Abstract

Most research software is written by researchers without professional programming training, often causing subtle but costly performance pitfalls. Small choices “like inefficient data structures”can greatly increase execution time. Even skilled developers face challenges, as language- and library-specific nuances are often learned informally. Slow software reduces a researcher’s productivity and wastes energy and shared resources (e.g., HPC systems), affecting others’ work as well.

SIG-RPC builds community around research, development, and advocacy for software performance best practices. We curate a knowledge base of common performance traps and profiling tools, helping researchers and developers write faster, more efficient, and more sustainable code.

sig-rpc.github.io

SIG Reasonable Performance Computing

A community dedicated to the research, development and advocacy of performance best practices.

Robert Chisholm

Supported by

- Society of Research Software Engineering
- Software Sustainability Institute
- University of Sheffield RSE

Typical research code author

- Domain expert
  - They know lots about their field
- Self-taught programmers
  - Bad habits
- Short of time
  - Just need the code to work
- Often working with inherited code
  - May not be familiar with entire codebase

This leads to coding traps

- Often unnoticed in code that's in use for many years by many users
- Common traps found across vastly different projects
- Potentially unreasonably poor performance

Managing uniques with an array

```
list_out = []
while len(list_out) < 1000:
    t = random.randint(0, 2000)
    if not t in list_out:
        list_out.append(t)
```

Rather than a set

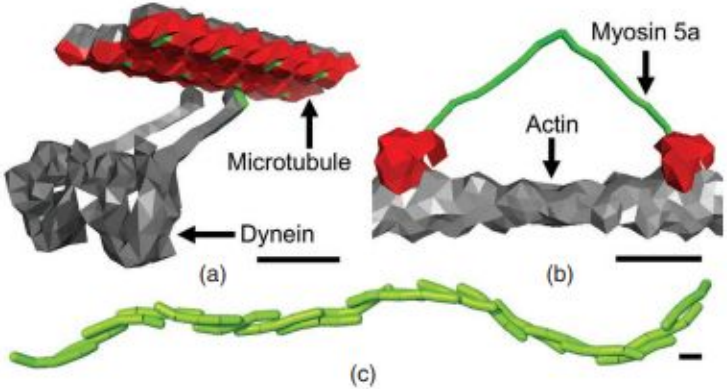
```
set_out = {}
while len(set_out) < 1000:
    t = random.randint(0, 2000)
    set_out.add(t)
```

Most common mistake.
With enough data 1000x speedup possible.

FFEA - A recent case study (26th August)

- Fluctuating Finite Element Analysis (Molecular Modelling Software)
- C++ & OpenMP
- Development began 2010
- 9 authors
  - PhD Students
  - Postdoc Research Associates
- ~12 publications

Representative of much research software



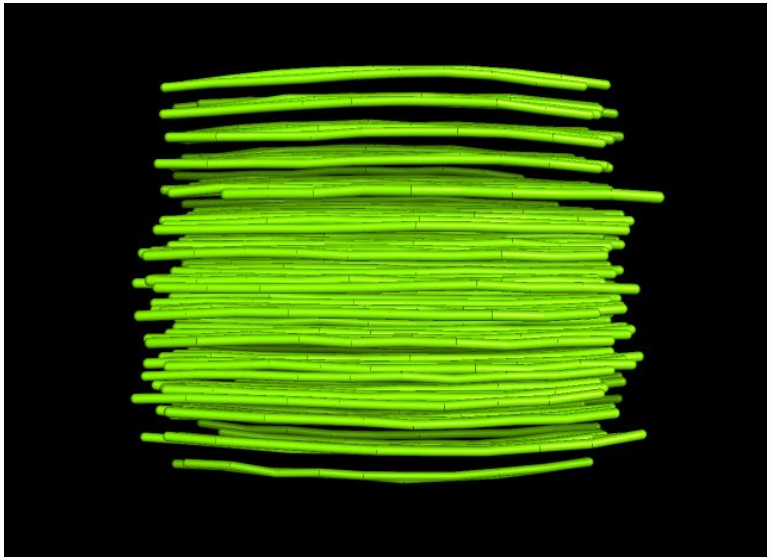
https://ffea.bitbucket.io/

FFEA - A recent case study (26th August)

- Asked to review it’s performance
- Sent an example of a current user’s workload.
- Models a bundle of “rods”

```
./ffea myofilaments.ffea
```

Runtime 312 seconds



FFEA - A recent case study (26th August)

- Re-compile for gprof (basic C/C++ profiler)

```
cmake
  -DCMAKE_C_FLAGS=-pg
  -DCMAKE_CXX_FLAGS=-pg
  -DCMAKE_EXE_LINKER_FLAGS=-pg
  -DCMAKE_SHARED_LINKER_FLAGS=-pg
  -DUSE_OPENMP=OFF
  ..

cmake --build .
```

- Then profile

```
./ffea myofilaments.ffea

gprof ffea gmon.out > analysis.txt
```

FFEA - A recent case study (26th August)

The results:

```
Flat profile:

Each sample counts as 0.01 seconds.

  %   cumulative   self              self     total

 time   seconds   seconds    calls  ms/call  ms/call  name

 53.94      2.12     2.12 36497000     0.00     0.00  rod::Rod::Rod(rod::Rod const&)

 19.85      2.90     0.78 72994000     0.00     0.00  std::vector<...>* std::__do_uninit_copy<...>(...)

 11.45      3.35     0.45 36497000     0.00     0.00  rod::Rod::~Rod()

  2.29      3.44     0.09 20715750     0.00     0.00  rod::get_element_midpoint(...)

  1.78      3.51     0.07 18073000     0.00     0.00  rod::Rod::get_p(int, std::array<float, 3ul>&, bool)

...
```

FFEA - A recent case study (26th August)

The results:

54% Rod copy constructor
20% Vector copy method
11% Rod destructor
🚩🚩🚩🚩🚩🚩🚩🚩

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.94      2.12     2.12 36497000     0.00     0.00  rod::Rod::Rod(rod::Rod const&)
 19.85      2.90     0.78 72994000     0.00     0.00  std::vector<...>* std::__do_uninit_copy<...>(...)
 11.45      3.35     0.45 36497000     0.00     0.00  rod::Rod::~Rod()
  2.29      3.44     0.09 20715750     0.00     0.00  rod::get_element_midpoint(...)
  1.78      3.51     0.07 18073000     0.00     0.00  rod::Rod::get_p(int, std::array<float, 3ul>&, bool)
...
```

FFEA - A recent case study (26th August)

Reviewing the code:

Per Rod:
30 scalars
28 vectors
3 strings

