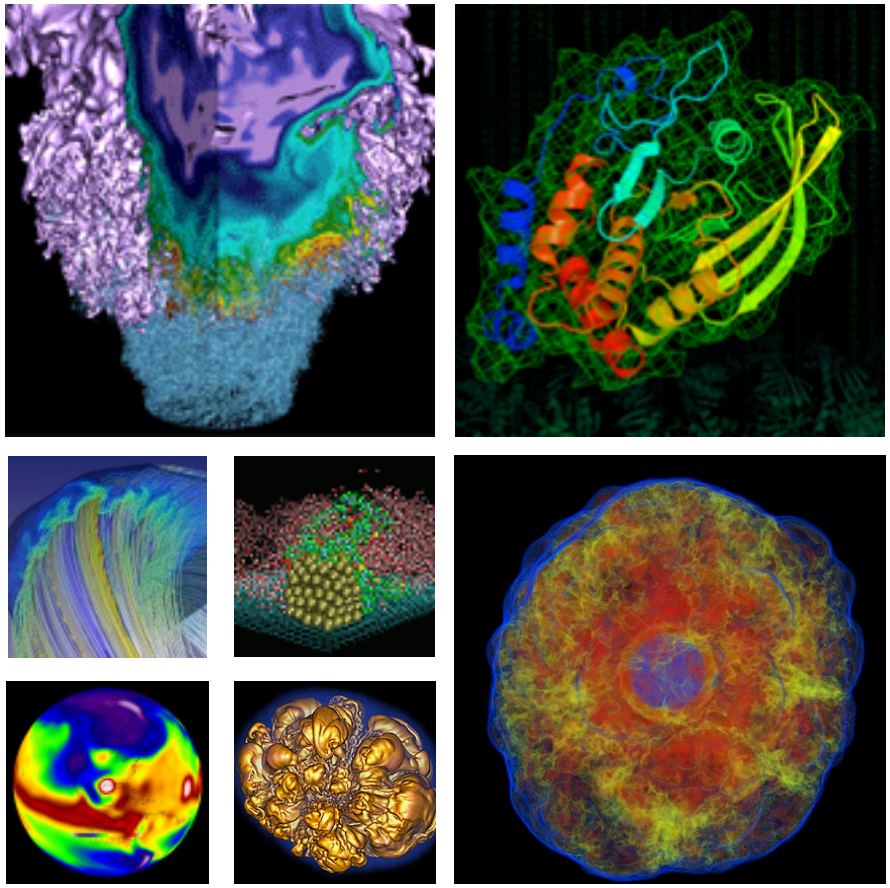
Measuring and understanding parallel scalability



Steve Leak

NERSC User Engagement Group

June 24, 2017

Goals of this tutorial

- **Introductory and practical**
  - Concepts are simple .. Devil is in the details!
  - Gain familiarity with concepts and experience with some tools for identifying scaling limiters
- **Hands-on!**
  - Log in to Cori or Blue Waters, try the exercises

Agenda

- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- OpenMP scaling

---- 1.45pm PDT: 15 minute break ----

- Weak scaling
- Debugging at scale
- Finding the sweet spot
- Summary and Conclusions

Housekeeping

- **This tutorial includes a number of small hands-on exercises**
  - Scripts and instructions are for Cori and Blue Waters
  - Using publicly-available code and mostly Open Source tools => exercises should translate to Stampede or the cluster at your home institution, but scripts will need modification
  - Several tools are GUI-based, login with "ssh –X"
  - On Cori, using NX is highly recommended:

  http://www.nersc.gov/users/connecting-to-nersc/using-nx/

  (much faster GUI performance)

Housekeeping

- **This tutorial includes a number of small hands-on exercises**
  - On Cori, to ensure you have latest version of exercises:

```
module load training
rsync -av $TRAINING .
git pull
```

  - On Blue Waters:

```
git clone https://gitlab.com/s2pi/s2pi2017.git
git pull
```

  - `cd s2pi2017/ScalingProfiling`

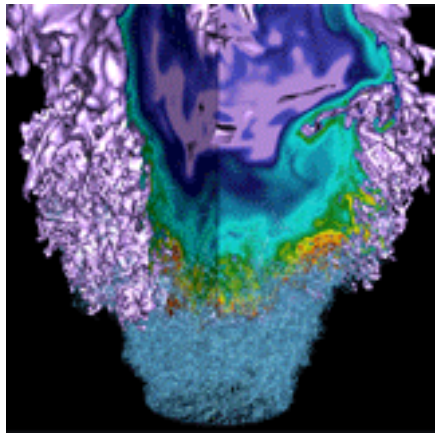
Housekeeping

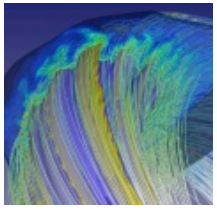
- **This tutorial includes a number of small hands-on exercises**
  - Exercises involve building an executable and submitting a job, then working with the results
  - It's ok not to finish an exercise!
    - There will be some waiting in queues, slow X connections, etc
  - We'll go over canned results after each
  - Open lab at end of day, and/or you can take exercises away to work through later

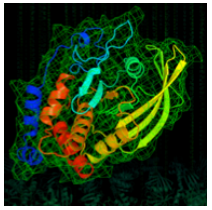
Housekeeping

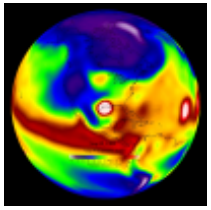
- **Q & A**
  - Each section will have a few minutes at the end for Q & A, given the number of sites and participants it would be best if you can post your questions on the Slack channel
  - Helen, Thorsten and Brandon are online and will answer some questions directly in the Slack channel during the presentation, others will be covered in the Q&A

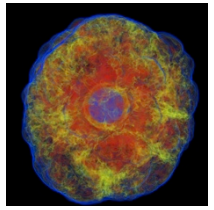
Amdalh's law
Understanding parallel efficiency

