Overview on testing parallel code & performance engineering

Melven Röhrig-Zöllner (SC-HPC)
Schedule

Part 1: Testing parallel code

• Levels of parallelism
• New “parallel” bugs
• Tools for specific bugs
• Unit tests
• Conclusion

Part 2: Performance engineering

• CPU architecture
• Performance modeling
• Performance “bugs”
• Finding bottlenecks
• Conclusion
Levels of parallelism: **SIMD**

- **SIMD:** *“single instruction, multiple data”*
- Also called SIMT ("*single instruction, multiple threads*") on GPUs

- **Example:** AVX-floating point unit of the CPU:
  (FMA operation calculates 4 double-precision fused multiply-add commands in one step)

  \[
  \begin{pmatrix}
  d_0 \\
  d_1 \\
  d_2 \\
  d_3 \\
  \end{pmatrix}
  \leftarrow
  \begin{pmatrix}
  a_0 \\
  a_1 \\
  a_2 \\
  a_3 \\
  \end{pmatrix}
  \cdot
  \begin{pmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  b_3 \\
  \end{pmatrix}
  +
  \begin{pmatrix}
  c_0 \\
  c_1 \\
  c_2 \\
  c_3 \\
  \end{pmatrix}
  \]

- **Requirement:** **Alignment of data** (pointer addresses must be a multiple of 32 bytes)
  - Handled by the compiler
  - Debugging only needed for hand-written SIMD code
  \[\Rightarrow\text{not further discussed here}\]

Levels of parallelism: **multi-threading**

- **Threads**: "lightweight processes"
  - Own execution stack
  - Shared data & resources (like files)

- Requires **synchronization**
  - to access shared data & exchange results
  - to access unique resources

- **Programming models**:
  - Work sharing
  - Tasked based
  - Master-worker / Thread-pool, …

- **Programming “languages”**:
  - Languages: C++11, Java, Python
  - Directives: **OpenMP** with C/C++/Fortran
  - Libraries: Qt (high-level), pthreads (low-level), …
Levels of parallelism: *multi-processing*

- Processes: “*individual execution contexts*”
  - Own execution stack & data
  - Shared OS environment

- Requires inter-process *communication*
  - Shared data (files, memory)
  - Message passing

- Programming models:
  - Server-client
  - SPMD (“*single program multiple data*”)
  - PGAS (“*partitioned global address space*”)

- Programming “languages”:
  - SPMD: MPI + C/C++/Fortran
  - PGAS: GASPI, C++Dash, Fortran’08
  - ...
New “parallel” bugs: **race conditions**

- Concurrent access to the same data element:
  - Read + write
  - Write + write

- Common pitfall for multi-threading

- **Non-deterministic** ⇒ difficult to reproduce & examine

- Another example TOCTTOU (“time of check to time of use”)
  - See programming challenge
  - Also possible over network (client-server scenario)
New “parallel” bugs: **deadlocks**

- **Circular blocking waiting:**
  - 2 or more threads / processes
  - waiting while blocking other resources

- Rare, but no easy recovery / avoidance

- **Non-deterministic** ⇒ difficult to reproduce & examine
Tools for specific bugs: compiler instrumentation

**Sanitizer** options for modern GCC and Clang
- For C/C++/Fortran on Linux
- Quite fast
- Need to recompile everything
- Readable output with debug symbols
- Open Source: [https://github.com/google/sanitizers/wiki](https://github.com/google/sanitizers/wiki)

Not specific to parallel programs:

- **Address** sanitizer:
  - Detects **invalid memory access**
  - Detects memory (de)allocation errors
  - Activated with `-fsanitize=address`
  - Crucial for low-level or parallel code

**Thread** sanitizer:
- Detects **race conditions** and **deadlocks** for multi-threaded programs
- Activated with `-fsanitize=thread`
- Possibly reports false positives

**Undefined behavior (UB)** sanitizer:
- Finds unexpected bugs
- UB: special cases with no guaranteed behavior
- Activated with `-fsanitize=undefined`
- Useful from time-to-time...
Tools for specific bugs: **valgrind**

- **Debugging tool**
  - For Linux
  - Extremely slow
  - Works with (almost) all executables
  - Readable output with debug symbols
  - Open Source: [http://valgrind.org/](http://valgrind.org/)

- **Helgrind (or DRD) tool:**
  - Detects **race conditions** and **deadlocks** for multi-threaded programs
  - Run with `valgrind -tool=helgrind <exe>`
  - Possibly reports false positives

Not specific to parallel programs:

- **Memcheck tool:**
  - Detects **invalid memory accesses**
  - Detects memory (de)allocation errors
  - Detects uninitialized data
  - Run with `valgrind --tool=memcheck <exe>`
  - **MPI-support** to detect MPI buffer errors (needs special compiler flags + LD_PRELOAD)
  - Sometimes reports false positives
  - Crucial when address sanitizer is no option

