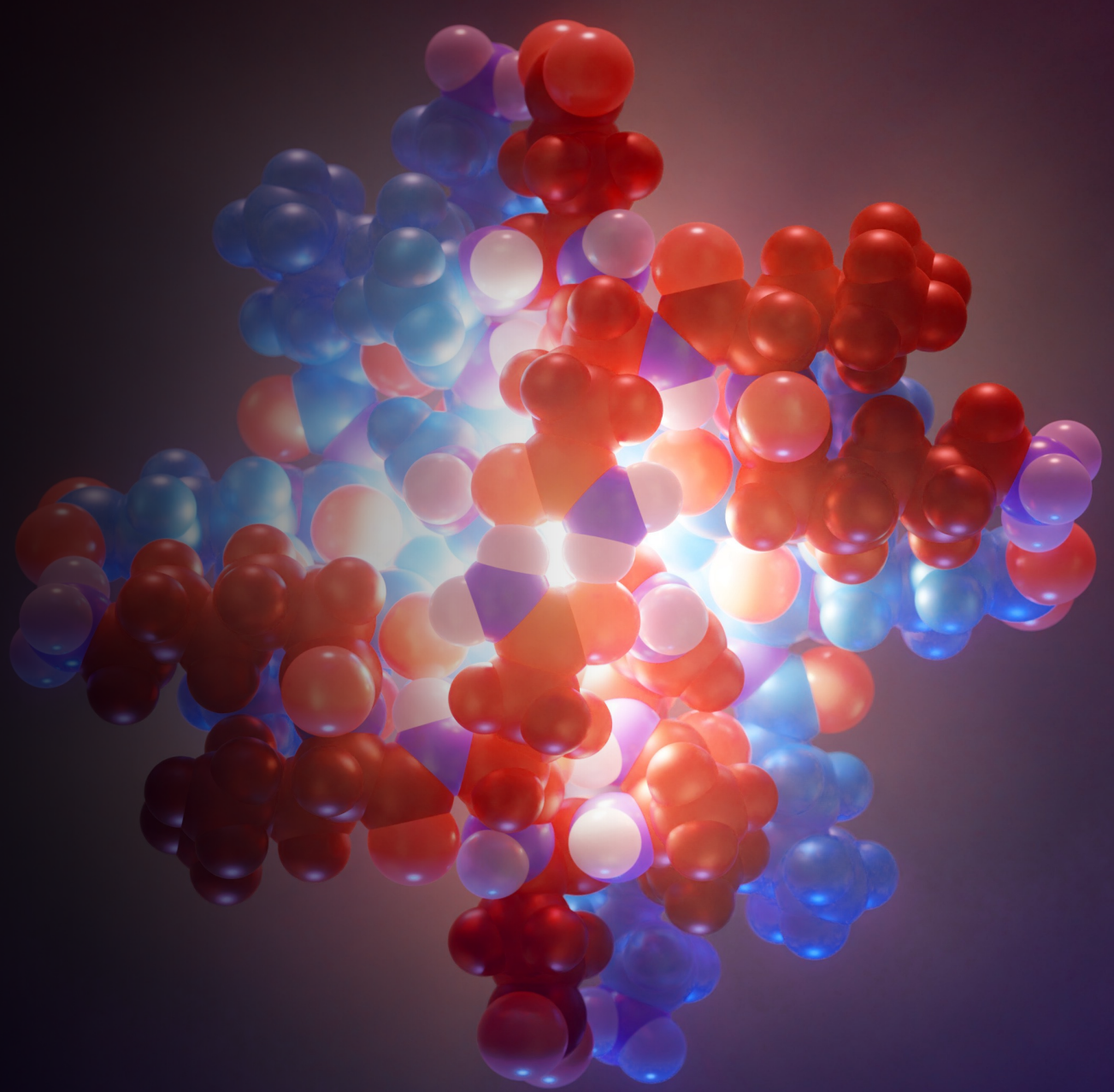
Tests, benchmarks, and parallelism: Lessons from Rosetta and Masala

Vikram K. Mulligan, Ph.D.
Research Scientist, Biomolecular Design Group
Center for Computational Biology
Flatiron Institute