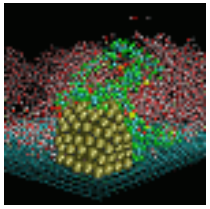






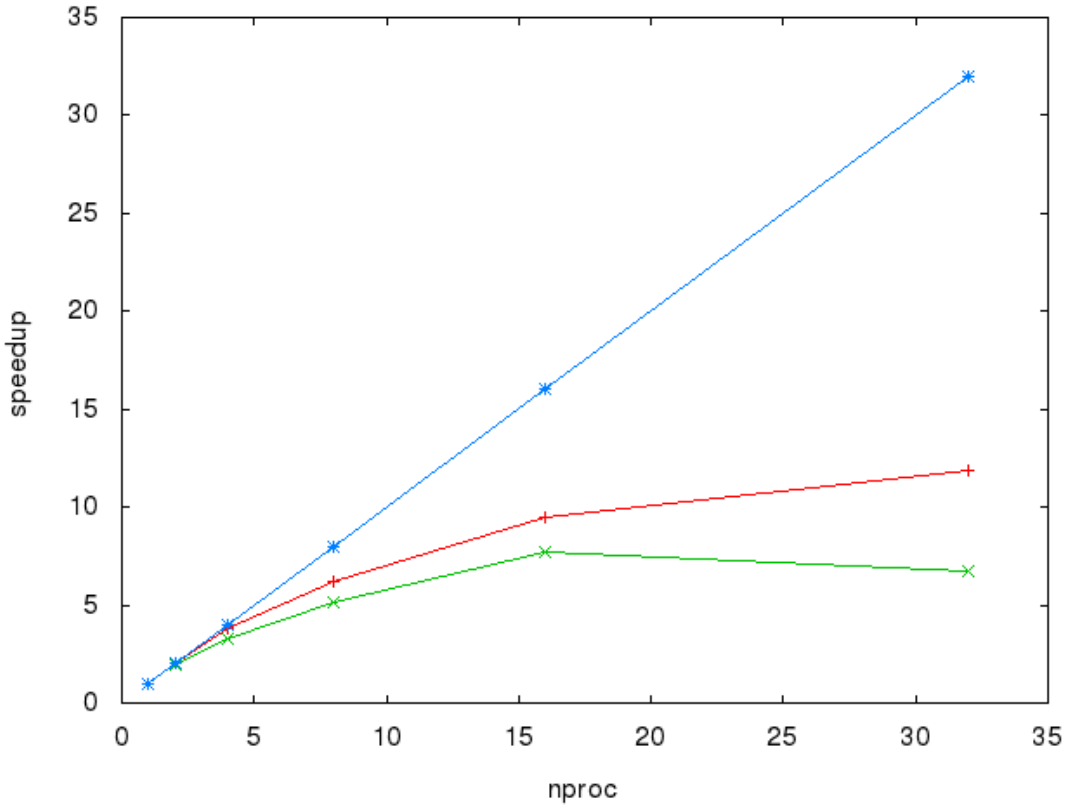






How fast will my parallel code run?

- **Scenario: I can simulate 1 model-day per day on a workstation (one core), how fast will it run on a Cori node (32 cores)?**
  - Or 100 Cori nodes?



Hands-on exercises

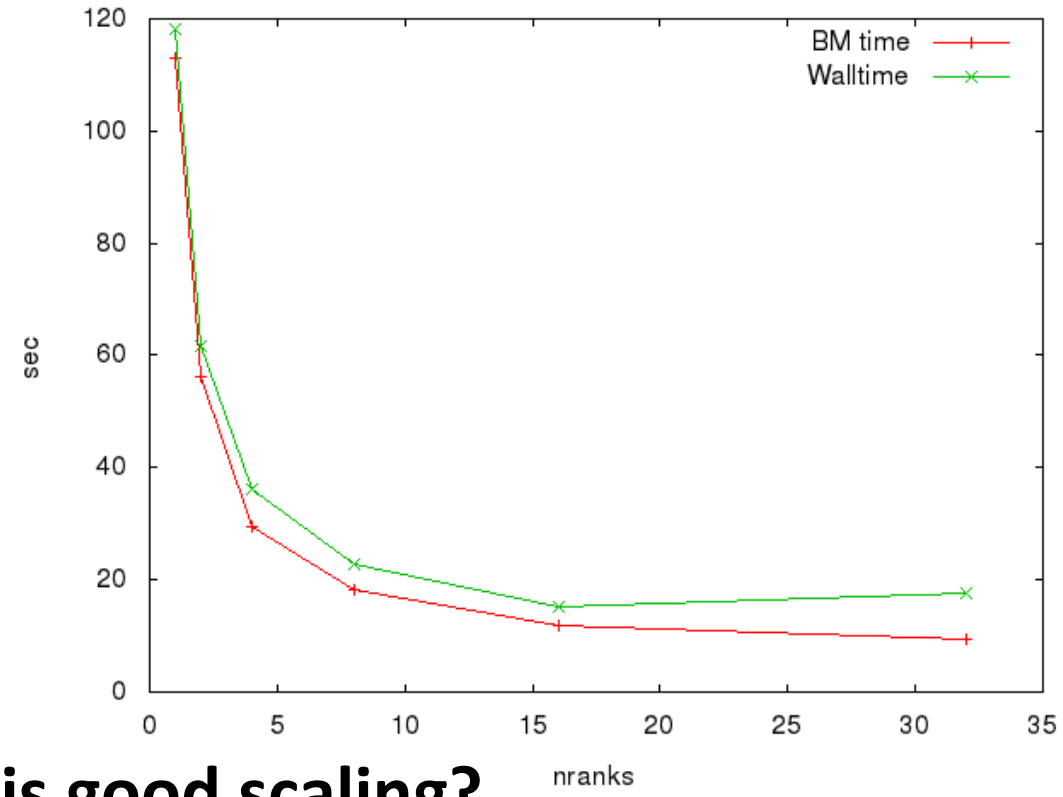
- **Tutorial format:**
  - Series of short hands-on exercises
  - Slides will walk through exercise outcomes and build on these to illustrate aspects of scalability
- **MiniFE**
  - a finite-element mini-application used in Cori procurement – downloadable from NERSC web page
- **Each exercise has:**
  - README.rst – instructions
  - cori-jobscript.sh - for Cori (uncomment reservation line!)
  - bw-jobscript.sh – for Blue Waters

Exercise 1: build and run miniFE

- **See ex1-scaling/README.rst**
  1. Build miniFE
  2. Submit a job to run it on 1, 2, 4, etc MPI processes
     Job script will extract timings into text file
  3. Plot timings with your favorite charting tool
     (Instructions in README.rst use gnuplot, you can use another charting library or a spreadsheet, if you prefer)
- **Note: exercises will use some tools via an X GUI**
  - Use `ssh -X` to connect, have X server running on laptop
  - OR use NX http://www.nersc.gov/users/connecting-to-nersc/using-nx/
- **5-10 minutes, then we'll walk through results**