- **Performance tools** (**cachegrind**, etc.):
  - Not so useful as the hardware is emulated…
Tools for specific bugs: **must**

- **MPI communication checker**
  - Detects MPI usage errors
  - Detects deadlocks with MPI
  - Will detect data races with one-sided communication in MPI
  - Run program with `mustrun -np <n> <exe>`
    (instead of `mpirun -np <n> <exe>`)  
  - Open Source: [https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST](https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST)
Unit tests: problems from the wild (1)

• Setup:
  • parallel unit tests with
  • 2 processes
  • Output on process 0

• Same error on all processes

⇒ Error reported correctly
Unit tests: problems from the wild (2)

- Setup:
  - parallel unit tests with
  - 2 processes
  - Output on process 0

- Error only on process 1

⇒ Error not reported!
Unit tests: **problems from the wild (3)**

- **Setup:**
  - parallel unit tests with
  - 2 processes
  - Output on all processes

- Error only on process 1

⇒ **Multiple processes write into the same file!**
Unit tests: **problems from the wild (4)**

- **Setup:**
  - parallel unit tests with
  - 2 processes
  - Output on process 0

- Error only on process 1, process 0 waiting

⇒ **No output & program does not terminate!**
Unit tests: our solution

• Setup:
  • parallel unit tests with
  • 2 processes
  • Global assertions and output

• Error only on process 1

⇒ Error reported correctly, program terminates!
Unit tests: frameworks

- For C/C++: googletest+MPI
  - Thread-safe, but no multi-threading functions
  - MPI support from SC-HPC: [https://gitlab-ee.sc.dlr.de/HPC/googletest_mpi](https://gitlab-ee.sc.dlr.de/HPC/googletest_mpi)
  - Open Source (no MPI): [https://github.com/google/googletest](https://github.com/google/googletest)

- For C/C++: Trilinos package Teuchos
  - Tools package of Trilinos
  - Large library for scientific computing
  - Open Source: [https://trilinos.org](https://trilinos.org)

- For Fortran: pFUnit
  - Supports OpenMP and MPI
  - Developed by the NASA

- For Java: (JUnit??)

- Others???
Unit tests: **test setup**

- To detect (all important) bugs:
  - Run tests with different tools
  - Vary number of threads / processes

  ⇒ Drawback: exploding number of combinations

- Limited time / resources:
  - Automation with CI (e.g. Jenkins)
  - Start with simple tests (1 process/thread)
  - Combine tests for “orthogonal” problems
Conclusion

• Parallel code is complex & **non-deterministic**:  
  • Multiple levels of parallelism  
  • Different programming models  
  ⇒ New **parallel bugs** (data races, deadlocks)

• **Tool support** is crucial:  
  • Problems not easy to reproduce (in debugger)  
  • Tools can help to detect bugs  
  ⇒ Choose correct tool(s) for your use case.

• **Parallel unit tests:**  
  • Serial frameworks may lead to more problems.  
  ⇒ Tests should support the desired parallelism.
  • Test setup (combine tools and #threads/procs)

• Not handled:  
  • more subtle errors like starvation  
  • differing results through non-ordered operations
Schedule

Part 1: Testing parallel code

• Levels of parallelism
• New “parallel” bugs
• Tools for specific bugs
• Unit tests
• Conclusion

Part 2: Performance engineering

• CPU architecture
• Performance modeling
• Performance “bugs”
• Finding bottlenecks
• Conclusion
CPU architecture: computing units

- Intel “Skylake” Gold (SC HPDA cluster) core:
  - 2 FMA (fused multiply-add) units
  - **SIMD** width: 512 bit (e.g. AVX512):
    fits 16 single or 8 double precision numbers
  - **Latency**: 4 cycles (FMA/add/sub/mul)
  - Other operations (div, sqrt) are much slower

⇒ **Need lots of independent “multiply-additions”**
  (e.g. 128 to fill the pipeline of 1 core)
CPU architecture: memory hierarchy

- Intel “Skylake” Gold (SC HPDA cluster) socket:
  - 14 cores per socket
  - 3 cache levels with:
    - L1 cache (32kB, 4 cycles latency)
    - L2 cache (1MB, 14 cycles latency)
    - L3 shared cache (19MB, >50 cycles latency)
  - “Slow” main memory
    (94GB per socket, >400 cycles latency)
  - Caches organized in lines of 64 bytes and optimized for “streaming accesses”

⇒ Need lots of contiguous accesses to a small data set
Performance modeling: roofline

- The **roofline model**
  - applicable **peak performance**: \( P_{\text{max}} \left[ \frac{\text{Flop}}{s} \right] \)
    (of the required operations)
  - computational intensity: \( I \left[ \frac{\text{Flop}}{\text{byte}} \right] \)
    (“work” per byte transferred of the algorithm)
  - applicable **peak bandwidth**: \( b_s \left[ \frac{\text{byte}}{s} \right] \)
    (of the slowest data path utilized)
  - Expected performance: \( P = \min(P_{\text{max}}, I \cdot b_s) \)

