

A brief introduction to profiling with the Intel toolchain

Mark Turner and Mark Dixon
Advanced Research Computing

July 7th, 2022

Welcome and icebreaker

Thank you for joining!

- Meet the team
- 1 hour course (12pm-1pm)
- Please do post in the chat or speak up when questions arise
- Feel free to post something in the chat about where in the university you come from and something about your research interests



Course structure

- What is performance analysis?
- Automatic tracing of MPI codes using ITAC
- Leveraging the Intel Traceanalyzer
- User defined instrumentation using ITAC
- An exercise...




What is performance analysis?

Strongly depends on what you are trying to investigate / achieve



quick-bench.com

 Quick C++ Benchmark

Run Quick Bench locally

Support Quick Bench Suite ▾ More ▾

```
1 #include <memory>
2
3 namespace {
4     constexpr int TEST_VAL = 10;
5 }
6
7 static void c_style_ptr(benchmark::State& state) {
8     for (auto _ : state) {
9         auto ptr = new int(TEST_VAL);
10        benchmark::DoNotOptimize(ptr);
11        delete ptr;
12    }
13 }
14 BENCHMARK(c_style_ptr);
15
16 static void unique_ptr(benchmark::State& state) {
17     for (auto _ : state) {
18         auto ptr = std::make_unique<int>(TEST_VAL);
19        benchmark::DoNotOptimize(ptr);
20    }
21 }
22 BENCHMARK(unique_ptr);
23
24 static void shared_ptr(benchmark::State& state) {
25     for (auto _ : state) {
26         auto ptr = std::make_shared<int>(TEST_VAL);
27        benchmark::DoNotOptimize(ptr);
28    }
29 }
30 BENCHMARK(shared_ptr);
```

compiler = Clang 13.0 ▾

std = c++20 ▾




optim = O3 ▾

STL = libstdc++(GNU) ▾

⚙ Run Benchmark

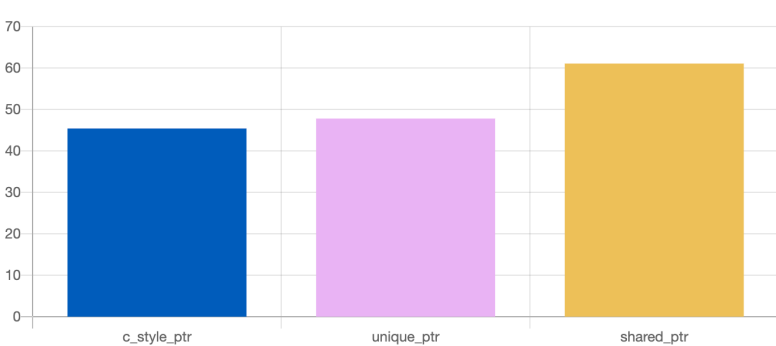
☒ Record disassembly

☐ Clear cached results



Charts

Assembly



Pointer Type	Ratio (CPU time / Noop time)
c_style_ptr	~45
unique_ptr	~48
shared_ptr	~62

godbolt.org

COMPILER EXPLORER Add... More Site Templates C++ North - The Canadian C++ Conf July 17-20 x Backtrace intel Solid Sands Share Policies Other

C++ source #1 X x86-64 clang 13.0.0 (C++, Editor #1, Compiler #1) x

A C++ x86-64 clang 13.0.0 -std=c++2a -O0 Libraries (1) + Add new... Add tool...

```
1 int main() {
2     constexpr int length = 10, width = 5;
3     return length * width;
4 }
```

```
1 main:                                     # @main
2     push    rbp
3     mov     rbp, rsp
4     mov     dword ptr [rbp - 4], 0
5     mov     dword ptr [rbp - 8], 10
6     mov     dword ptr [rbp - 12], 5
7     mov     eax, 50
8     pop     rbp
9     ret
```

COMPILER EXPLORER Add... More Site Templates C++ North - The Canadian C++ Conf July 17-20 x Backtrace intel Solid Sands Share Policies Other

C++ source #1 X x86-64 clang 13.0.0 (C++, Editor #1, Compiler #1) x

A C++ x86-64 clang 13.0.0 -std=c++2a -O3 Libraries (1) + Add new... Add tool...

```
1 int main() {
2     constexpr int length = 10, width = 5;
3     return length * width;
4 }
```

```
1 main:                                     # @main
2     mov     eax, 50
3     ret
```



Measuring like this is great, but...

how do we study performance of entire codes – in particular, those that run across supercomputers?

What if we want to understand the behaviour of an entire simulation?