Exercise 1

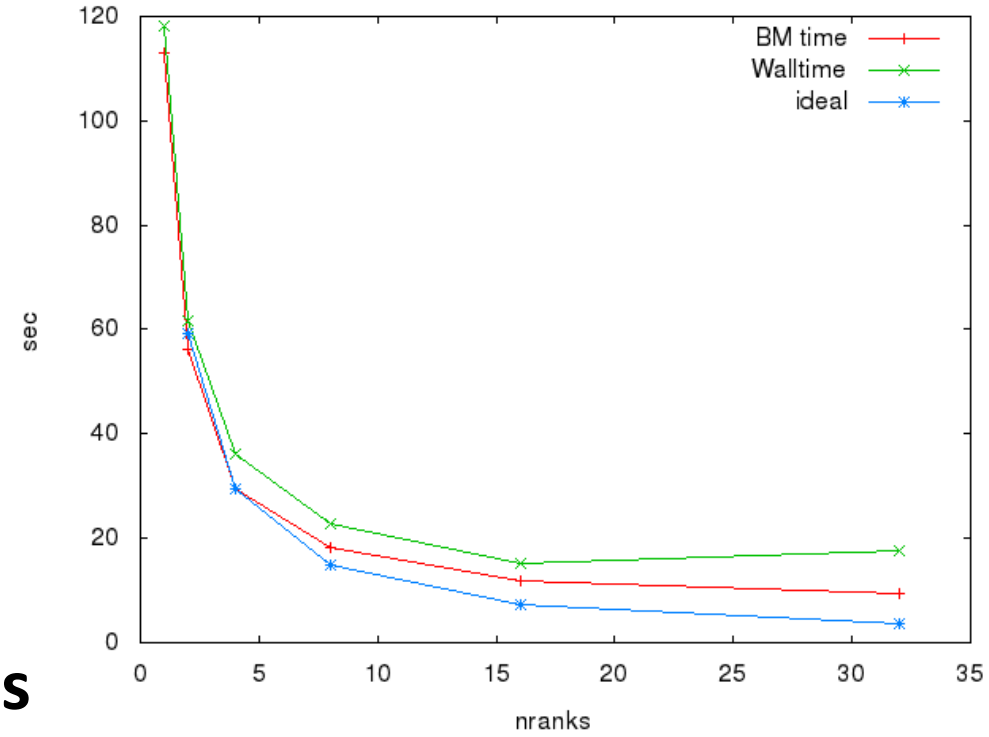
- **First plot probably looks something like this:**



- **Is this good scaling?**

Exercise 1

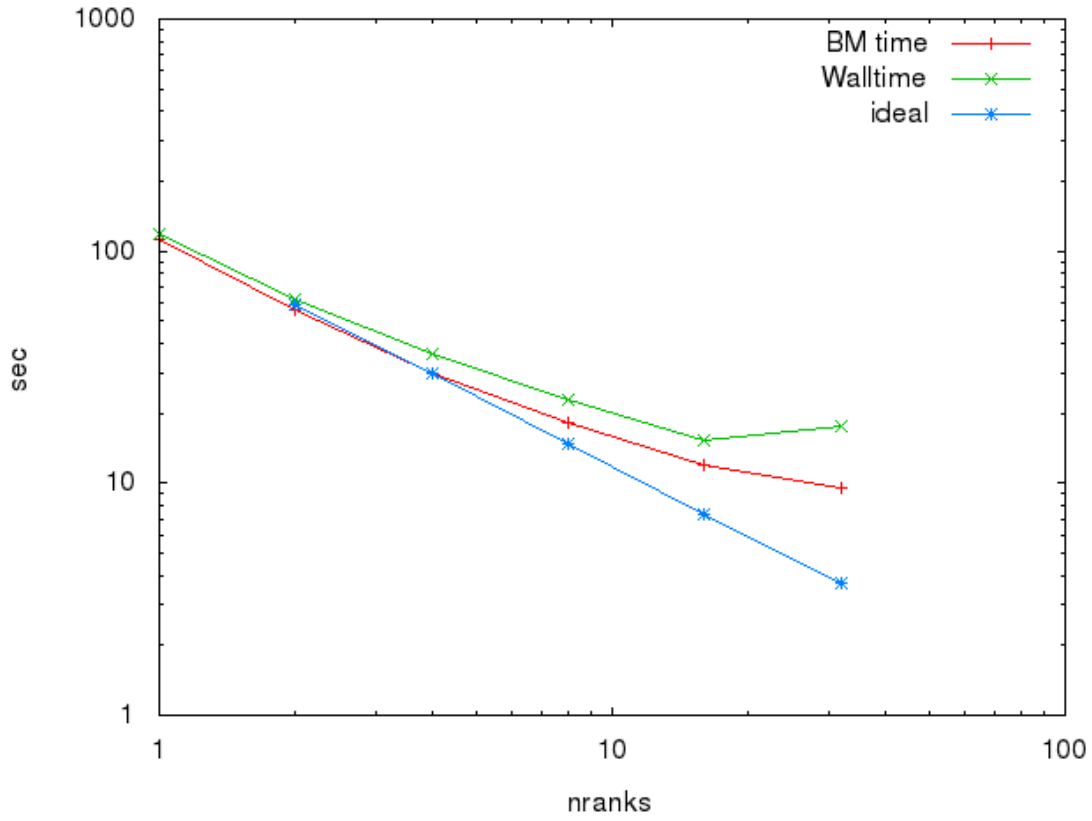
- **Adding a curve for the expected time if speedup were linear:**



- **Diminishing returns**
- **Observation: linear plot of time obscures the scaling behavior**