⇒ A lot of problems are **memory-bound**
(nice hack: we can do more operations for free)
Performance modeling: workflow

1. Analyze algorithm:
   - Calculate computational intensity
   - Estimate working set size (does it fit into L3?)

2. Benchmark
   - Select relevant operations (FMA or pure add?)
   - Calculate peak performance (CPU family specific)
   - Measure peak bandwidth (system specific)

⇒ Goal: Hit the right bottleneck!
   (and publish that your code is as fast as it gets)

General remarks:
   - works well for “simple” computational kernels
   - assumes the problem is big/parallel enough
   - Predictions are almost 100% accurate for large contiguous main memory accesses
   - Non-contiguous accesses have overhead (e.g. consider cache lines and cache misses)
   - It’s hard to tune code to obtain ≥ 10% peak...
Performance modeling: automated in ESSEX

- Generic interface for a roofline model (#ops, #bytes, relevant benchmark)
- All kernel functions are modeled (just provide #ops, etc)
- Small benchmark gathers data on startup
- “realistic” variant models strided data accesses & cache line size

⇒ Summary with biggest deviations:

<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>total time</th>
<th>%roofline</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,*iflag=0)</td>
<td>6.156e+00</td>
<td>11.7</td>
<td>174</td>
</tr>
<tr>
<td>STREAM_TRIAD((nV+nW)<em>n</em>sizeof(<em>ST</em>))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PHIST PerfCheck: Anasazi BKS with $n_b = 4$ gives lines like this:

<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>total time</th>
<th>%roofline</th>
</tr>
</thead>
<tbody>
<tr>
<td>phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,dV=85,*iflag=0)</td>
<td>6.013e+00</td>
<td>23.8</td>
</tr>
<tr>
<td>STREAM_TRIAD((nV+nW)<em>n</em>sizeof(<em>ST</em>))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance “bugs”: false sharing

- Scenario:
  - Cache line modified by threads on multiple cores (e.g. different elements in a small chunk of 64b)
  - System must guarantee cache coherence
  - Code completely correct – no data race, etc.

  ⇒ Behavior:
  Cache line written to main memory and reloaded

- Mitigation:
  - Work on **local data** where possible
  - Avoid array[nThreads], add **padding** to 64b (e.g. in C: `double array[8][nThreads];`)
Performance “bugs”: NUMA effects

• NUMA (non-unified memory access):
  • Faster/slower access to different memory parts
  • Systems with multiple CPU sockets
    (each socket has its own memory banks)
  • Some AMD CPUs
    (NUMA in a single socket)

• Mitigation:
  1. Pinning: bind processes and threads to cores
  2. First-touch policy: memory belongs to the
     NUMA domain that uses it first. (not trivial!)
Finding bottlenecks: measuring with ScoreP (1)

• Tool to measure performance:
  • Compiler wrapper for C/C++/Fortran
  • Nice and easy-to-use
  • Supports multi-threading & -processing (OpenMP and MPI)
  • Useful for a fast & rough overview
• Open Source:
  http://www.vi-hps.org/projects/score-p/
• Basis for more advanced tools: Scalasca, Vampir …
Finding bottlenecks: measuring with Scorep (2)

- **Workflow:**
  - Instrument compiler with ScoreP wrapper (e.g. CXX=scorep-g++ cmake <path>)
  - Run test case
  - Investigate measurement overhead (using scorep-score)
  - Filter out small functions (SCOREP_FILTERING_FILE, simple text format)
  - Rerun test case…

  ⇒ **Ensure same runtime as without ScoreP**

- **Hardware counters:**
  - CPU measures itself!
  - Available in ScoreP through PAPI
  - Open Source: [http://icl.utk.edu/papi/](http://icl.utk.edu/papi/)
  - Real run-time data per function about Operations, cache accesses, …
  - Interesting points:
    - Vectorized (SIMD) vs. non-SIMD FP ops
    - Achieved memory bandwidth
    - Cache misses
  - However: not all CPUs provide correct results (tool will usually not provide counters then)
Conclusion

- Know your architecture:
  - **SIMD** operations
  - Memory / **cache hierarchy**
    ⇒ Ideally: lots of similar operations on small data

- Setup a model:
  - Simple model of algorithm + hardware
  - Compare actual & predicted runtime
    ⇒ Goal: **hit the right bottleneck**

- Avoid pitfalls like false sharing, NUMA, …
  - Use tools for timings and hardware counters
  - Read a book:
    Hager & Wellein: “Introduction to High Performance Computing for Scientists and Engineers”, 2018

- Practical observations:
  - Optimized vs. normal code: factor >100
  - Problems: vectorization, temporary objects, …
References