```
/** Global simulation parameters - eventually read in from the .ffea file **/
float viscosity = 0.6913 * pow(10, -3) / (mesoDimensions::pressure * mesoDimensions::time);  // denominator: poiseui
float timestep = 1e-12 / mesoDimensions::time;
float kT = 0;
float perturbation_amount = 0.001 * pow(10, -9) / mesoDimensions::length; /** Amount by which nodes are perturbed du
int calc_noise = 0;
int calc_steric = 0;  // Repulsive overlap of elements
int calc_vdw = 0;   // Attractive protein-protein interactions
int pbc = 0;
float max_steric_energy = 50;    /** Potential energy when two rod elements are fully overlapped [energy units] **/
std::string flow_profile;       // the type of background flow experienced by the rod (set by .ffea file)
float flow_velocity[3] = {0};   // background flow imposed on the rod (set by .ffea file)
float shear_rate = 0;           //

float translational_friction;
float rotational_friction; /** these will have to be changed when I end up implementing per-element radius, to be co

/** Each set of rod data is stored in a single, c-style array, most of which go as {x,y,z,x,y,z...} */

std::vector<float> equil_r;                                /** L, Equilibrium configuration of the rod nodes. */
std::vector<float> equil_m;                                /** L, Equilibrium configuration of the material frame. */
std::vector<float> current_r;                              /** L, Current configuration of the rod nodes. */
std::vector<float> current_m;                              /** L, Current configuration of the material frame. */
std::vector<float> internal_perturbed_x_energy_positive; /** L, Energies associated with the perturbations we do in
std::vector<float> internal_perturbed_y_energy_positive; // L
std::vector<float> internal_perturbed_z_energy_positive; // L
```

https://bitbucket.org/FFEA/ffea/src/master/include/rod_structure.h

FFEA - A recent case study (26th August)

Reviewing the code:

All methods return a copy!

```
Rod(int length, int set_rod_no);
Rod(std::string path, int set_rod_no);
Rod set_units();
Rod compute_rest_energy();
Rod do_timestep(std::shared_ptr<std::vector<RngStream>> &rng);
Rod add_force(const float4 &force, int node_index);
Rod pin_node(bool pin_state, int node_index);
Rod load_header(std::string filename);
Rod load_contents(std::string filename);
Rod load_vdw(const std::string filename);
Rod write_frame_to_file();
Rod write_mat_params_vector(const std::vector<float> &vec, float stretch_scale_factor, float t
Rod change_filename(std::string new_filename);
Rod equilibrate_rod(std::shared_ptr<std::vector<RngStream>> &rng);
Rod translate_rod(std::vector<float> &r, const float3 &translation_vec);
Rod rotate_rod(const float3 &euler_angles);
Rod scale_rod(float scale);
Rod get_centroid(const std::vector<float> &r, float3 &centroid);
Rod get_min_max(const std::vector<float> &r, OUT float3 &min, float3 &max);
Rod get_p(int index, OUT float3 &p, bool equil);
Rod get_r(int node_index, OUT float3 &r, bool equil);
```

https://bitbucket.org/FFEA/ffea/src/master/include/rod_structure.h

FFEA - A recent case study (26th August)

Reviewing the code:

- So the developer intended to return a pointer (or reference).
  - Not a copy.
- None of these methods were actually being chained.
  - It wouldn't have worked.

```
    std::cout << "Set units of rod " << this->rod_no << "\n";