Wednesday, 14 June 2023



Disclaimer

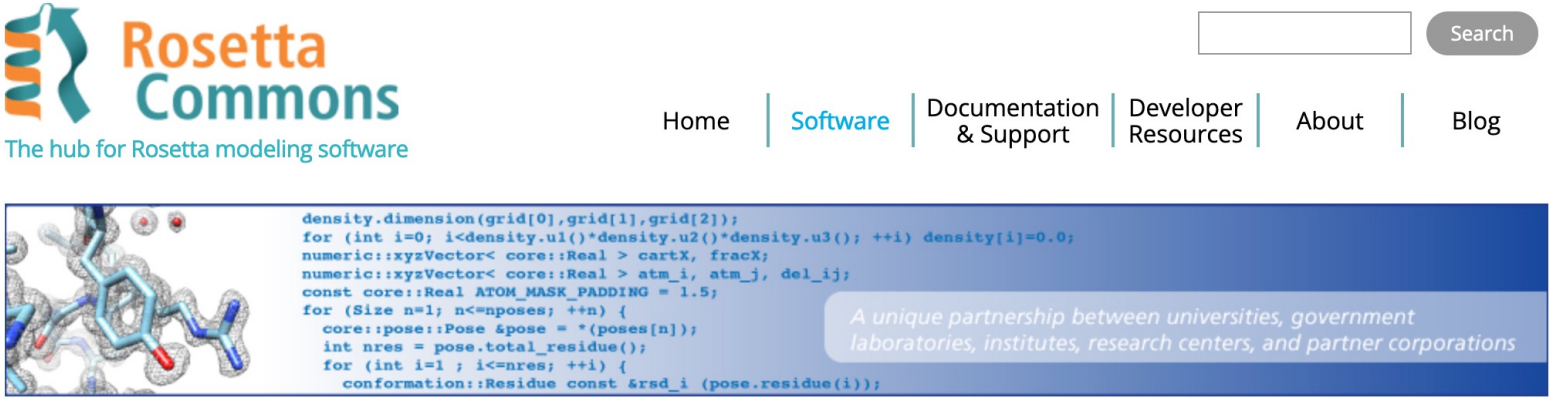
Organization of this talk

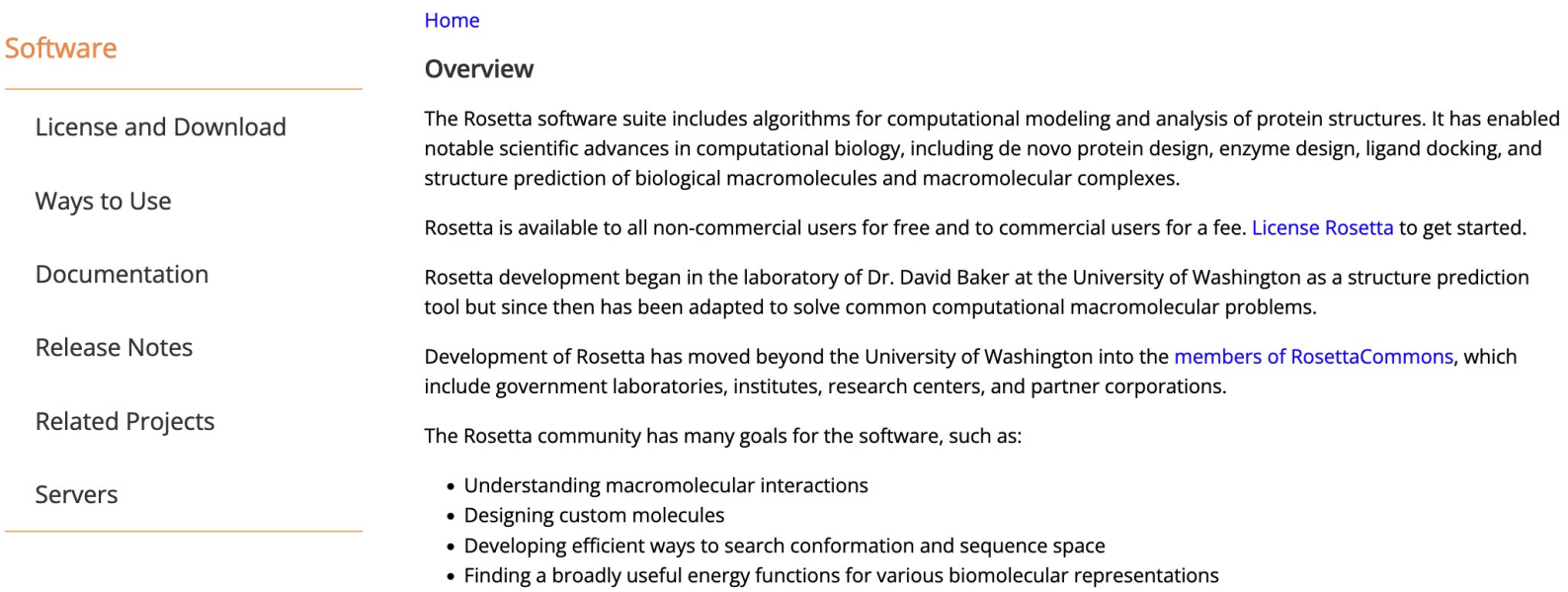
1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

Organization of this talk

1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

What are Rosetta and Masala (and why are you hearing about them)?





- Rosetta is protein modelling software that has been generalized for more exotic macromolecules.
- The software is free for academics, nonprofits, and governments, and is licenced for a fee for commercial use.
- Originally started in David Baker's lab, Rosetta is now developed and maintained by more than 70 labs in many countries.

What are Rosetta and Masala (and why are you hearing about them)?

- Masala is a free and open-source successor to Rosetta under development at the Flatiron institute.
- It is structured to take full advantage of modern massively-parallel CPU and GPU hardware.
- It has a versatile plugin architecture permitting easy extensibility.
- It is intended to be used as standalone software *or* as a library in other projects. (Rosetta, for instance, can link Masala for high-efficiency design calculations.)

Organization of this talk

1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

The Rosetta development community

- Rosetta was originally developed in the late 1990s in the Baker laboratory at the University of Washington.

- The original Fortran version was auto-converted to C++ (Rosetta 2.0) in the early 2000s, then manually rewritten in C++ (with sensible classes and modern architecture) around 2010.

- Many of David Baker's former postdocs and students have started labs of their own that have continued Rosetta development. Today, about 70 labs in several countries contribute to Rosetta.

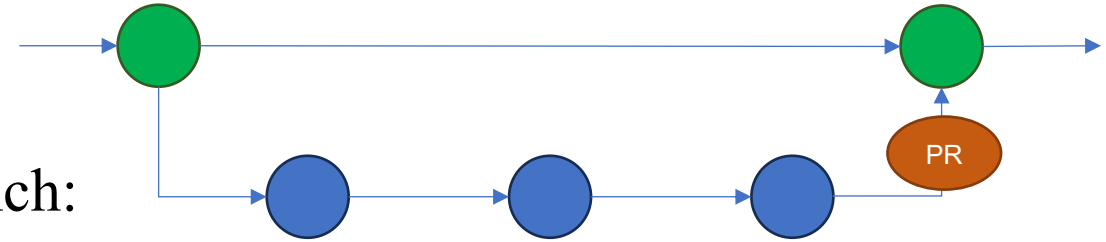
- Many Rosetta developers are biologists with limited C++ programming experience who make small contributions to the codebase.

- A small handful of senior developers are more experienced with software engineering (though many of these have left the community recently).

The Rosetta development cycle

Master branch:

Development branch:



RosettaCommons / main Private

Unwatch 227 Fork 2 Starred 58

Code Issues 414 Pull requests 136 Actions Projects 22 Wiki Security Insights

Add better MPI support for EnsembleMetrics #5855

Edit Code

Open **vmullig** wants to merge 257 commits into `master` from `vmullig/ensemble_metrics_mpi_support`

Conversation 0 Commits 250 Checks 0 Files changed 157 +32,622 −57

vmullig commented on Feb 19, 2022 • edited

One current limitation of EnsembleMetrics is that there's no good way to generate an ensemble of structures in a distributed fashion and perform analysis on the whole ensemble without writing everything out to disk and doing the analysis in a subsequent step. (I did add support for ensembles generated in parallel threads in pull request #5838, but this is experimental, and revealed some more thread-safety issues that must be resolved before multi-threaded ensemble generation is ready for production runs.) This PR aims to address this limitation by adding support for automatically collecting the results of the sampling from many MPI-distributed instances of an ensemble metric so that the process 0 instance can perform the analysis, in the context of JD2 RosettaScripts or other JD2 apps that can use the JD2 MPI job distributor. This can be generalized in a subsequent PR for JD3 and other MPI communication schemes.

Note that allowing modules to make MPI calls is fraught. This PR constrains the type and location of MPI calls to two functions, and ensures that the job distributor guarantees synchronicity to avoid deadlock.

- [ ] Pull request Add EnsembleMetrics for measuring properties of an ensemble of poses #5824 must be merged before this one.

Tasks:

- [x] Add `EnsembleMetric::supports_mpi()`, `EnsembleMetric::send_mpi_summary()`, and `EnsembleMetric::recv_mpi_summary()` to base class.
  - [x] Implement these for `CentralTendencyEnsembleMetric`.
  - [x] Add these to the code templates.
  - [x] Account for fact that instance on master MPI process may be uninitialized.
  - [x] Better `static_asserts` that `double == core::Real` and `unsigned long == core::Size`, assumptions needed for MPI communication.
- [x] Ensure that these functions are called *only* at the end of a JD2 MPI protocol, and *only* for those EnsembleMetrics that are configured to report on all poses seen in the protocol (and which aren't used on internally-generated ensembles or ensembles generated by a multiple pose mover).

Reviewers
- weitzner
- everyday847
- dougrenfrew
- jadolfbr
- jkleman
- roccomoretti

At least 1 approving review is required to merge this pull request.

