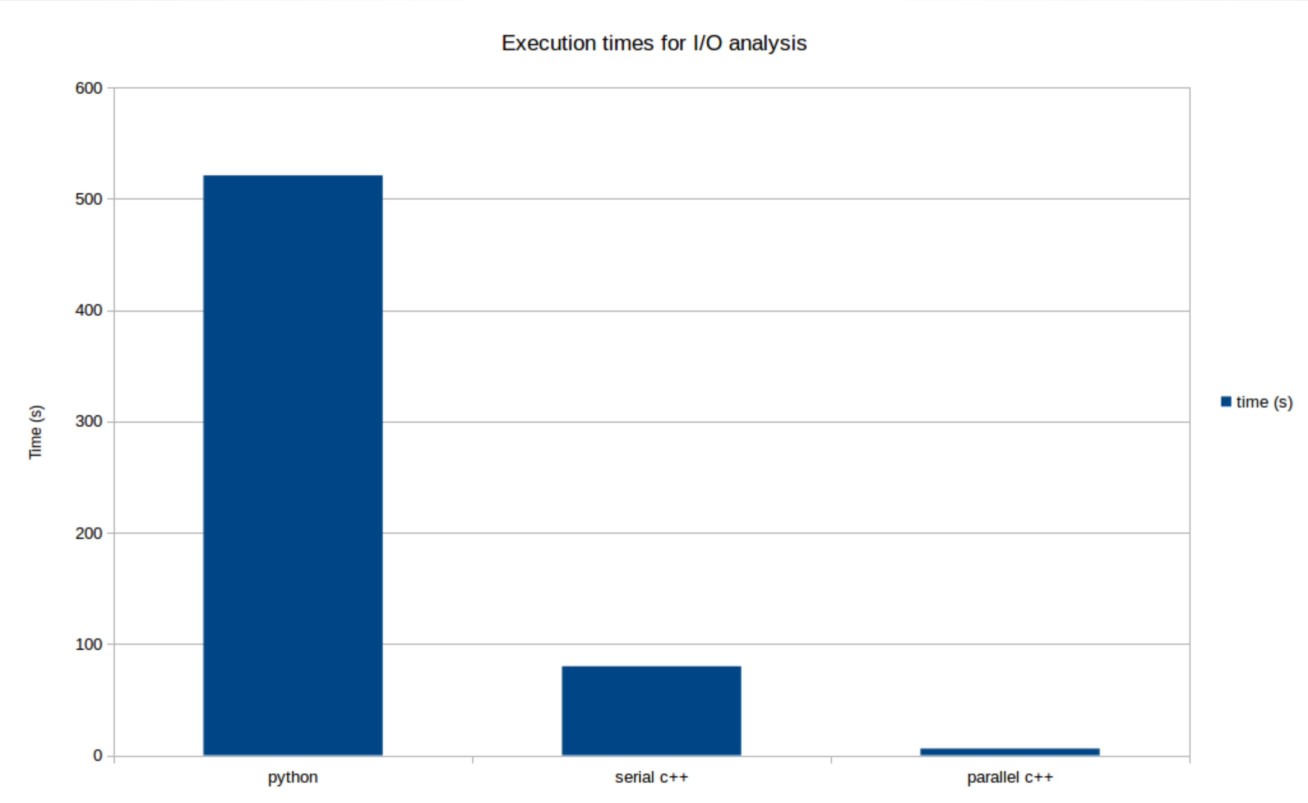
- Short-term goals
  - Apply to more types of analyses, such as current state, memory
  - Better load balancing through a hybrid per-stream/per-time range partitioning
- Medium-term goals
  - Add support for parallelizing the XML state system analysis
  - Output into State History Tree
- Long-term goals
  - Distributed analysis
  - Live tracing analysis

Conclusion

Research question:
Does the use of parallel computing methods allow for an acceleration of the analysis of kernel traces, which is both efficient and scalable?

The preliminary results seem to indicate that parallel processing is a viable way to achieve better, more scalable performances for the analysis of large traces.

One more thing...



Thank you!

Questions?