    return *this; /** Return a pointer to the object itself instead of void. Allows for method chaining! **/
}
```

https://bitbucket.org/FFEA/ffea/src/master/src/rod_structure.cpp

FFEA - A recent case study (26th August)

The fix:

- Replace all returns with `void`, they weren't being used.
  - Changing to a pointer/reference as intended would also work.

FFEA - A recent case study (26th August)

The fix:

- Replace all returns with `void`, they weren't being used.
  - Changing to a pointer/reference as intended would also work.

The result:

- Rebuild with gprof disabled, OpenMP enabled
- ./ffea myofilaments.ffea
- ***Runtime 31 seconds***
  - 10x speedup

FFEA - A recent case study (26th August)

The significance:

- This mistake was introduced in February 2018
  - Identified 7.5 years later
- About an hour’s work to identify and address
  - It took me longer to workout how to run the example
- Central to all simulations using Rods
  - 10x speedup can be assumed broad
- The full simulation previously took “a week” to run.
  - Now less than 11 hours.

How widespread are these kinds of issues?

(we don’t know…yet)

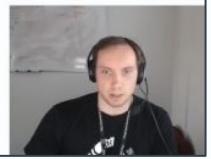
University of Sheffield

Turing 25/26

Reasonable Performance Computing at Sheffield

A case study of easy performance wins

Robert Chisholm

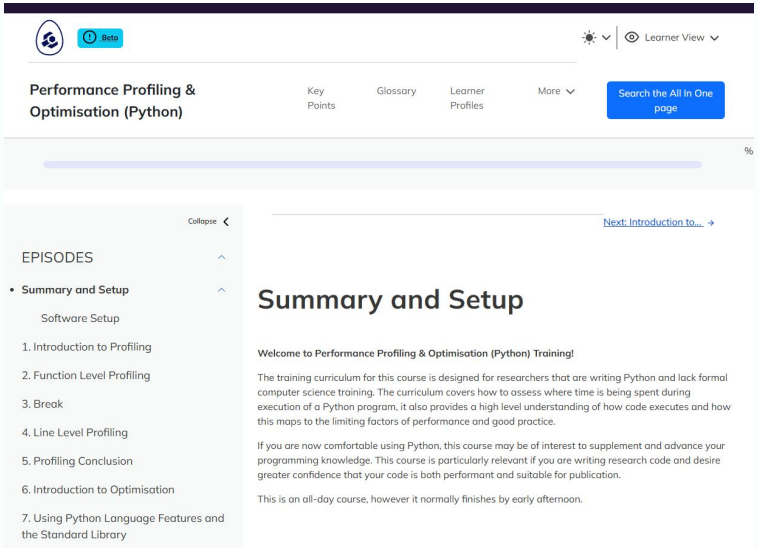


(I’m working on it, right now)

How can we enable researchers to catch and address similar problems sooner?

Develop Training

- Profiling & Optimisation (Python)
  - Carpentries style short-course
  - Developed Jan 2024
  - Now in beta status!
  - Maintained by Jost Migenda (KCL) and myself



https://github.com/carpentries-incubator/pando-python
https://doi.org/10.5281/zenodo.16902755

Develop Training - Doesn’t Scale

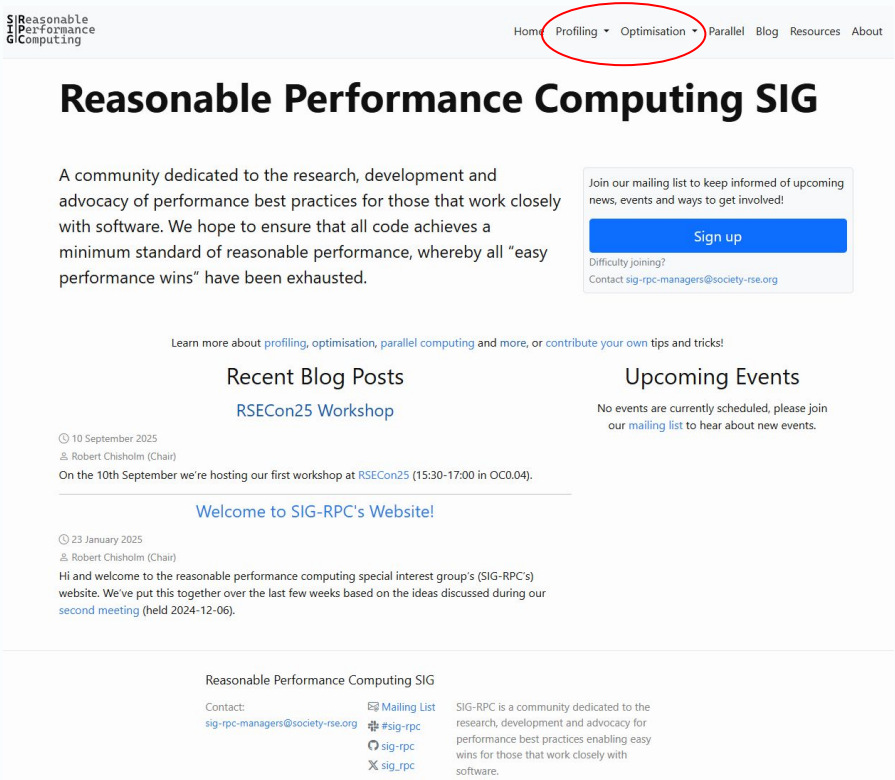
- It took a month to develop less than a day’s worth of training.
  - Further time spent refining/updating it with feedback.
- It only covers the most general Python.
- It’s not possible to create/deliver bespoke training for every combination of languages and libraries used in research.

SIG-RPC Knowledge Base

https://sig-rpc.github.io/

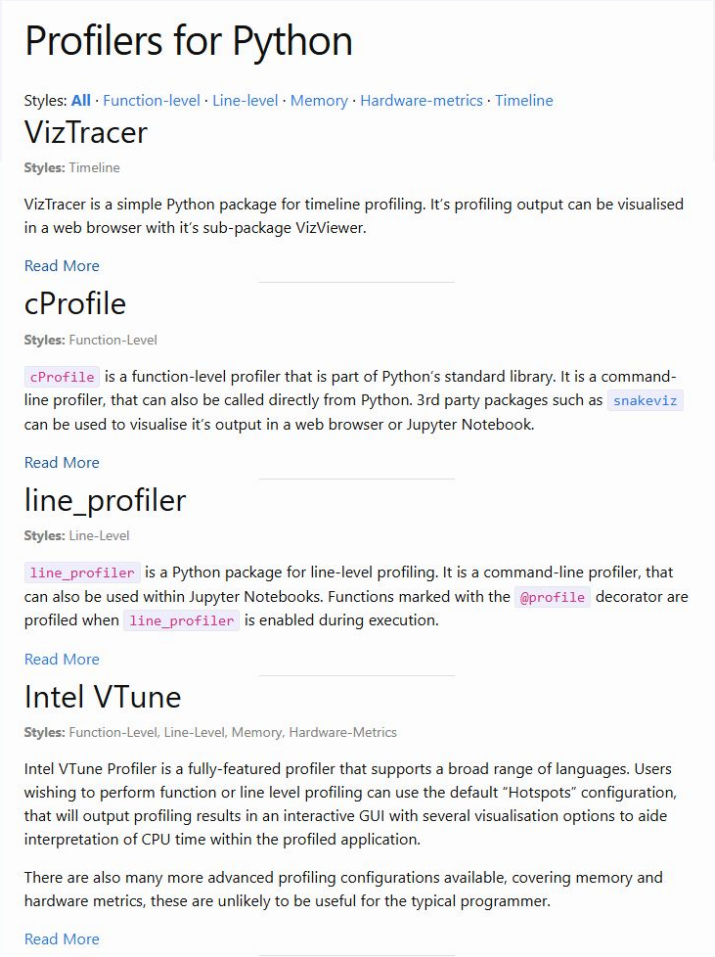
- Mini guides
  - Profiler how-to's
  - Performance patterns
- Case Studies *(eventually)*
- Quick to write
- Easy to understand*