Still in progress? Learn about draft PRs

Assignees
- vmullig

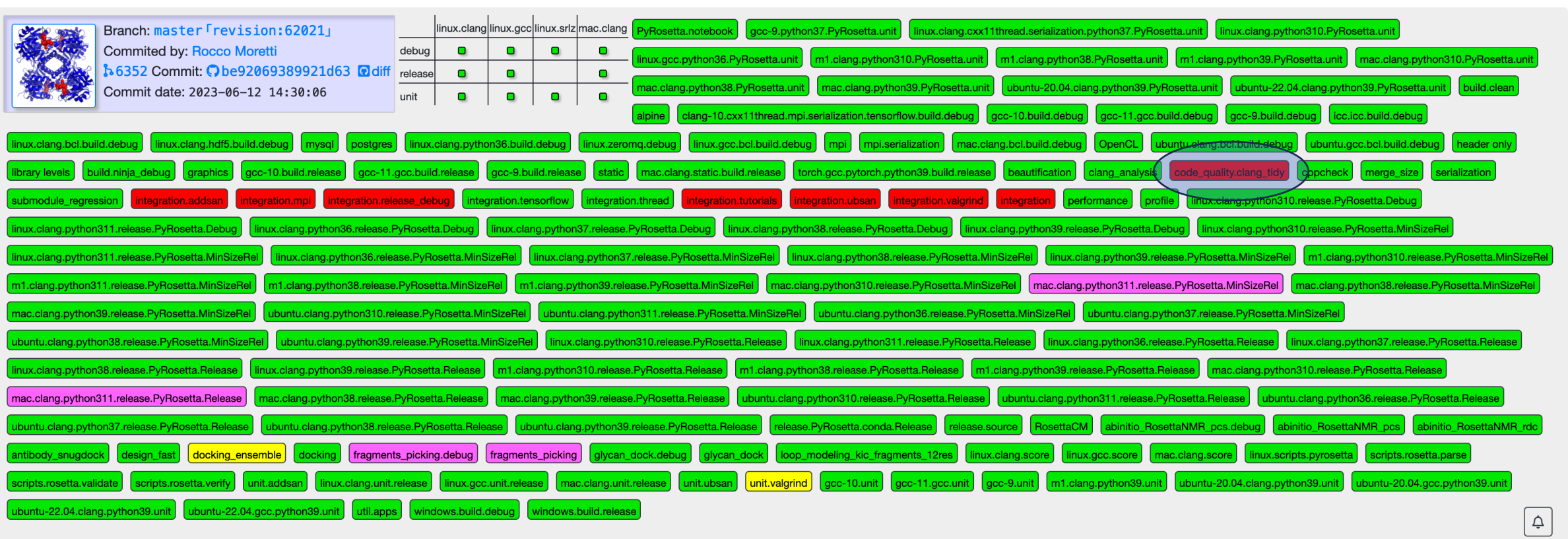
Labels
enhancement MPI ready for review

Projects
Modernize RosettaScripts
In progress

Organization of this talk

1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

The Rosetta testing server



https://b3.graylab.jhu.edu/revisions/master

The Rosetta testing server

previous revision

「note that this is test page of B3 βῆτα server, so if you experience any issues with it please report it on Slack and visit B2 page instead」

find code_quality.clang_tidy tests

login to queue/cancel/re-run this test

Test: **ubuntu.clang.code_quality.clang_tidy**
Branch: master「revision:62021」
Test files: 「 file-tree-view 」 「 file-list-view 」 「 test log 」 「 JSON output 」
Daemon: nobu-3 Runtime: 31m 31s State: failed
Failed sub-tests: src/apps/public/loop_modeling/transform_loodo.cc src/protocols/ptm_prediction/PTMPredictionMetric.cc src/protocols/ptm_prediction/PTMPredictionTensorflowProtocolBase.cc src/protocols/ptm_prediction/PTMPredictionTensorflowProtocol.cc

test log

```
Clang tidy with `-checks=clang-diagnostic-*,clang-analyzer-*,bugprone-*,misc-*,modernize-use-nullptr,modernize-use-override,-bugprone-forward-declaration-namespace,-misc-unconventional-assign-op

Clang Tidy Version:
LLVM (http://llvm.org/):
  LLVM version 6.0.0

  Optimized build.
  Default target: x86_64-pc-linux-gnu
  Host CPU: skylake-avx512
```

failed sub-tests

src/apps/public/loop_modeling/transform_loodo.cc

```
/home/benchmark/rosetta/source/cmake/build_clang_tidy/../../src/numeric/xyzMatrix.hh:564:8: warning: Assigned value is garbage or undefined [clang-analyzer-core.uninitialized.Assign]
                      xx_ = m.xx_; xy_ = m.xy_; xz_ = m.xz_;
                          ^
/home/benchmark/rosetta/source/src/apps/public/loop_modeling/transform_loodo.cc:112:3: note: Taking false branch
                if ( !stream ) {
                ^
/home/benchmark/rosetta/source/src/apps/public/loop_modeling/transform_loodo.cc:118:3: note: Loop condition is true.  Entering loop body
                while ( getline(stream,line) ) {
                ^
/home/benchmark/rosetta/source/src/apps/public/loop_modeling/transform_loodo.cc:124:4: note: Taking false branch
                        if ( stub_index == std::string::npos ) {
                        ^
/home/benchmark/rosetta/source/src/apps/public/loop_modeling/transform_loodo.cc:127:4: note: Taking false branch
                        if ( name_index == std::string::npos ) {
                        ^
```

https://b3.graylab.jhu.edu/revisions/master

Types of tests that Rosetta developers write: unit tests

- Unit tests test whether a particular module works correctly in isolation.
- In the Rosetta unit test suite, unit tests are typically set up as pass/fail tests that check to see whether a function returns a known, “right” answer.
- These are generally expected to have low computational cost.

```
Pose pose;
make_pose_from_sequence( pose, "DGGGEKGG", "fa_standard");
TS_ASSERT_EQUALS(pose.residue_type(1).net_formal_charge(), -1);
TS_ASSERT_EQUALS(pose.residue_type(2).net_formal_charge(),  0);
TS_ASSERT_EQUALS(pose.residue_type(5).net_formal_charge(), -1);
TS_ASSERT_EQUALS(pose.residue_type(6).net_formal_charge(),  1);
```

Types of tests that Rosetta developers write: integration tests

- Integration tests test whether several modules work properly together.
- In the Rosetta test suite, integration tests are regression tests (testing whether behaviour has *changed* from one revision of the software to another). This need not be true for all software.
- In the Rosetta test suite, integration tests are typically fast runs of a simplified version of a full protocol (*e.g.* a simple design script, a quick docking run, *etc*.)

Types of tests that Rosetta developers write: scientific tests and performance benchmarks

- These are more computationally expensive tests that carry out full protocols to benchmark performance and/or test scientific accuracy against some gold standard or prior knowledge.

- Unlike unit and integration tests, which run on every revision of Rosetta, scientific tests are run on a best-effort basis.

