GAPP: A Fast Profiler for Detecting Serialization Bottlenecks in Parallel Linux Applications

Reena Nair   Tony Field
What causes serialization bottlenecks?

- Resource Contention
- Load Imbalance

**Hardware**
- CPU
- Peripherals

**Software**
- Locks

![Graph showing execution time versus thread ID](image-url)
Serialization Bottlenecks – Reduced Parallelism

Max Parallelism

Reduced Parallelism

Thread Blocked
There are many different sources of bottlenecks.

Profilers for debugging performance issues

- Memory
- Locks
- Peripherals
- Critical Thread

- Profiler A
- Profiler B
- Profiler C
- Profiler D
GAPP – Generic Automatic Parallel Profiler

- Can identify several different types of serialization bottlenecks.
- No need to instrument the application.
- Validated on multithreaded and multi-process parallel applications written in C/C++.
- Implemented using extended Berkley Packet Filter (eBPF).
  - Provides fast and secure kernel tracing (~4% average runtime overhead).
Harness the symptom rather than the cause

- Identify **when** and **where** reduced parallelism is exhibited
  - Number of active threads, $N_{act} \leq N_{min}$, a tuneable threshold variable with a default value of $N/2$, where $N$ is the total number of threads
  - Trace context switch events in the kernel.
  - Retrieve stack trace at the end of a time slice

- Reduce overhead - retrieve stack traces only from critical time slices
- Critical time-slice – whose average active thread count is $\leq N_{min}$
- Omit ST$_2$
Are stack traces enough to identify bottleneck?

- Stack traces retrieved at the end of a time-slice would point to bottleneck code only if it happened to execute at the end of a time-slice.

Missed Bottleneck?

Active Threads: 1 2 1 3 2 1
Stack Traces (ST): ST₁ ST₂ ST₄ ST₃
Combining bottleneck code and call paths

- Periodically sample instruction pointers.
- Reject samples if $N_{act} > N_{min}$
- Combine instruction pointers and stack traces of each critical time-slice
- Each critical time-slice is assigned a metric, Criticality Metric ($C_{metric}$), which takes into account the duration and degree of parallelism of a time-slice.

---

Ranking Bottlenecks

• Similar call paths, their samples and CMetric are combined and sorted to display potential critical call paths, functions and lines of codes and Cmetric of individual threads.

Critical Path 1:
  deflate_slow()
  <---deflate()
  <---compress()
  <---Compress()

Functions and lines + Frequency
  deflation: slow – 1465
  deflate.c:1650 (StackTop) -- 575
  deflate.c:1580 -- 354

Optimization Opportunities

<table>
<thead>
<tr>
<th>ThreadID</th>
<th>CMetric</th>
</tr>
</thead>
<tbody>
<tr>
<td>25778</td>
<td>256130902</td>
</tr>
<tr>
<td>25779</td>
<td>417320962</td>
</tr>
<tr>
<td>25783</td>
<td>5003332502</td>
</tr>
<tr>
<td>25784</td>
<td>5003756997</td>
</tr>
</tbody>
</table>

Load Imbalance, if any
GAPP - Evaluation

- Evaluated using applications from the Parsec-3.0 benchmark suite and two large open source projects, MySQL and Nektar++.

- All applications except Nektar++ were multithreaded
  - Each was executed with 64 threads.

- Nektar++, a spectral/hp element framework which uses message passing, was executed with 16 MPI processes.
Load imbalance from thread CMetric
Multithreaded Task Parallel Application - Ferret

- Six pipeline stages - first and last stages perform I/O with single threads.

![Ferret pipeline stages with initial thread allocation](image)

**Critical Path 1:**
- `emd()`
- `sdist_emd()`
- `raw_query()`
- `cass_table_query()`
- `t_rank()`
- `start_thread()`

**Functions and lines + Frequency**
- `isOptimal` -- 41314
- `emd.c:422` -- 20813
- `emd.c:423` -- 10760
- `emd.c:420` -- 6657
- `findBasicVariables` -- 41301
- `emd.c:350` -- 7366
- `emd.c:353` -- 6713
- `emd.c:383` -- 5827

![GAPP Profile for Ferret](image)
Optimizing Ferret by thread reallocation

- Ranking phase exhibited higher CMetric when compared to other stages.
- Optimized by re-allocating threads to ranking phase.