**in theory*



Profilers

- Short high-level profiling intro
- Filtered by
  - Language
  - “Style”
- Suggested Sections:
  - Quickstart
  - Interpreting output
  - Limitations



Profilers for Python

Styles: **All** · Function-level · Line-level · Memory · Hardware-metrics · Timeline

VizTracer

Styles: Timeline

VizTracer is a simple Python package for timeline profiling. It's profiling output can be visualised in a web browser with it's sub-package VizViewer.

Read More

cProfile

Styles: Function-Level

`cProfile` is a function-level profiler that is part of Python's standard library. It is a command-line profiler, that can also be called directly from Python. 3rd party packages such as `snakeviz` can be used to visualise it's output in a web browser or Jupyter Notebook.

Read More

line_profiler

Styles: Line-Level

`line_profiler` is a Python package for line-level profiling. It is a command-line profiler, that can also be used within Jupyter Notebooks. Functions marked with the `@profile` decorator are profiled when `line_profiler` is enabled during execution.

Read More

Intel VTune

Styles: Function-Level, Line-Level, Memory, Hardware-Metrics

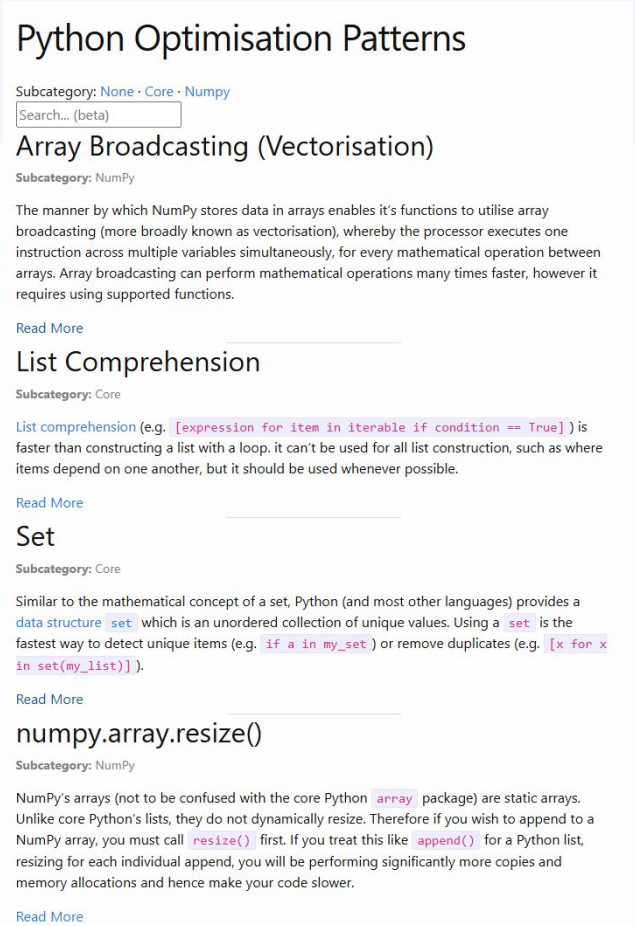
Intel VTune Profiler is a fully-featured profiler that supports a broad range of languages. Users wishing to perform function or line level profiling can use the default “Hotspots” configuration, that will output profiling results in an interactive GUI with several visualisation options to aide interpretation of CPU time within the profiled application.

There are also many more advanced profiling configurations available, covering memory and hardware metrics, these are unlikely to be useful for the typical programmer.

Read More

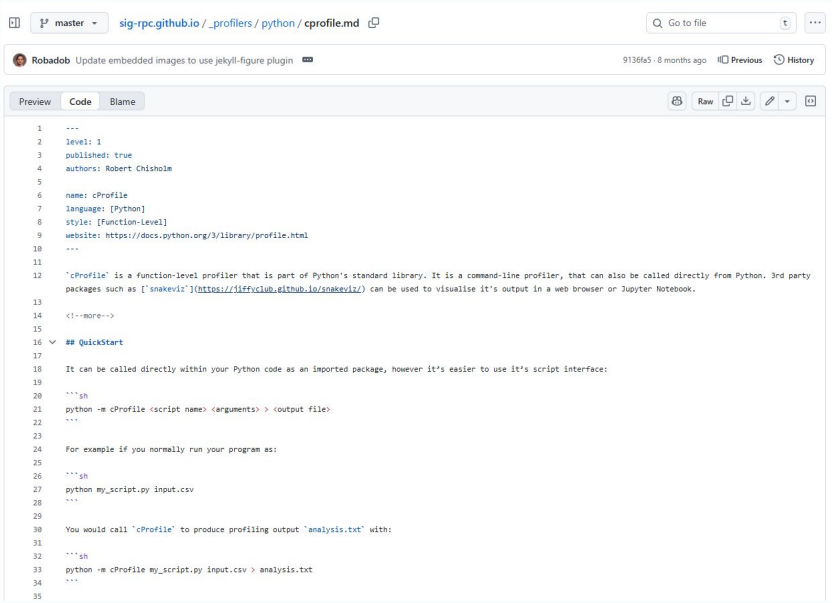
Optimisations

- Filtered by
  - Language
  - “Subcategory”
- Suggested Sections:
  - Description
  - Example benchmark
  - Technical Detail



Easy to Maintain & Extend

- Static Jekyll website
- Guides written in markdown
  - YAML header; priority, authors, name, language, style, website
  - Markdown body
    - <!-- more --> create the fold



master sig-rpc.github.io / _profilers / python / cprofile.md

Go to file

Robadob Update embedded images to use jekyll-figure plugin 9136fa5 · 8 months ago Previous History

Preview Code Blame Raw