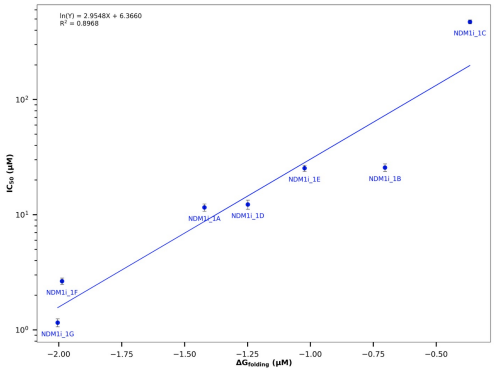
Scientific test: peptide_pnear_vs_ic50

FAILURES

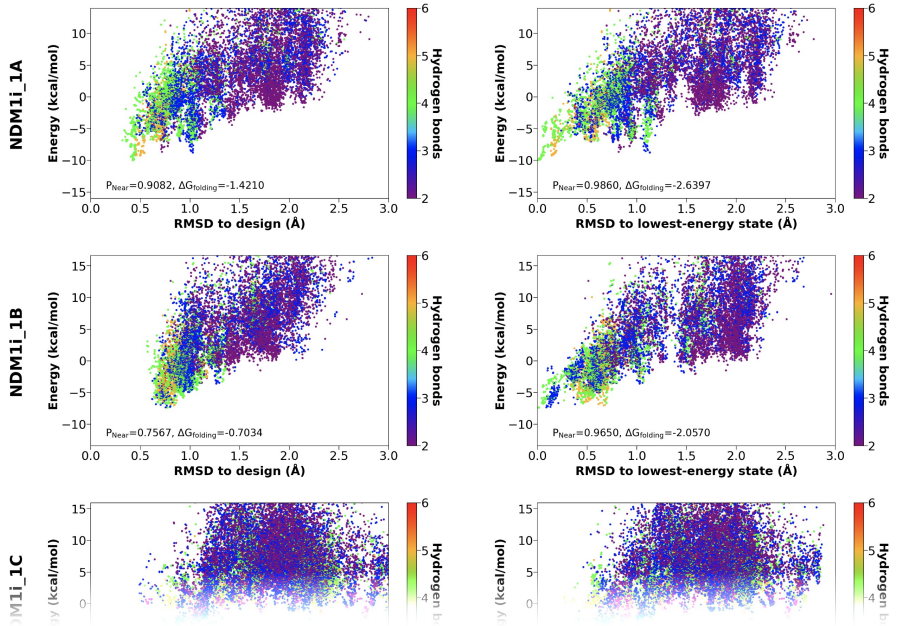
None.

RESULTS SUMMARY

Analysis of correlation of predicted folding propensity with experimentally-measured inhibition values (which should be linear):



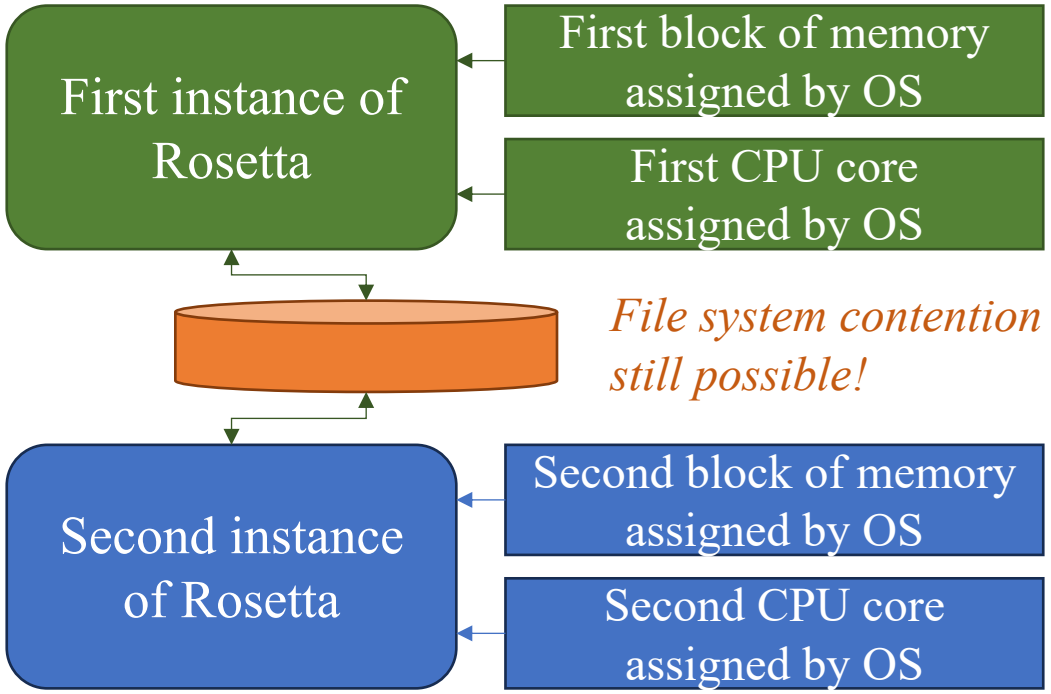
Folding funnels:



Organization of this talk

1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

Types of parallelism in Rosetta: Embarrassing parallelism



The user launches entirely independent processes, each doing entirely separate work.

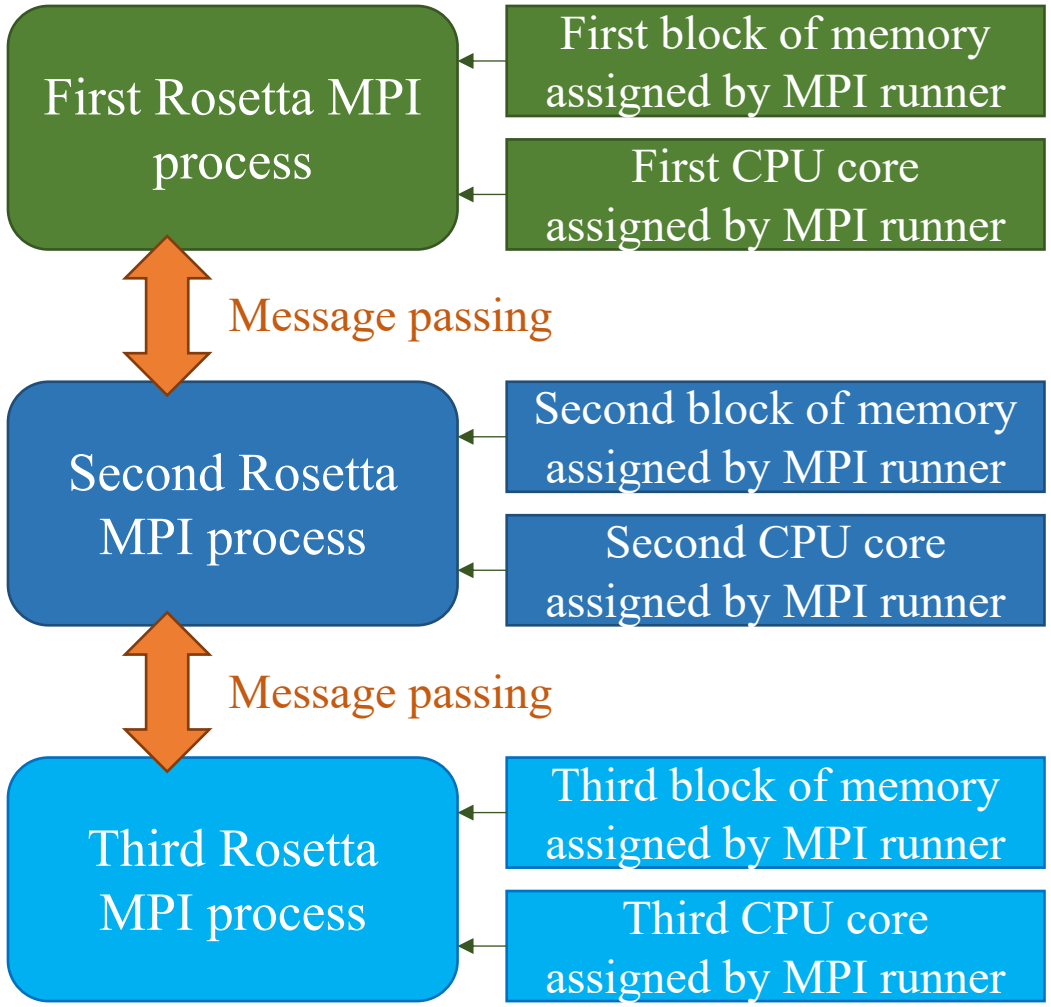
Advantages

- Conceptually simple.
- Typically, little to no resource contention for memory or cores.