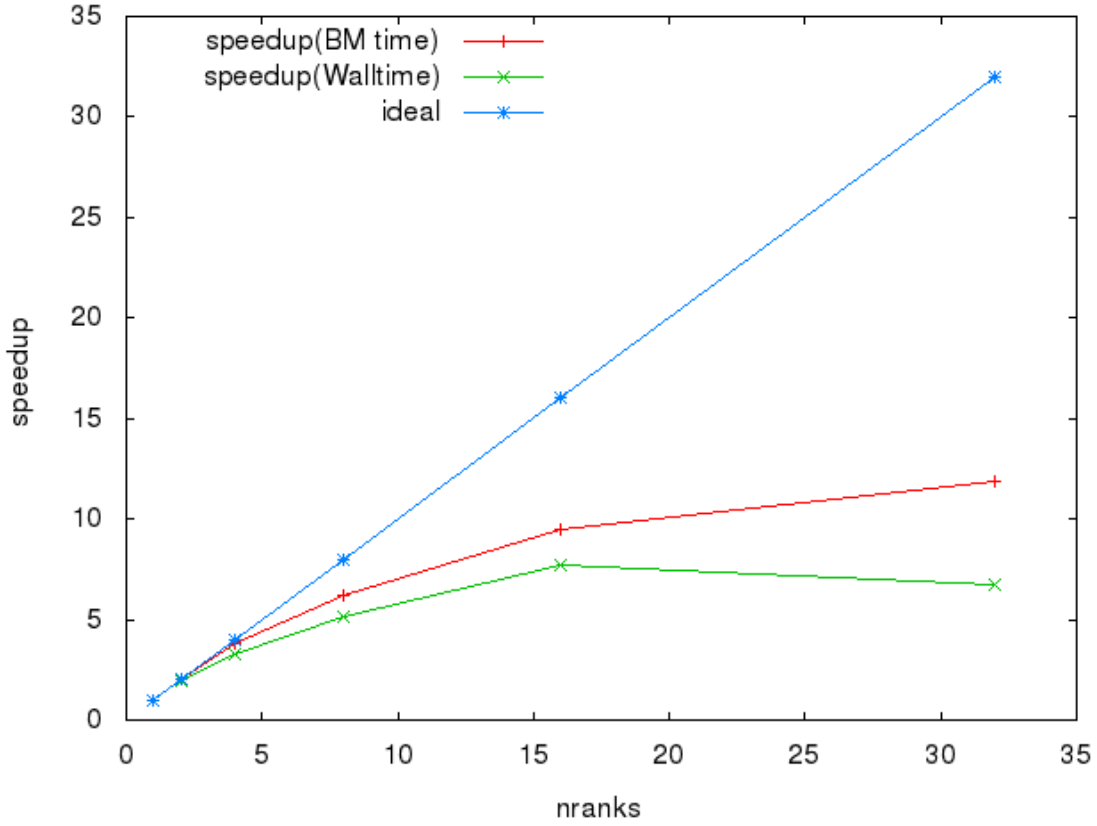
Exercise 1

- **Using a log-log scale is better**
  - Ideal scaling is now a straight line



Exercise 1

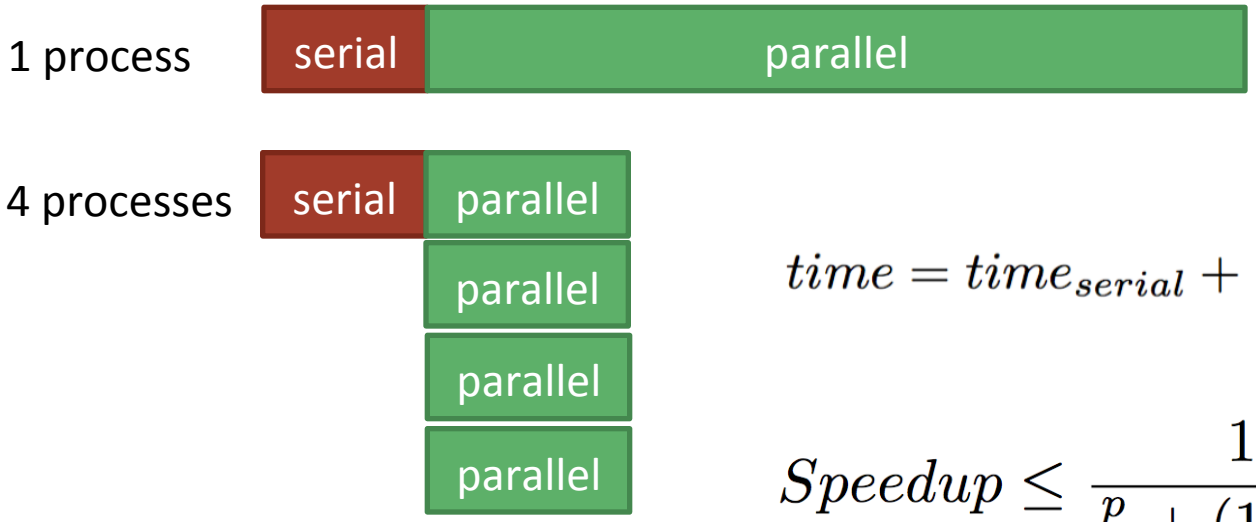
- **Plotting speedup (which is proportional to 1/time) gives a curve that is more intuitively understood:**



$$Speedup(n) = \frac{Time(1)}{Time(n)}$$

Amdahl's law

- **Not all of the work in a sequential program is amenable to parallelism, and that serial component limits the parallel performance**



$$time = time_{serial} + \frac{time_{parallel}}{nproc}$$

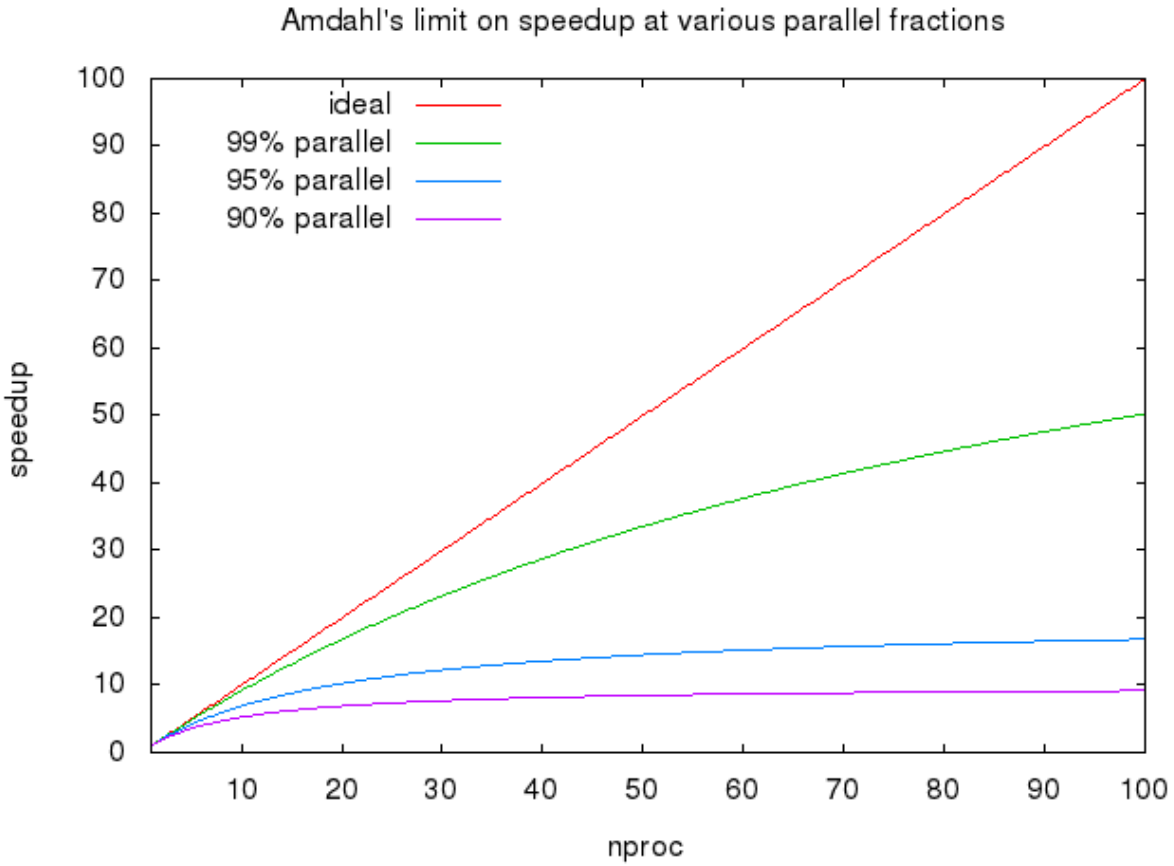
$$Speedup \leq \frac{1}{\frac{p}{n} + (1-p)}$$