```
---
level: 1
published: true
authors: Robert Chisholm

name: cProfile
language: [Python]
style: [Function-Level]
website: https://docs.python.org/3/library/profile.html
---

`cProfile` is a function-level profiler that is part of Python's standard library. It is a command-line profiler, that can also be called directly from Python. 3rd party packages such as [`snakeviz`](https://jiffyclub.github.io/snakeviz/) can be used to visualise it's output in a web browser or Jupyter Notebook.

<!--more-->

## QuickStart

It can be called directly within your Python code as an imported package, however it's easier to use it's script interface:

```sh
python -m cProfile <script name> <arguments> > <output file>
```

For example if you normally run your program as:

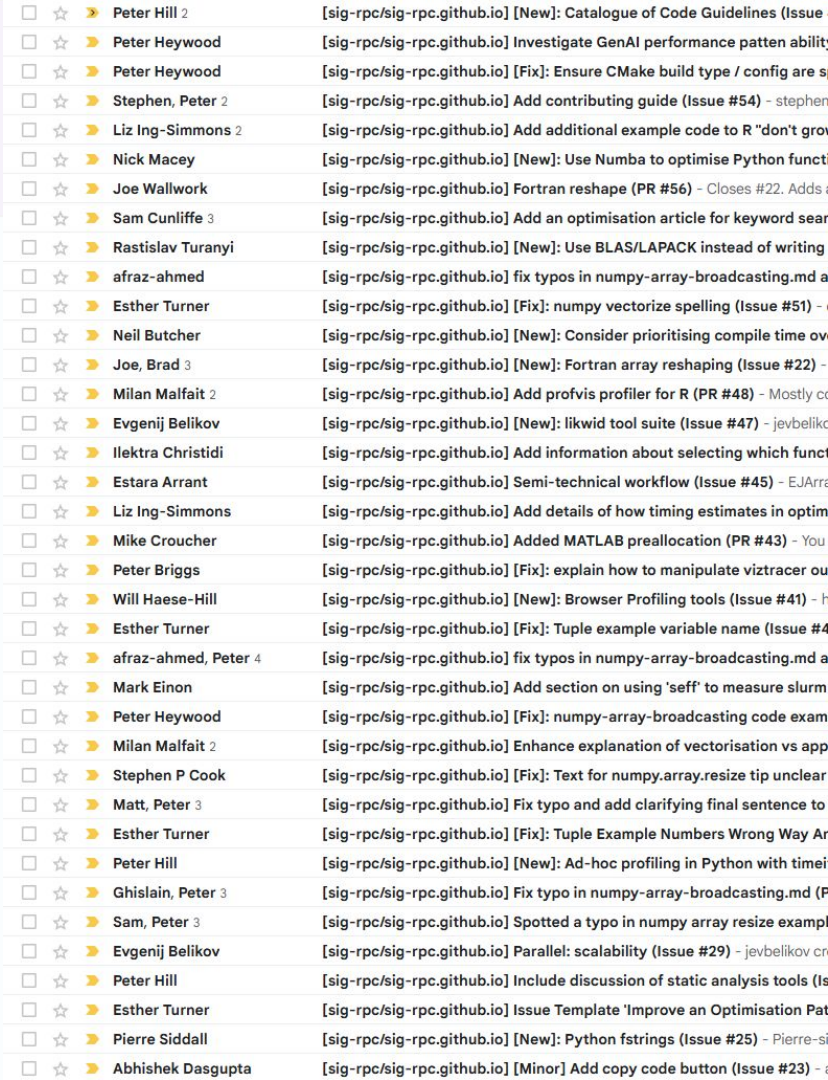
```sh
python my_script.py input.csv
```

You would call `cProfile` to produce profiling output `analysis.txt` with:

```sh
python -m cProfile my_script.py input.csv > analysis.txt
```
```

RSECon25 Workshop

- Recently ran a workshop to elicit feedback/submissions
- ~40 issues/PRs to work through



Get Involved!

- SocRSE Slack #sig-rpc
- Mailing List (via website)
- AGM tomorrow 2pm
- We have hex-stickers

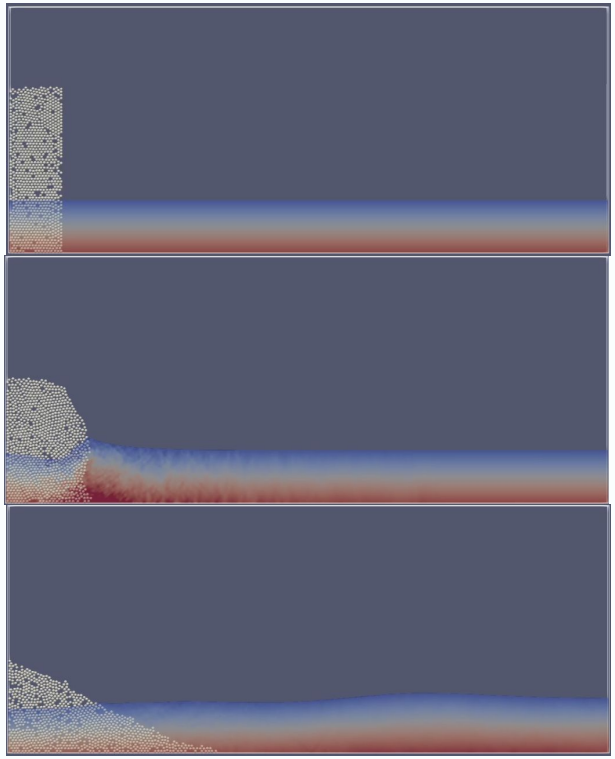
Contact me: robert.chisholm@sheffield.ac.uk

Case Study #2

(If i’m too fast)

HYBIRD

- HYBIRD (Combined LBM & DEM solver)
- C++ & OpenMP
- Development began ~2013
- 1 author
  - PhD student then, now senior lecturer
- 2 contributors
  - PDRAs, reluctant programmers
- ~23 publications
- Used by UGT/PGR students, PDRAs, collaborators



https://github.com/gnomeCreative/HYBIRD

First Profile (DEM Tutorial, 1005 particles)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls     s/call   s/call  name
  9.60    167.59   167.59 4068178008     0.00     0.00  DEM::particleParticleCollision(particle const*,
particle const*, tVect const&, Elongation*)
  8.66    318.68   151.08 153054682294   0.00     0.00  tVect::operator+(tVect const&) const
  6.38    429.95   111.27 12659255385    0.00     0.00  tVect::norm() const
  5.92    533.25   103.30 2392019595     0.00     0.00  elmt::predict(double const*, double const*)
  5.88    635.80   102.55 117157305920   0.00     0.00  tVect::operator*(double const&) const
  5.58    733.22    97.42 9568078380     0.00     0.00  project(tVect, quaternion)
  5.30    825.80    92.58 2392019595     0.00     0.00  elmt::correct(double const*, double const*)
  4.78    909.21    83.40 47840391900    0.00     0.00  quaternion::operator+(quaternion const&) const
  4.00    979.01    69.81  2380119       0.00     0.00  DEM::particleParticleContacts()
  3.61   1041.96    62.95 23934514861    0.00     0.00  tVect::reset()
  3.55   1103.95    61.99 45963670642    0.00     0.00  tVect::operator-(tVect const&) const
  3.40   1163.29    59.33 47840391900    0.00     0.00  quaternion::operator*(double const&) const
  3.32   1221.25    57.97  2380119       0.00     0.00  DEM::evaluateForces()
  2.99   1273.52    52.27 4784039190     0.00     0.00  quaternion::normalize()
  2.99   1325.73    52.21 16726846623    0.00     0.00  tVect::cross(tVect const&) const
  2.61   1371.27    45.54 4295249450     0.00     0.00  DEM::normalContact(...) const
  2.60   1416.69    45.41                              tVect::operator-=(tVect const&)
  2.39   1458.37    41.68 16727253648    0.00     0.00  tVect::operator+=(tVect const&)
  2.13   1495.58    37.21 4295241157     0.00     0.00  DEM::FRtangentialContact(...)
  1.28   1517.96    22.38 6460471666     0.00     0.00  tVect::operator/(double const&) const
```

First Profile (DEM Tutorial, 1005 particles)

Self implemented vector type?