Disadvantages

- No load-balancing.
- Can have hardware contention for I/O devices (particularly the file system).
- Memory wasteful if each instance must load the same data.

Types of parallelism in Rosetta: Message Passing Interface (MPI) based process-level parallelism



The user uses an MPI launcher to launch many parallel processes that can communicate through *message passing*.

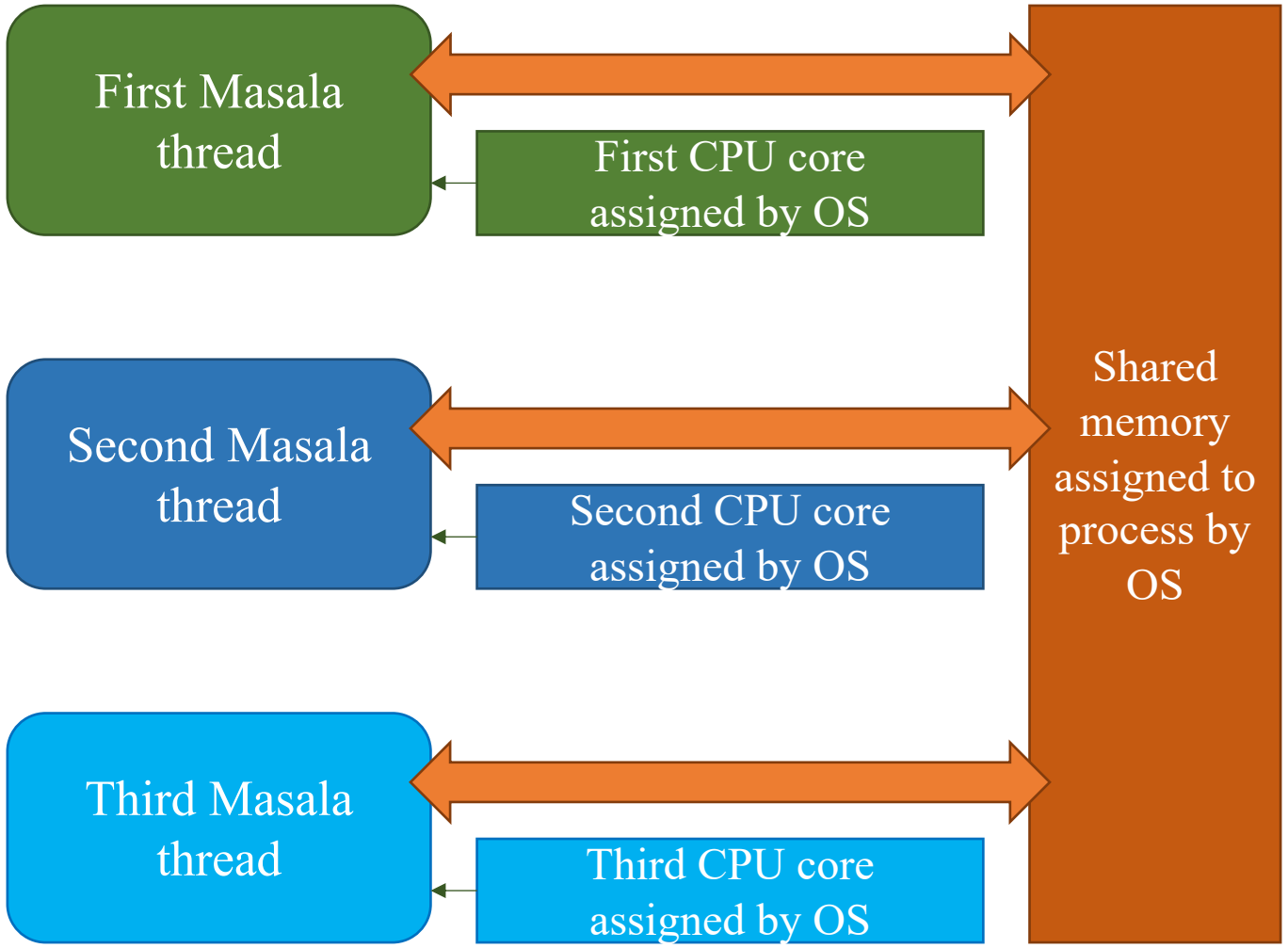
Advantages

- Little/no resource contention for memory or cores.
- Development model requires no complicated mutex locking.
- Can achieve load-balancing.
- Can minimize disk contention with I/O nodes.
- Works across all cores on large clusters.

Disadvantages

- Memory wasteful if each process must load the same data.
- Deadlock is a danger.
- Lots of void pointers: little type safety for objects passed between processes, facilitating dev error.

Types of parallelism in Rosetta and Masala: POSIX thread-based shared-memory parallelism



The user launches one instance of the program which internally launches multiple *threads* that share memory.

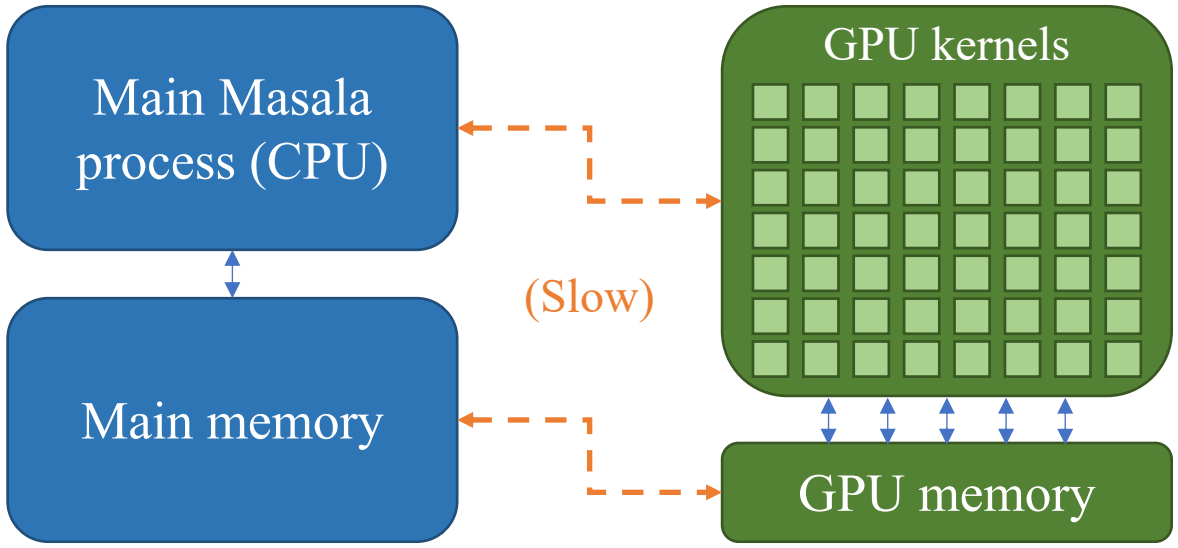
Advantages

- Can achieve load-balancing.
- Can have dedicated I/O threads to avoid disk contention.
- Much lower memory usage since memory is shared amongst threads.
- Allows more finely-grained parallelism.