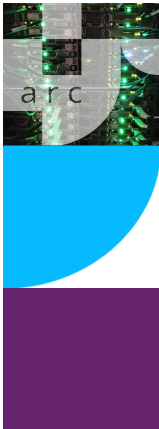


A flavour of performance analysis in HPC

- Identifying code flaws:
 - Hotspots (always a good place to focus dev time)
 - Load imbalance
 - Inefficient communication patterns
 - Poor scaling
 - Inefficient use of hardware (cache misses, excessive I/O etc.)
- Establishing the obstacles to being **compute bound**
 - Memory bound; MPI bound



The
aim!



How can tools understand our codes?

- **Sampling**

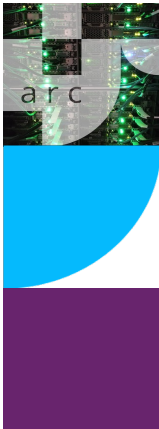
- Regular reading of typical averages, patterns etc.
- Recording runtime characteristics (e.g., hardware counters)
- `likwid-perfctr`

- **Tracing**

- Chronologically ordered events/timelines
- Logging **events** (e.g. function calls) in code



For today, we're interested in tracing



Intel Trace Analyzer and Collector

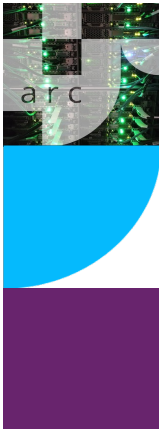


- A graphical tool for profiling and understanding MPI behaviour
- Presents the chronology of your runtime in a visual way
- Easier to reason about behaviour of parallel codes and identify:
 - temporal dependencies and bottlenecks
 - load imbalance
 - Hotspots
 - And more...





**So, on the surface, what does
ITAC give us?**



A high-level overview page:

Summary: peano4.stf

Total time: **1.07e+04** sec. Resources: **2** processes, **4** threads, **1** node.

Continue >

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.



Serial Code	- 1.03e+04 sec	96.9 %
OpenMP	- 0 sec	0 %
MPI calls	- 320 sec	3 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Iprobe	244 sec (4.58 %)
MPI_Test	75.7 sec (1.42 %)
MPI_Isend	0.0473 sec (0.000887 %)
MPI_Comm_dup	0.0466 sec (0.000873 %)
MPI_Irecv	0.0178 sec (0.000335 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:

Intel® VTune™ Profiler for:

- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

Intel® Advisor for:

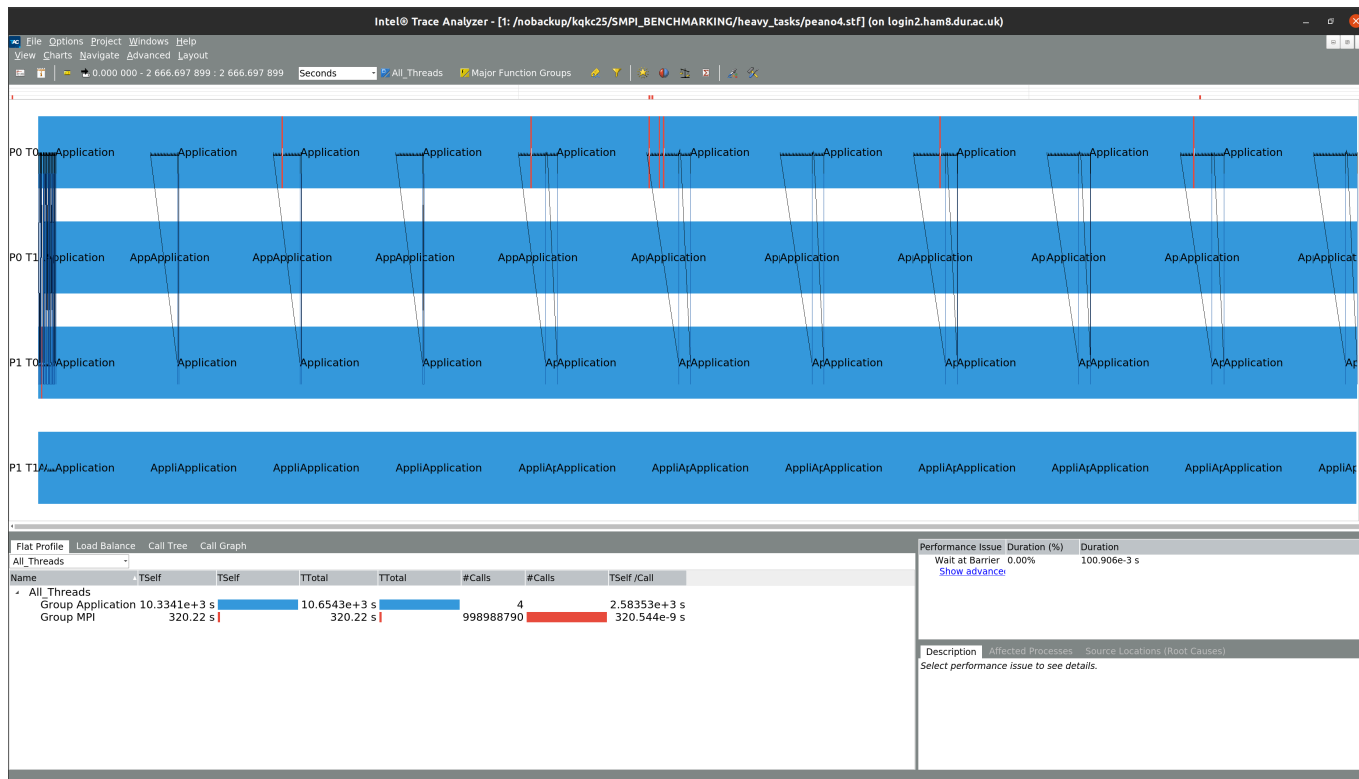
- vectorization optimization and thread prototyping.