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
9.60	167.59	167.59	4068178008	0.00	0.00	DEM::particleParticleCollision(particle const*, particle const*, tVect const&, Elongation*)
8.66	**318.68**	**151.08**	**153054682294**	**0.00**	**0.00**	**tVect::operator+(tVect const&) const**
6.38	**429.95**	**111.27**	**12659255385**	**0.00**	**0.00**	**tVect::norm() const**
5.92	533.25	103.30	2392019595	0.00	0.00	elmt::predict(double const*, double const*)
5.88	**635.80**	**102.55**	**117157305920**	**0.00**	**0.00**	**tVect::operator*(double const&) const**
5.58	733.22	97.42	9568078380	0.00	0.00	project(tVect, quaternion)
5.30	825.80	92.58	2392019595	0.00	0.00	elmt::correct(double const*, double const*)
4.78	**909.21**	**83.40**	**47840391900**	**0.00**	**0.00**	**quaternion::operator+(quaternion const&) const**
4.00	979.01	69.81	2380119	0.00	0.00	DEM::particleParticleContacts()
3.61	**1041.96**	**62.95**	**23934514861**	**0.00**	**0.00**	**tVect::reset()**
3.55	**1103.95**	**61.99**	**45963670642**	**0.00**	**0.00**	**tVect::operator-(tVect const&) const**
3.40	**1163.29**	**59.33**	**47840391900**	**0.00**	**0.00**	**quaternion::operator*(double const&) const**
3.32	1221.25	57.97	2380119	0.00	0.00	DEM::evaluateForces()
2.99	**1273.52**	**52.27**	**4784039190**	**0.00**	**0.00**	**quaternion::normalize()**
2.99	**1325.73**	**52.21**	**16726846623**	**0.00**	**0.00**	**tVect::cross(tVect const&) const**
2.61	1371.27	45.54	4295249450	0.00	0.00	DEM::normalContact(...) const
2.60	1416.69	45.41				tVect::operator-=(tVect const&)
2.39	**1458.37**	**41.68**	**16727253648**	**0.00**	**0.00**	**tVect::operator+=(tVect const&)**
2.13	1495.58	37.21	4295241157	0.00	0.00	DEM::FRtangentialContact(...)
1.28	**1517.96**	**22.38**	**6460471666**	**0.00**	**0.00**	**tVect::operator/(double const&) const**

First Profile (DEM Tutorial)

- Homemade vector, matrix & quaternion classes.
- Lots of tiny methods
- Called **billions** of times

9% Runtime

```
tVect tVect::operator+(const tVect& vec) const {
    return tVect(
        x+vec.x,
        y+vec.y,
        z+vec.z);
}
```

6% Runtime

```
tVect tVect::operator*(const double& scalar) const {
    return tVect(
        x*scalar,
        y*scalar,
        z*scalar);
}
```

First Profile (DEM Tutorial)

- Homemade vector, matrix & quaternion classes.
- Lots of tiny methods
- Called **billions** of times
- Nothing inlined
  - Relative call overhead, likely high!

9% Runtime

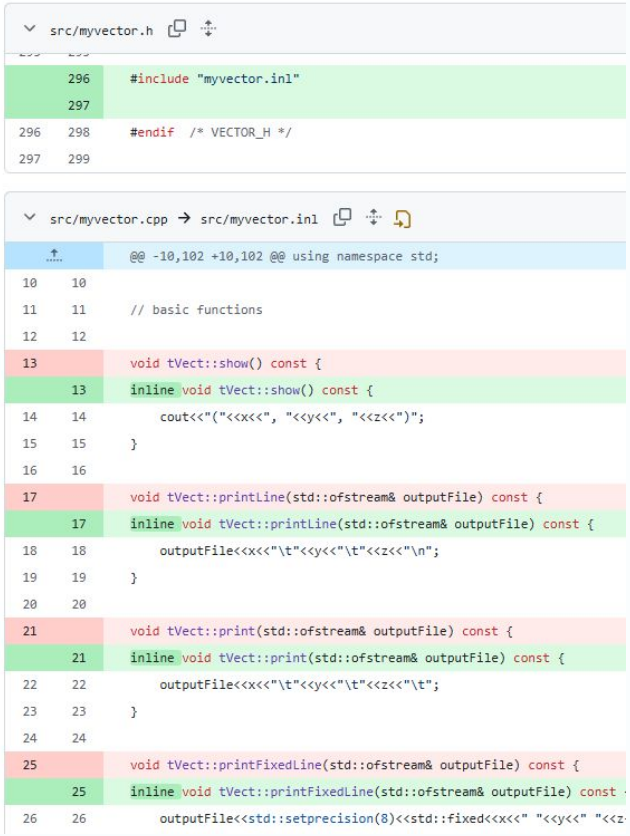
```
tVect tVect::operator+(const tVect& vec) const {
    return tVect(
        x+vec.x,
        y+vec.y,
        z+vec.z);
}
```

6% Runtime

```
tVect tVect::operator*(const double& scalar) const {
    return tVect(
        x*scalar,
        y*scalar,
        z*scalar);
}
```

First Profile (DEM Tutorial)

- Rename file `.cpp` -> `.inl`
- Include it at the end of the respective header
- Mark everything inline
- Runtime before: 24 mins



src/myvector.h

```
296  296  #include "myvector.inl"
297  297
296  298  #endif  /* VECTOR_H */
297  299
```

src/myvector.cpp → src/myvector.inl

```
     @@ -10,102 +10,102 @@ using namespace std;
10   10
11   11  // basic functions
12   12
13       void tVect::show() const {
     13  inline void tVect::show() const {
14   14      cout<<"("<<x<<", "<<y<<", "<<z<<")";
15   15  }
16   16
17       void tVect::printLine(std::ofstream& outputFile) const {
     17  inline void tVect::printLine(std::ofstream& outputFile) const {
18   18      outputFile<<x<<"\t"<<y<<"\t"<<z<<"\n";
19   19  }
20   20
21       void tVect::print(std::ofstream& outputFile) const {
     21  inline void tVect::print(std::ofstream& outputFile) const {
22   22      outputFile<<x<<"\t"<<y<<"\t"<<z<<"\t";
23   23  }
24   24
25       void tVect::printFixedLine(std::ofstream& outputFile) const {
     25  inline void tVect::printFixedLine(std::ofstream& outputFile) const
26   26      outputFile<<std::setprecision(8)<<std::fixed<<x<<" "<<y<<" "<<z
```

First Profile (DEM Tutorial)