Disadvantages

- Resource contention for memory.
- Requires mutexes.
- Compiler optimization complicates.
- Bugs hard to diagnose.
- Performance issues hard to diagnose.
- Works only *within* a compute node.

Types of parallelism planned for Masala: GPU-based parallelism



The user launches one instance of the program which internally launches kernels to run on the GPU. Each GPU core is slow, but there are many.

Advantages

- Allows most finely-grained parallelism.
- Can enormously accelerate those tasks that can be divided into thousands of small parts.

Disadvantages

- Contention for GPU memory must be managed.
- Transfer of instructions and data from main memory to the GPU is slow (*i.e.* there is a startup cost to GPU tasks).
- Programming model is complex and often hardware-specific (*e.g.* CUDA). (Some cross-platform APIs – OpenCL, SYCL, *etc.*)

Organization of this talk

1. What are Rosetta and Masala?
2. How has Rosetta been developed, and what challenges are presented by this development model?
3. How is Rosetta tested?
4. What types of parallelism do we have in Rosetta and Masala?
5. What are the lessons learned from parallel work in these software packages?

Challenges of testing, profiling, and debugging parallel code

- Errors sometimes only appear at scale (*e.g.* when thousands of processes are trying to communicate *via* MPI).
- Errors are often stochastic (dependent on the relative speed with which different processes or threads complete their work).
- Testing needs to be performed in debug and release mode. Compiler optimizations can introduce bugs, especially in multithreaded code.
- Performance needs to be tested in release mode. Linear scaling in debug mode can disappear after compiler optimization.
- Performance testing can be complicated by CPUs' low-energy mode when few threads are running, or by auto-vectorization and other compiler optimizations.
- Performance issues are hard to debug. (Profilers and analysis tools like Gprof or Valgrind can help, but it may not be obvious what the bottleneck is.)
- Debugging parallel code is still more of an art than an exact science. Even highly experienced developers struggle with this.

Common parallel computing problems (*i.e.* things to test for!)

- Disk I/O is a common killer of performance. Ideally, reads from disk should be lazy, done by one parallel process or thread, threadsafe, and one-time-only. (The "delete the database" test helps with this.)
- With MPI, data corruption during transmission is a common issue. MPI_Send and MPI_Recv operations need to be tightly coupled, and bits must be interpreted the same way!
- Deadlock is another common issue with MPI. Consider edge cases carefully to make sure that you never have a receiver waiting forever for a message that's never sent, or two processes both waiting for messages from one another.
- With threads, common (hard-to-detect) bugs are simultaneous reads and writes of the same data in memory, simultaneous writes of the same data in memory, or accidentally unsafe writes due to compiler optimization. *Memory fences* and *mutexes* are important.
- With threads, common killers of performance are unnecessary mutex locking, read-write mutexes (which are rarely worth it), and smart pointers.

Debugging threaded code with GDB (or LLDB)

In general, multithreaded programs can be debugged in exactly the same way as single-threaded programs using GDB (the GNU debugger) or LLDB (the LLVM/Clang debugger). The only additional command is the **t** command to switch threads.

catch throw (or **break set –E C++**): Stop when an exception is thrown, by any thread.

t 5: Switch to thread 5.

bt: Get a backtrace (a list of what was calling what when this thread halted).

f 3: Go to the third *frame* of the backtrace (the grandparent of the thing being called).

list: Print the lines of code around the point at which we halted.

print myvar: Interrogate the value of the variable called “myvar” in the current frame.

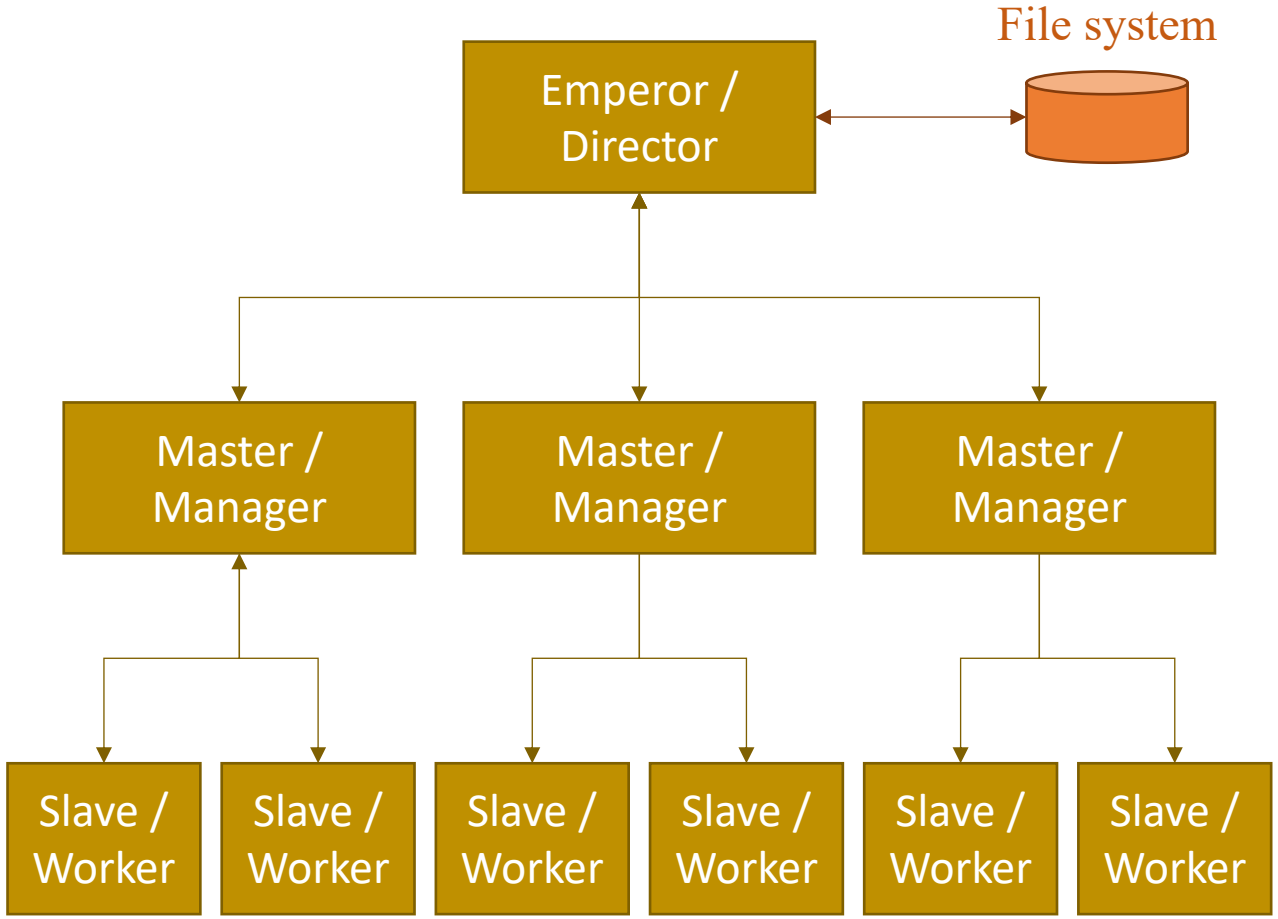
Debugging MPI code with GDB (or LLDB)