For more information, see documentation for the respective tool:

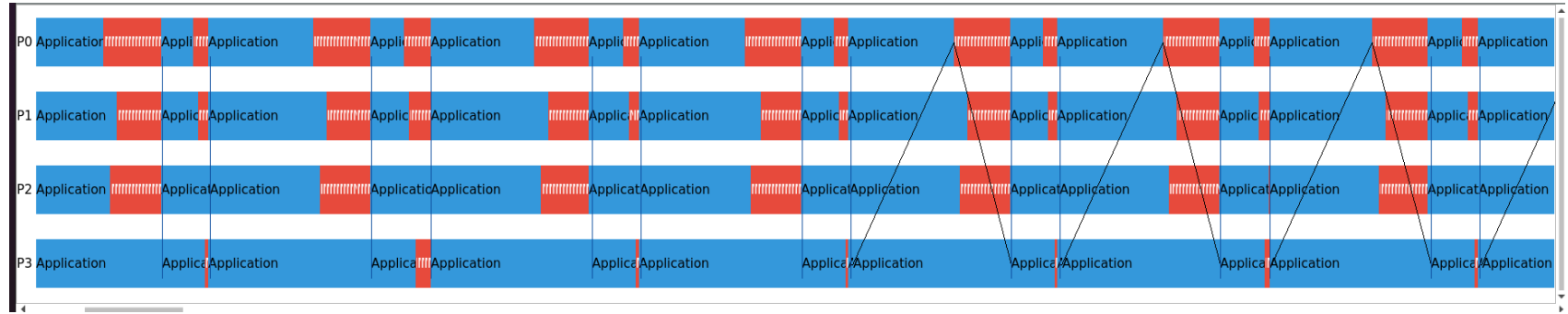
[Analyzing MPI applications with Intel® VTune™ Profiler](#)

[Analyzing MPI applications with Intel® Advisor](#)