- Rename file `.cpp` -> `.inl`
- Include it at the end of the respective header
- Mark everything inline
- Before: 28 mins
- After: 14 mins
  - 1.95x speedup

src/myvector.h

```
296 296  #include "myvector.inl"
297 297
296 298  #endif  /* VECTOR_H */
297 299
```

src/myvector.cpp → src/myvector.inl

```
@@ -10,102 +10,102 @@ using namespace std;
10 10
11 11  // basic functions
12 12
13    - void tVect::show() const {
   13 + inline void tVect::show() const {
14 14      cout<<"("<<x<<", "<<y<<", "<<z<<")";
15 15  }
16 16
17    - void tVect::printLine(std::ofstream& outputFile) const {
   17 + inline void tVect::printLine(std::ofstream& outputFile) const {
18 18      outputFile<<x<<"\t"<<y<<"\t"<<z<<"\n";
19 19  }
20 20
21    - void tVect::print(std::ofstream& outputFile) const {
   21 + inline void tVect::print(std::ofstream& outputFile) const {
22 22      outputFile<<x<<"\t"<<y<<"\t"<<z<<"\t";
23 23  }
24 24
25    - void tVect::printFixedLine(std::ofstream& outputFile) const {
   25 + inline void tVect::printFixedLine(std::ofstream& outputFile) const
26 26      outputFile<<std::setprecision(8)<<std::fixed<<x<<" "<<y<<" "<<z
```

Second Profile (LB tutorial 1, 50 nodes)

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 28.73     16.42    16.42  5451435     0.00     0.00  node::computeApparentViscosity(double const*, FluidMaterial const&)
 23.36     29.77    13.35 16000012     0.00     0.00  void std::__heap_select<std::reverse_iterator<__gnu_cxx::__normal_i
__gnu_cxx::__ops::_Iter_less_iter)
 14.58     38.10     8.33  5372781     0.00     0.00  node::computeEquilibrium(double*)
 11.08     44.43     6.33      156     0.04     0.04  LB::getZ(unsigned int const&) const
 10.52     50.44     6.01  7072394     0.00     0.00  node::addForce(double*, tVect const&)
  5.53     53.60     3.16  7512840     0.00     0.00  node::reconstruct()
  1.92     54.70     1.10  6324299     0.00     0.00  node::solveCollision(double const*)
  0.98     55.26     0.56  5242043     0.00     0.00  node::shiftVelocity(tVect const&)
  0.98     55.82     0.56  2000001     0.00     0.00  LB::streaming(std::vector<wall, std::allocator<wall> >&, std::vecto
  0.89     56.33     0.51  9011735     0.00     0.00  node::store()
  0.31     56.51     0.18  4000003     0.00     0.00  LB::cleanLists()
  0.19     56.62     0.11       68     0.00     0.00  node::initialize(double const&, tVect const&, double const&, double

...
```

Second Profile (LB tutorial 1, 50 nodes)

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 28.73     16.42    16.42  5451435     0.00     0.00  node::computeApparentViscosity(double const*, FluidMaterial const&)
 23.36     29.77    13.35 16000012     0.00     0.00  void std::__heap_select<std::reverse_iterator<__gnu_cxx::__normal_i
__gnu_cxx::__ops::_Iter_less_iter)
 14.58     38.10     8.33  5372781     0.00     0.00  node::computeEquilibrium(double*)
 11.08     44.43     6.33      156     0.04     0.04  LB::getZ(unsigned int const&) const
 10.52     50.44     6.01  7072394     0.00     0.00  node::addForce(double*, tVect const&)
  5.53     53.60     3.16  7512840     0.00     0.00  node::reconstruct()
  1.92     54.70     1.10  6324299     0.00     0.00  node::solveCollision(double const*)
  0.98     55.26     0.56  5242043     0.00     0.00  node::shiftVelocity(tVect const&)
  0.98     55.82     0.56  2000001     0.00     0.00  LB::streaming(std::vector<wall, std::allocator<wall> >&, std::vecto
  0.89     56.33     0.51  9011735     0.00     0.00  node::store()
  0.31     56.51     0.18  4000003     0.00     0.00  LB::cleanLists()
  0.19     56.62     0.11       68     0.00     0.00  node::initialize(double const&, tVect const&, double const&, double
...
```

`std::__heap_select`

23% runtime

Internal C++ list reallocation method

Second Profile (LB tutorial 1, 50 nodes)

```
...
-----------------------------------------------
                                16000012             void std::__heap_select<std::reverse_iterator<__gnu_cxx::__normal_itera
__gnu_cxx::__ops::_Iter_less_iter) [3]
                0.00    0.00       1/16000012     LB::latticeBolzmannInit(std::vector<cylinder, std::allocator<cylinder>
                1.67    4.51 2000001/16000012     LB::latticeBolzmannStep(std::vector<elmt, std::allocator<elmt> >&, std:
                1.67    4.51 2000001/16000012     LB::latticeBoltzmannFreeSurfaceStep() [6]
               10.01   27.08 12000009/16000012      LB::cleanLists() [4]
[3]     86.5   13.35   36.10 16000012+16000012 void std::__heap_select<std::reverse_iterator<__gnu_cxx::__normal_iterator
[3]
               16.42    0.00 5451435/5451435     node::computeApparentViscosity(double const*, FluidMaterial const&) [8]
                8.33    0.00 5372781/5372781     node::computeEquilibrium(double*) [9]
                6.01    0.00 7072394/7072394     node::addForce(double*, tVect const&) [13]
                3.16    0.00 7512840/7512840     node::reconstruct() [14]
                1.10    0.00 6324299/6324299     node::solveCollision(double const*) [17]
                0.56    0.00 5242043/5242043     node::shiftVelocity(tVect const&) [18]
                0.51    0.00 9011735/9011735     node::store() [20]
                0.01    0.00 9854995/19856644     node::isFluid() const [30]
                0.00    0.00 22744828/22744828     node::massStream(unsigned int const&) const [45]
                0.00    0.00 22152591/32108542     node::isInterface() const [44]
                                16000012             void std::__heap_select<std::reverse_iterator<__gnu_cxx::__normal_itera
__gnu_cxx::__ops::_Iter_less_iter) [3]
-----------------------------------------------
...
```