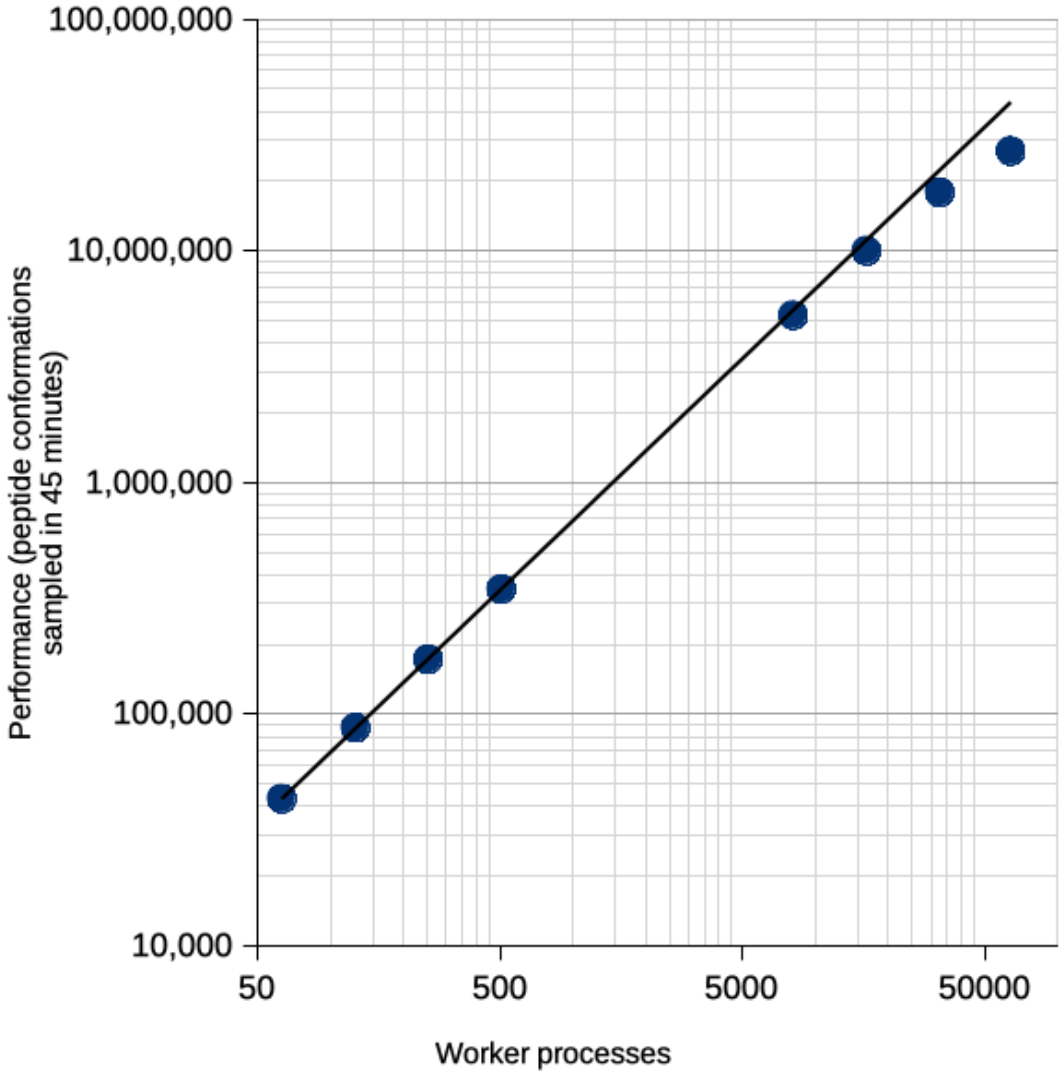
Multi-process code is harder to debug. Typically, one must launch MPI processes with **mpirun**, **srun**, or **aprun**, and then attach GDB to a running process by launching GDB and using the **attach <PID>** command to attach to a given process ID. One common trick is to add the following code early in one's MPI program:

```
if( mpi_rank == 2 ) { // Assuming we want to debug MPI process with rank 2
     int i = 1;
     while( i == 1 ) { sleep(3); /*loop forever*/ }
}
```

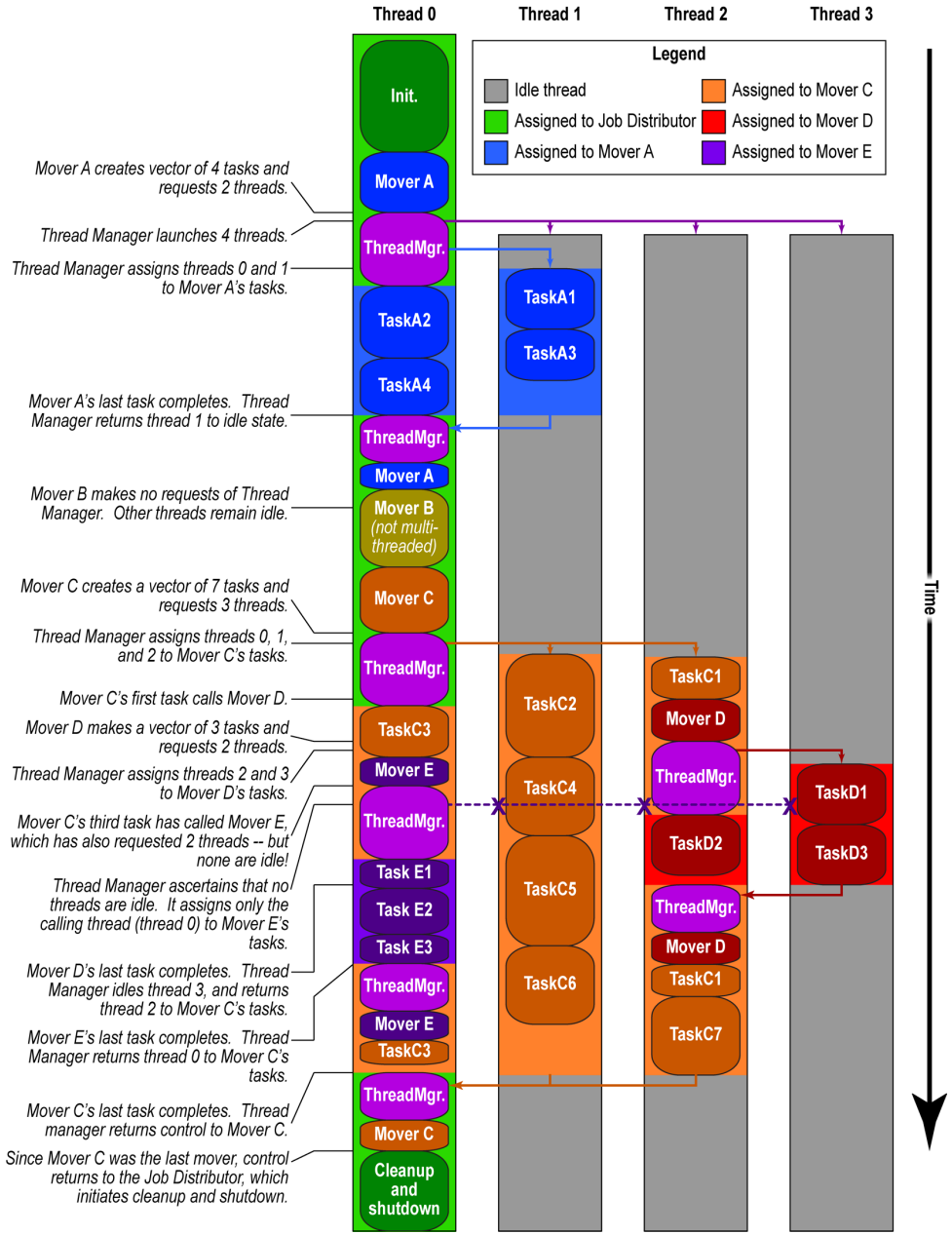
We launch the MPI program, then attach GDB to the second MPI process. We can then press Ctrl+C to halt the program, add our **catch throw**, then type **set var i = 2** and **continue** to break out of the loop and begin debugging.