Amdahl's law

$$Speedup \leq \frac{1}{\frac{p}{n} + (1-p)}$$

p = parallel fraction
n = number of processes

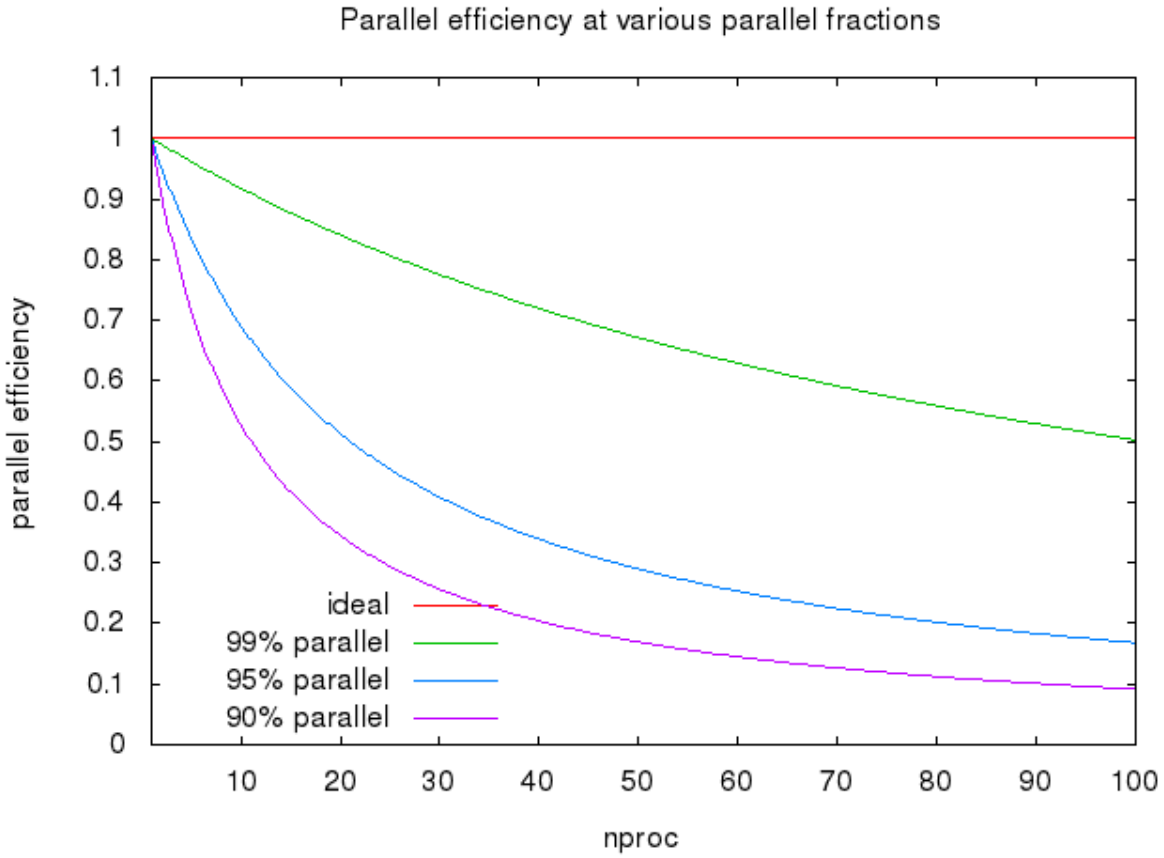
$$Speedup_{p \Rightarrow \infty} \leq \frac{1}{1-p}$$



Parallel efficiency

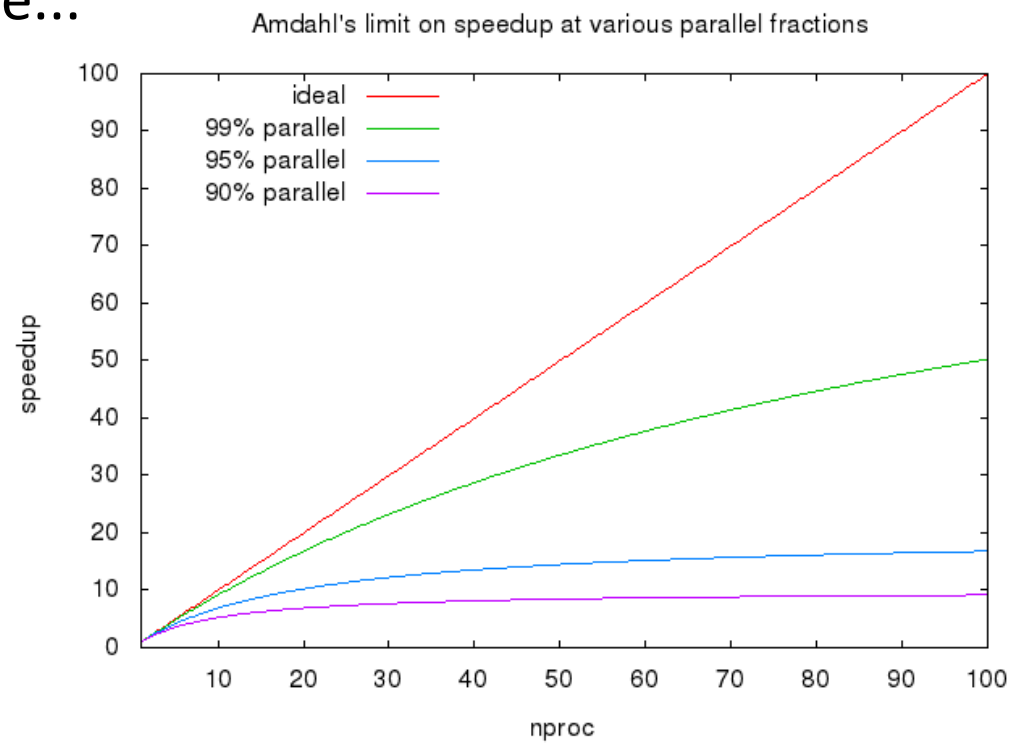
p = parallel fraction
n = number of processes

Efficiency = Speedup / n
("how much value am I getting from my processors?")