Fig: Cmetric for different thread allocations - Ferret
Resource Contention – MySQL
Sysbench OLTP_read_write workload

Critical Path 1

fil_flush()[mysqld]
<---log_write_up_to()
<--trx_commit_complete_for_mysql()
<---innobase_commit()
<---ha_commit_low()
<---TC_LOG_DUMMY::commit()
<---ha_commit_trans()
<---trans_commit()
<---mysql_execute_command()
<---Prepared_statement::execute()

Functions and lines + frequency
pfs_os_file_flush_func -- 1462
os0file.ic:507 (StackTop) -- 1462

Critical Path 2

sync_array_reserve_cell()
<---rw_lock_s_lock_spin()
<---pfs_rw_lock_s_lock_func()
<---row_search_mvcc()
<---ha_innobase::index_read()
<---handler::ha_index_read_idx_map()
<---join_read_const_table()
<---JOIN::extract_func_dependent_tables()
<---JOIN::make_join_plan()
<---JOIN::optimize()

Functions and lines + frequency
sync_array_reserve_cell() -- 469
sync0arr.cc:389 (StackTop) -- 469

Disk I/O

Spin-wait Loop
Optimizing MySQL

Critical Function1
(Hardware Resource Contention)

- `pfs_os_file_flush_func()`
  - Invoked by InnoDB, flushes write buffers to disk
  - Increasing buffer size improved transaction rate by 19% and reduced latency by 16%

Critical Function2
(Software Resource Contention)

- `sync_array_reserve_cell()`
  - Invoked from a custom built spin lock, that blocks after spinning for a predefined time.
  - Increasing spin wait time reduced cache misses by 10.6%

These 2 modifications cumulatively improved query transaction rate by 34% and reduced average latency by 25%.
Bodytrack – Parsec3.0

Multithreaded application that follows producer-consumer paradigm

Main Producer Loop

Images

Queue

AsyncIO Thread

Critical Call Path1
void FlexDownSample2 ()
<---TrackingModel::OutputBMP()
<---mainPthreads()
<---main ()

Improved performance by 22%
GAPP on MPI Applications

- Nektar++ - a spectral/hp element framework that implements several PDE solvers.
- Evaluated using the Incompressible Navier-Stokes Solver with 16 MPI processes.

- Load imbalance was found to be due to non-uniform partitioning of the mesh.

![Fig:Cmetric of Individual Processes](image)
GAPP Profile - Nektar ++

For each critical path:

Critical Path 1:

```c
__GI___poll ()[libc-2.27.so]
<---MPIDI_CH3I_Progress ()[libmpi.so.12.1.1]
<---MPIC_Wait ()[libmpi.so.12.1.1]
<---MPIC_Recv ()[libmpi.so.12.1.1]
<---MPIR_Bcast_binomial ()[libmpi.so.12.1.1]
<---MPIR_Bcast_intra ()[libmpi.so.12.1.1]
<---MPIR_Bcast ()[libmpi.so.12.1.1]
<---MPIR_Bcast_impl ()[libmpi.so.12.1.1]
<---MPIR_Allreduce_intra ()[libmpi.so.12.1.1]
<---MPIR_Allreduce_impl ()[libmpi.so.12.1.1]
```

Combining functions and lines from critical paths:

Top Critical Functions and lines + Frequency:

```
dgemv_ () [libblas.so.3.8.0] -- 781

double Vmath::Dot2<double>()
[libLibUtilities-g.so.5.0.0b] -- 170

gather_double_add ()
[libMultiRegions-g.so.5.0.0b] -- 100
```

Functions and lines + Frequency:

```
dgemv_ () [libblas.so.3.8.0] -- 594

double Vmath::Dot2<double>()
[libLibUtilities-g.so.5.0.0b] -- 116

gather_double_add ()
[libMultiRegions-g.so.5.0.0b] -- 58
```
- **Bottleneck Function** – matrix multiplication routine exported by the BLAS library.
- **Replacing the default BLAS libraries with OpenBLAS improved run time by 27%.**

---

**Before Optimization**

- Function Name: F1
- Function Name: F2
- Function Name: F3

**After Optimization**

- Function Name: F2
- Function Name: F4
- Function Name: F1

**Bottleneck Function** (dgemv)
Conclusion

- GAPP was able to identify different types of serialization bottlenecks in different class of applications.
- Robust
  - Consistent results across multiple runs under the same test condition.
- Customizable
  - Tuneable parameters: $N_{\text{min}}$, sampling frequency, stack depth, option to include results from dynamic libraries
- Limitation
  - Will not work with spin-wait loops which doesn’t block.
- Available at
  - https://github.com/RN-dev-repo/GAPP
Thank You