Achieving near-linear performance scaling in Rosetta's simple_cycpep_predict application with MPI





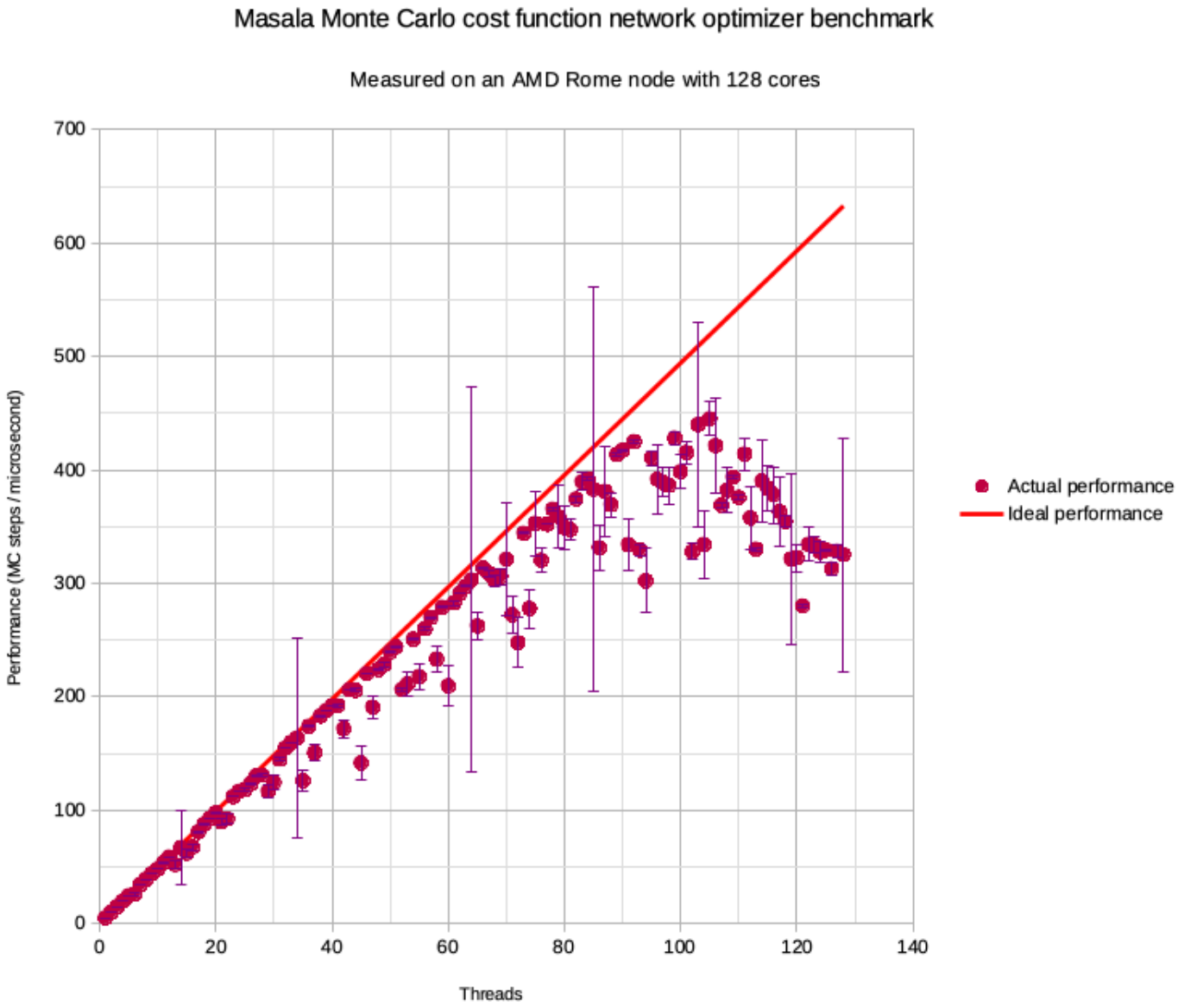
Minimizing overhead of starting threaded work: Use of thread pools in Rosetta and Masala



- Thread pools keep threads in existence (but idle), waiting for work. This saves the overhead of creating and destroying threads for new work.
- Rosetta's thread manager allows efficient *hierarchical* assignment of work to available threads in the thread pool.
- Masala's thread manager improves on this slightly, allowing threads to be released for new tasks when their own work is done (rather than when the block of work is done).
- Certain multi-threading APIs like OpenMP may offer thread pool functionality. (Or, if your code is GPL-licenced, you could re-use Masala's.)

Achieving near-linear performance scaling in Masala's cost function network optimizer with POSIX threads

- Executing independent simulated annealing trajectories in threads that share memory.
- *Minimal* mutex locking for write in the threaded work. (One lock at the end of each trajectory to write final results to a shared memory object.)
- *No* mutex locking for read in the threaded work. (Shared-memory data structures have a setup phase in which mutexes are locked and a read-only phase in which multiple threads can read without locks.)
- *No* passing of shared pointers in threaded work (since incrementation of reference counts results in mutex locks that kill threaded performance).



Acknowledgements

Thank you to:

- Andrew Leaver-Fay
- Yuri Alexeev
- Sergey Lyskov
- Alexander Ford
- Andrew Watkins