Called inside `LB::cleanLists()`

Second Profile (LB tutorial 1, 50 nodes)

```
2082  void LB::cleanLists() {
2083      // sorts the active-node list and removes duplicates
2084      std::sort(fluidNodes.begin(), fluidNodes.end());
2085      std::sort(interfaceNodes.begin(), interfaceNodes.end());
2086
2087      for (int ind = fluidNodes.size() - 1; ind > 0; --ind) {
2088          node* i = fluidNodes[ind];
2089          node* j = fluidNodes[ind - 1];
2090          if (i == j) {
2091              cout << "duplicate-fluid!" << endl;
2092              fluidNodes.erase(fluidNodes.begin() + ind);
2093          }
2094      }
2095      for (int ind = interfaceNodes.size() - 1; ind > 0; --ind) {
2096          node* i = interfaceNodes[ind];
2097          node* j = interfaceNodes[ind - 1];
2098          if (i == j) {
2099              cout << "duplicate-interface!" << endl;
2100              interfaceNodes.erase(interfaceNodes.begin() + ind);
2101          }
2102      }
2103
2104      // list with active nodes i.e. nodes where collision and streaming are solved
2105      // solid nodes, particle nodes and gas nodes are excluded
2106      activeNodes.clear();
2107      activeNodes.reserve(fluidNodes.size() + interfaceNodes.size());
2108      activeNodes.insert(activeNodes.end(), fluidNodes.begin(), fluidNodes.end());
2109      activeNodes.insert(activeNodes.end(), interfaceNodes.begin(), interfaceNodes.end());
2110
2111  }
```

- Sort 2 lists
- Iterate both lists, to remove duplicate elements
- Fill a new combined list

Second Profile (LB tutorial 1, 50 nodes)

- Use a set?

Second Profile (LB tutorial 1, 50 nodes)

- Use a set?
- This validation code was redundant!
- Before runtime: 2 minute

Second Profile (LB tutorial 1, 50 nodes)

- Use a set?
- This validation code was redundant!
- Before runtime: 2 minute
- After runtime: 1 minute
  - 1.8x speedup
  - But it had poor scalability

Third Profile (LB, 2.2 million nodes)

```
Top Hotspots
Function                      Module          CPU Time  % of CPU Time(%)
---------------------------  ------------  ----------  ----------------
LB::smoothenInterface        hybird        18493.157s              41.4%
LB::streaming._omp_fn.1      hybird         3978.943s               8.9%
operator*                    hybird         2572.052s               5.8%
node::store                  hybird         2254.056s               5.0%
gomp_team_barrier_wait_end   libgomp.so.1   1868.693s               4.2%
  [Others]                     N/A          15531.628s              34.7%
```

41% of parallel CPU time inside `LB::smoothenInterface()`

Third Profile (LB, 2.2 million nodes)

`LB::smoothenInterface()` removes nodes neighbouring empty nodes from fluid nodes.

Pseudocode:

```
for emptyNode in emptiedNodes:
  for neighbour in emptyNode.neighbours:
    if neighbour.isValid() and neighbour.isFluid():
        newInterfaceNodes.append(neighbour)
        neighbour.redistributeMass()
        # ERROR: TAKE OUT FROM CYCLE, THIS IS KILLING PERFORMANCE
        for fluidNode in fluidNodes:
          if fluidNode == neighbour:
            fluideNodes.erase(fluidNode)
```

Third Profile (LB, 2.2 million nodes)

That comment is real.

Every fluid neighbour of every empty node, would iterate the entire fluid list to remove the neighbour if present.

```
for emptyNode in emptiedNodes:
  for neighbour in emptyNode.neighbours:
    if neighbour.isValid() and neighbour.isFluid():
        newInterfaceNodes.append(neighbour)
        neighbour.redistributeMass()
        # ERROR: TAKE OUT FROM CYCLE, THIS IS KILLING PERFORMANCE
        for fluidNode in fluidNodes:
          if fluidNode == neighbour:
            fluideNodes.erase(fluidNode)
```

Third Profile (LB, 2.2 million nodes)

This is obviously a natural fit for a `set()`

- Before: 6 hours 51 minutes
- After: 2 hours 23 minutes
  - 2.87x speedup

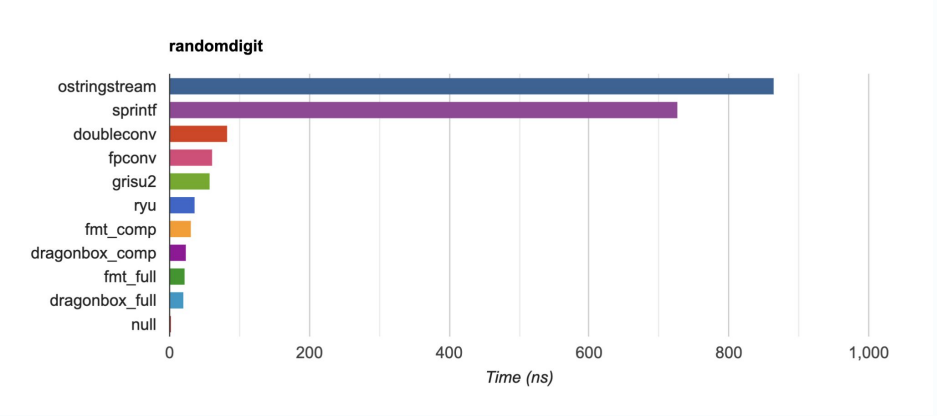
Researchers are almost always familiar with mathematical set, rarely the data-structure set.

```
3052                    // removing from fluid nodes
3053                    // ERROR: TAKE OUT FROM CYCLE, THIS IS KILLING PERFORMANCE
3054                    for (int ind = fluidNodes.size() - 1; ind >= 0; --ind) {
3055                        const unsigned int i = fluidNodes[ind]->coord;
3056                        if (i == link) {
3057                            fluidNodes.erase(fluidNodes.begin() + ind);
3058                        }
3059                    }
3060                    //nodesToErase.push_back(link);
        3052        // Store the node we want to erase later
        3053        nodesToErase.insert(link);
```

```
        3059    // Process and remove all fluid nodes that have been converted
        3060    for (int ind = fluidNodes.size() - 1; ind >= 0; --ind) {
        3061        const unsigned int i = fluidNodes[ind]->coord;
        3062        if (nodesToErase.find(i) != nodesToErase.end()) {
        3063            fluidNodes.erase(fluidNodes.begin() + ind);
        3064        }
        3065    }
        3066
```

Fourth Profile (LB, different IO settings)

- Paraview is used to render simulations
- HYBIRD exports to ASCII .vtk files
- It would take ~20s to load a frame from a large simulation
- 20.4% of runtime reported in `IO::ouputStep()`



https://github.com/fmtlib/dtoa-benchmark

Fourth Profile (LB, different IO settings)

- Rewrite for binary .vtk format
  - Now exporting whole buffers, rather than individual `doubles`
- 10-25x speedup to the IO method
- 40x speedup to Paraview loading frames!

Get Involved!

- SocRSE Slack #sig-rpc
- Mailing List (via website)
- AGM tomorrow 2pm
- We have hex-stickers

Contact me: robert.chisholm@sheffield.ac.uk