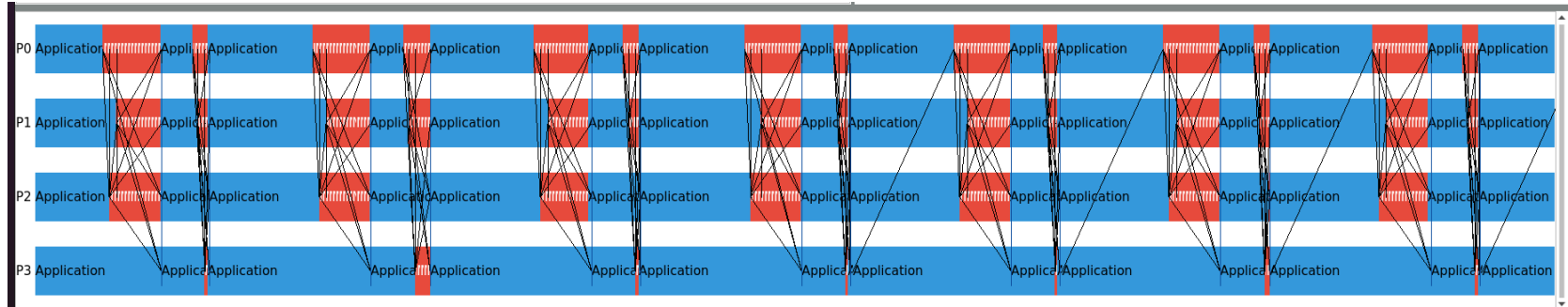
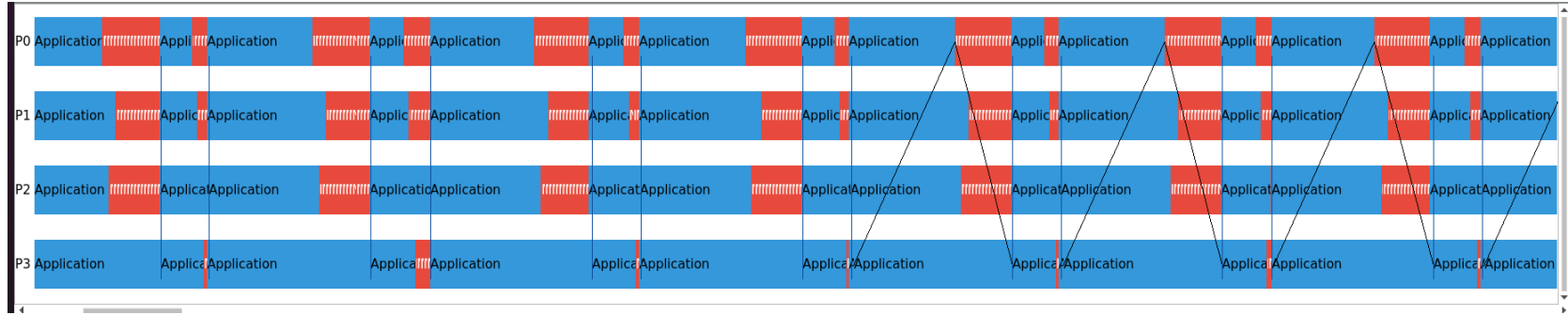
An event timeline:



A more problematic snapshot:



Ability to filter MPI communications:



A demo

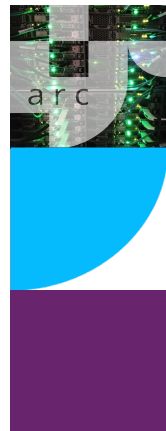


How to navigate the traceanalyzer and explore an event timeline





**And how can I generate these profiles
for my code on Hamilton?**



Generating a trace on Hamilton 8



- Compile with Intel MPI (Open MPI is not supported...)
- Load the ITAC module:
 - `module load itac`
- The trace files can be **BIG**. It is worth Asking SLURM for more temporary disk space on your nodes when doing big runs:
 - `#SBATCH --gres=tmp:400G`
- Then pass the `-trace` flag through to mpirun:
 - `mpirun -trace -np <N> <executable>`



Exercise

Ask us questions!



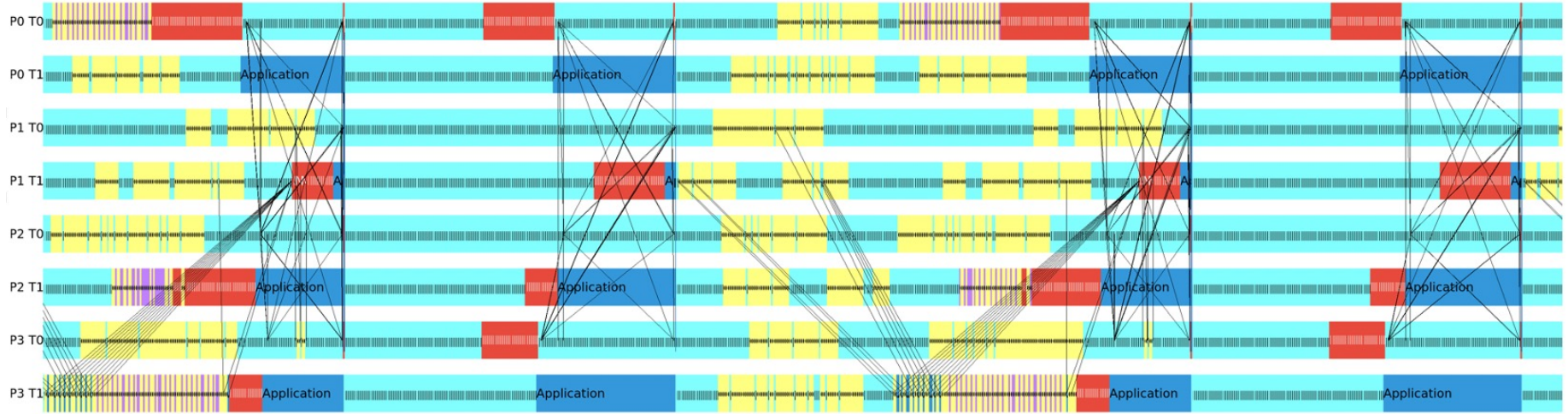


**So far, we have the tools to better
understand MPI behaviour**

**... But how can we visualize
application behaviour?**



User defined instrumentation:

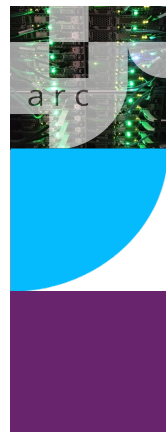


- Red = MPI
- Light Blue = AMR framework
- Yellow = Physics
- Dark blue = Open MP

Zoom in to see more detail. Right click a cell and select “ungroup” to view function call.



**And how can I generate these profiles
for my code on Hamilton?**



Generating a trace on Hamilton 8



1. Add instrumentation boilerplate

```
namespace {
    std::map<std::string, const int> itac_handles;
}

void trace_in(const std::string& trace) {
    if (itac_handles.count(trace) == 0) {
        int new_handle;
        VT_funcdef( trace.c_str() , VT_NOCLASS, &new_handle );
        itac_handles.insert({trace, new_handle});
    }

    VT_begin(itac_handles[trace]);
}

void trace_out(const std::string& trace) {
    VT_end(itac_handles[trace]);
}
```



Generating a trace on Hamilton 8



1. Add instrumentation boilerplate
2. Add your code (whatever that might be 😊)

```
void do_fancy_physics(const int rank) {  
    float aggregator = 0;  
    for (int i = 0; i < 100'000'000; ++i) {  
        aggregator += sin(i);  
    }  
    std::cout << "Result on " << rank << ": " << aggregator << std::endl;  
}
```

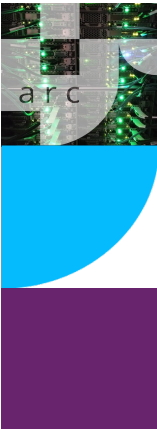


Generating a trace on Hamilton 8



1. Add instrumentation boilerplate
2. Add your code (whatever that might be 😊)
3. Add tracing around your code

```
int main(int argc, char* argv[]) {  
    MPI_Init(&argc, &argv);  
    int rank;  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
  
    const std::array<std::string, 3> phases {"alpha", "beta", "gamma"};  
    for (const auto &phase : phases) {  
        trace_in(phase);  
        do_fancy_physics(rank);  
        trace_out(phase);  
    }  
  
    MPI_Finalize();  
    return 0;  
}
```



Building/executing the code



1. Load modules:

- `module load oneapi intelmpi itac`

2. Launch the build:

- `mpiicpc -I$ITAC_HOME/itac/latest/include -L$ITAC_HOME/itac/latest/slib -lVTnull itac_test.cpp -o itac_test`

3. Run with ITAC tracing enabled:

- `mpirun -trace -np 2 ./itac_test`



Ungrouped output:

- `traceanalyzer itac_test.stf`

P0

alpha

beta

gamma

P1

alpha

beta

gamma

Code for reference:

```
#include <VT.h>
#include <cmath>
#include <iostream>
#include <map>
#include <mpi.h>
#include <string>

namespace {
    std::map<std::string, const int> itac_handles;
}

void trace_in(const std::string &trace) {
    if (itac_handles.count(trace) == 0) {
        int new_handle;
        VT_funcdef(trace.c_str(), VT_NOCLASS, &new_handle);
        itac_handles.insert({trace, new_handle});
    }

    VT_begin(itac_handles[trace]);
}

void trace_out(const std::string &trace) {
    VT_end(itac_handles[trace]);
}

void do_fancy_physics(const int rank) {
    float aggregator = 0;
    for (int i = 0; i < 100'000'000; ++i) {
        aggregator += sin(i);
    }
    std::cout << "Result on " << rank << ": " << aggregator << std::endl;
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const std::array<std::string, 3> phases{"alpha", "beta", "gamma"};
    for (const auto &phase : phases) {
        trace_in(phase);
        do_fancy_physics(rank);
        trace_out(phase);
    }

    MPI_Finalize();
    return 0;
}
```

Thank you!

- Feedback would really be appreciated:
<https://forms.office.com/r/YeFwCZ2kQi>
- Other training courses at ARC
- RSE support
- Join the Society of Research Software Engineering: <https://society-rse.org/join-us/>