Amdahl's law - implications

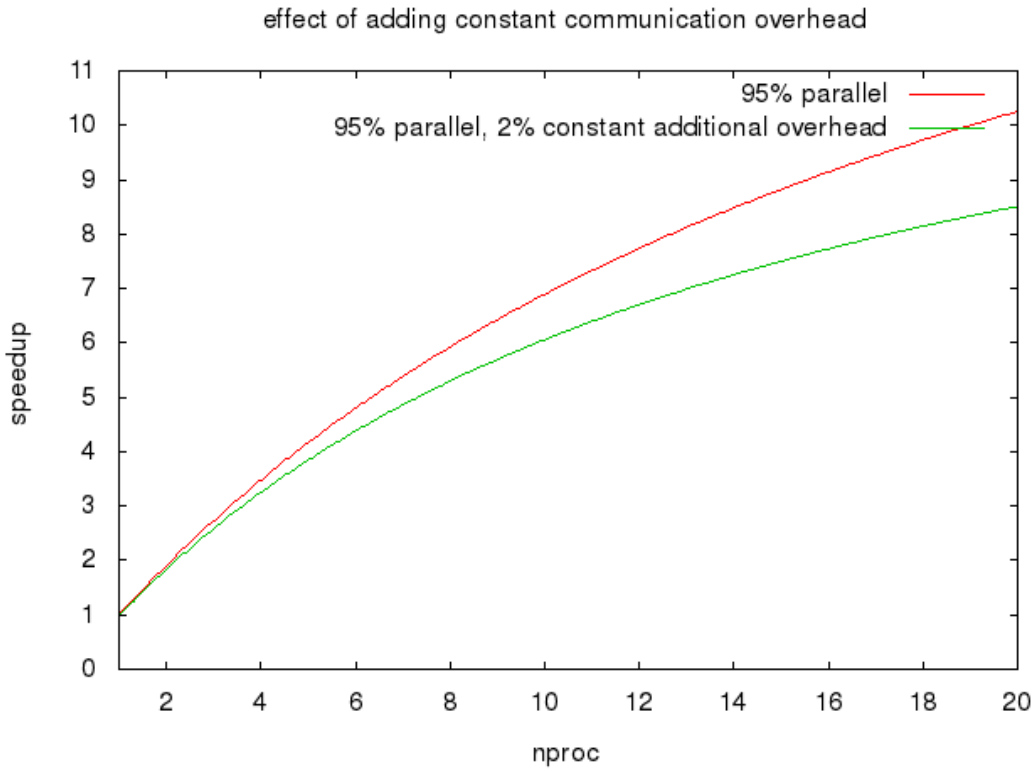
- **Problem: even a small serial fraction drastically limits parallelism!**
  - 1% serial – can never get more than 100x speedup
  - and it gets worse...



Parallel overhead

- **Not only is some work sequential, but parallelism usually adds some overhead**

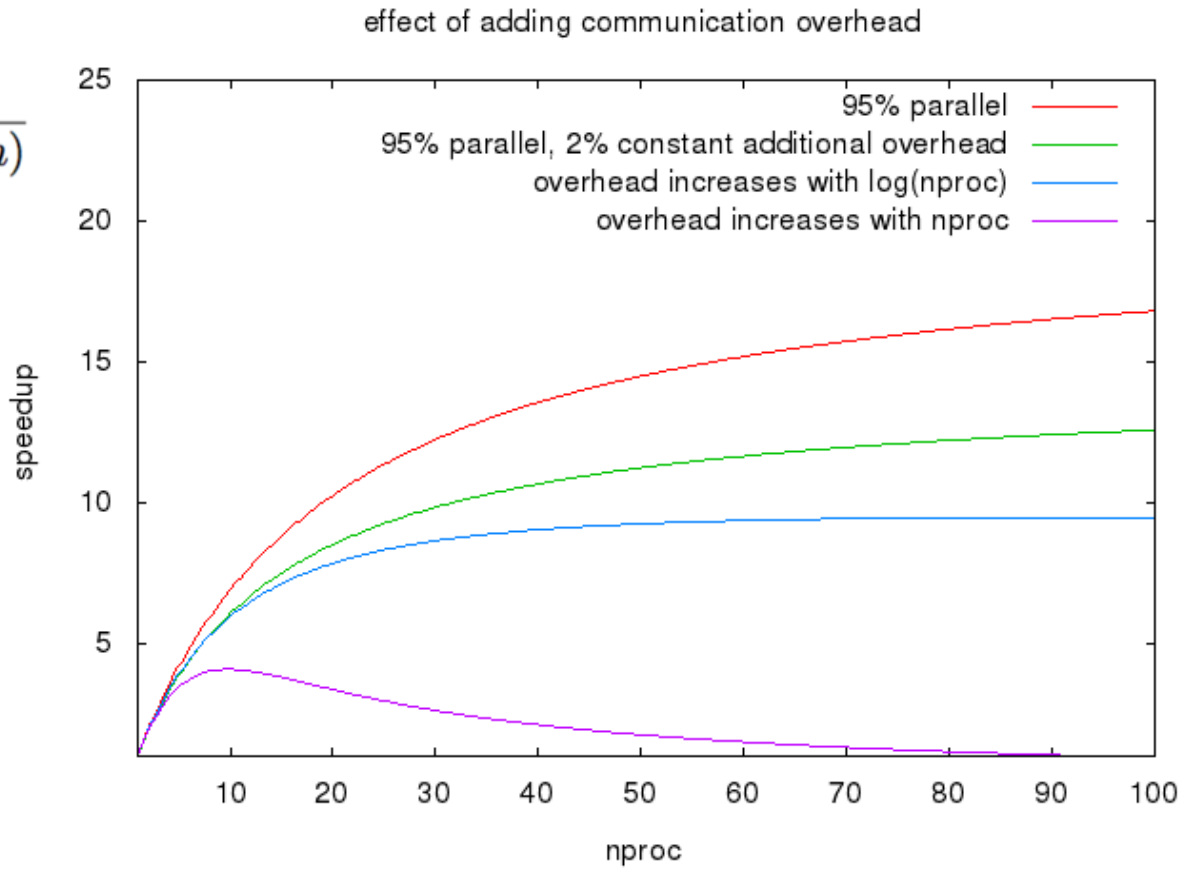
$$Speedup(n) \leq \frac{1}{\frac{p}{n} + (1-p) + c}$$



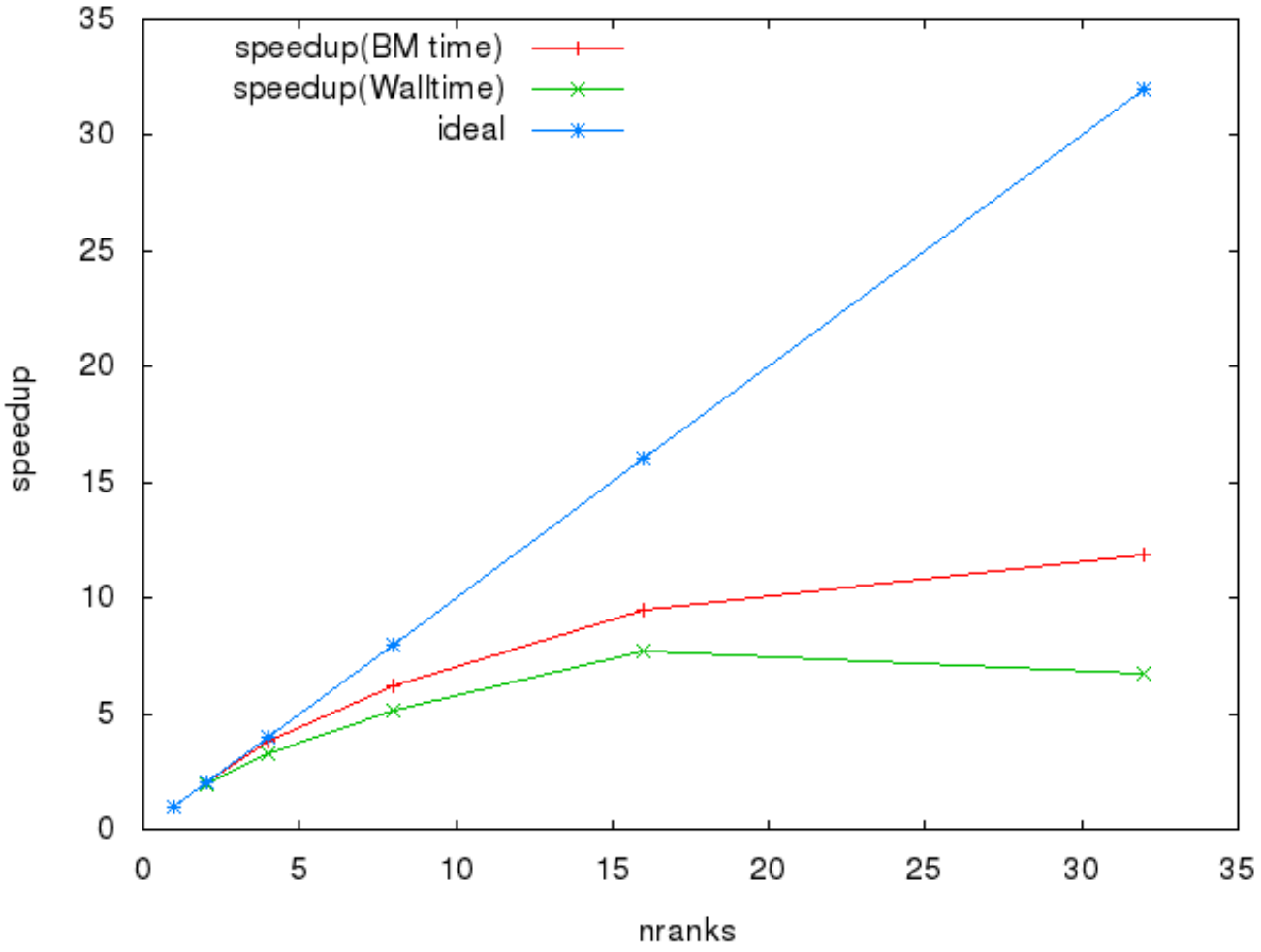
Parallel overhead

- .. And in some cases, the overhead can increase with process count

$$Speedup(n) \leq \frac{1}{\frac{p}{n} + (1-p) + f(n)}$$



Parallel overhead and exercise 1



Key ideas

- **A small reduction in parallelism drastically reduces scalability**
- **A small increase in communication overheads drastically reduces scalability**
- ➢ **Simply using more processors won't get you to petascale!**
  - Need to identify and remove parallelism limiters

Q & A

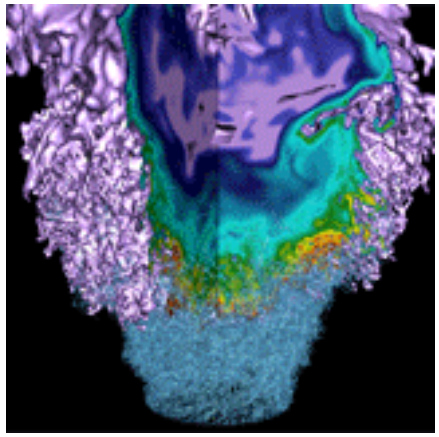
Agenda

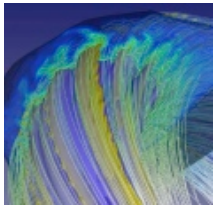
- Amdahl's law
- **Where is the bottleneck?**
- Why is *this* a bottleneck?
- OpenMP scaling

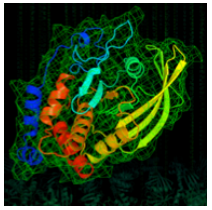
---- 1.45pm PDT: 15 minute break ----

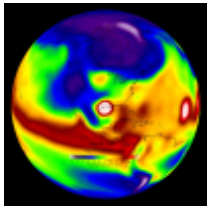
- Weak scaling
- Debugging at scale
- Finding the sweet spot
- Summary and Conclusions

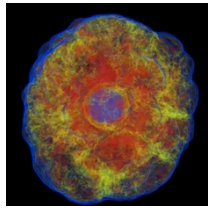
Where is the bottleneck?
Profiling an MPI application

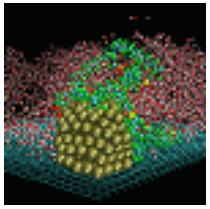






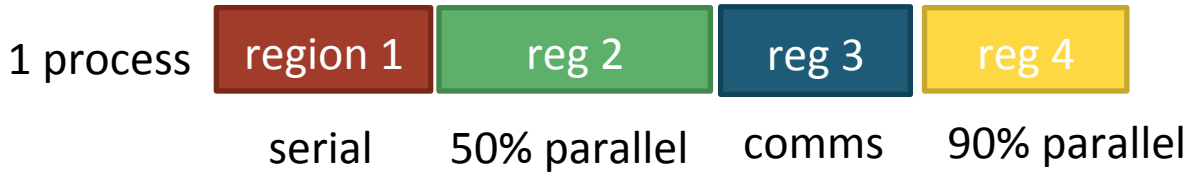


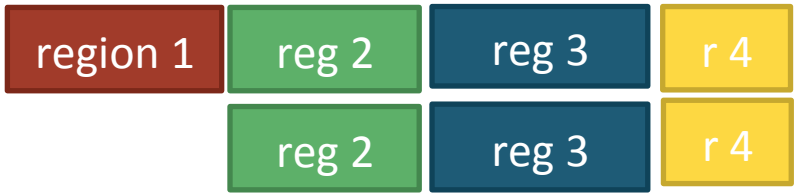


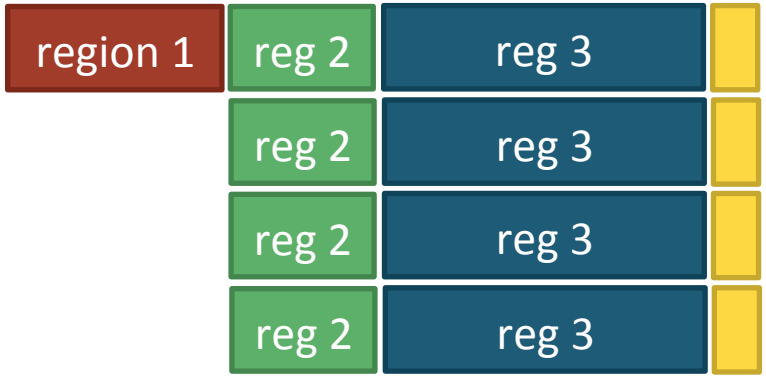


Revisiting Amdahl's law

- **Scaling of a real application is probably not homogeneous – some regions will scale well, others won't.**







- **Need to identify where to focus effort**

Profiling an MPI application

- **What is limiting the scaling?**
  - Do MPI calls dominate the run time at scale?
  - Are specific MPI calls more expensive at scale?
  - Do certain application routines scale poorly?
  - Is load unevenly distributed over processes?

- **Profiling tools help to answer these questions**

Profiling tools and terminology

- **There are many tools for parallel profiling!**
  - Open source: TAU, HPCToolkit, OpenSpeedShop, PerfSuite, IPM, MpiP, ScoreP, gprof, …
  - Proprietary: CrayPat, Allineap MAP, Intel Vtune, Vampir, ITAC, ...
- **Each has strengths and weaknesses**
  - Ease-of-use is frequently in the “weakness” column...
- **Much overlap between tools**
  - Which to use?
  - Familiarity (or support) and availability are good initial criteria!

Profiling tools and terminology

- **"Instrumenting"**
  - Modifying the source code or object file to add hooks for profiling routines
- **"Profiling"**
  - Usually refers to measuring time spent within routines, by capturing timestamp at entry and exit
- **"Sampling"**
  - Usually refers to periodically interrupting the process an recording the current instruction pointer, using statistics to identify hotspots (eg gprof)
  - Often very low overhead

Profiling tools and terminology

- **"Tracing"**
  - Usually means recording events, creating a timeline
    - Eg "started MPI_Send of 800 integers from process 1 to process 19", or "left routine do_matrix_multiply()"
  - Usually very high overhead
- **"Callpath profiling"**
  - When recording an event, capture the call stack too
    - Distinguishes between "MPI_Allreduce() called from solve_matrix()" and "MPI_Allreduce() called from verify_solution()"
  - Usually requires some low-level Linux libraries (libbfd, libunwind, libdwarf,..)

Profiling tools and terminology

- **"Phase profiling"**
  - Profiling in terms of application phases
    - (initialization, solver, checkpoint,..)

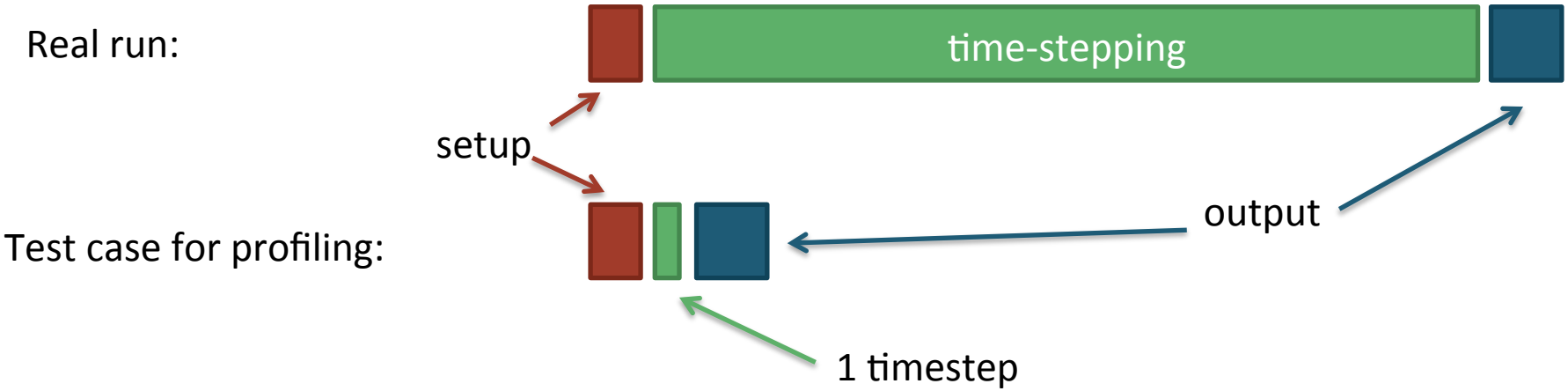
Profiling an MPI application

- **What is limiting the scaling?**
  - Do MPI calls dominate the run time at scale?
  - Are specific MPI calls more expensive at scale?
  - Do certain application routines scale poorly?
  - Is load unevenly distributed over processes?

- **Profiling tools help to answer these questions**
  - **Start with a lower-overhead approach (profiling or sampling) to identify locations of bottlenecks**

Profiling prerequisite

- **A small, short, but representative test case for your application**
  - Will run many times – want it finish within a few minutes
  - Must cover same paths through code as "real" example (or at least, the differences must be understood)
    - What could go wrong with this test case?



Hands-on exercise

- **See ex2-profiling/README.rst**
- **In this exercise we'll use TAU to instrument and profile the miniFE code we used in exercise 1**
  - https://www.cs.uoregon.edu/research/tau/home.php
  - See ex2/README.rst
- **Why TAU?**
  - Free, Open Source
  - Installed on Cori, Blue Waters and Stampede
- **Work through the README.rst file for the next 5-10 minutes, then we'll walk through some results**

Exercise 2

- **Key step is instrumenting the code**
  - Different tools use different means to instrument
  - Tau uses compiler wrapper script and pre-prepared Makefile with rules
    - Select the one that suits your environment / needs
  - (Can also build with dynamic linking and dynamically instrument the code at run-time)

Exercise 2

- **After building, you should see a lot of files relating to instrumentation:**

```
22:38 sleak@cori08:build$ ls -lt
total 38260
-rw-rw---- 1 sleak staff     1703 Jun 16 14:50 Makefile
-rwxrwx--- 1 sleak staff 20196312 Jun 16 14:49 miniFE-tau.x
-rw-rw---- 1 sleak staff    15248 Jun 16 14:49 mytimer.o
-rw-rw---- 1 sleak staff     3749 Jun 16 14:49 mytimer.inst.hr.cpp
drwxrws--- 2 sleak staff     4096 Jun 16 14:49 tau_headers_6c6a2b3b263b551134c789e3daff42d7
-rw-rw---- 1 sleak staff     3729 Jun 16 14:49 mytimer.inst.cpp
-rw-rw---- 1 sleak staff  1233541 Jun 16 14:49 mytimer.pdb
-rw-rw---- 1 sleak staff   237696 Jun 16 14:49 utils.o
-rw-rw---- 1 sleak staff     5060 Jun 16 14:49 utils.inst.hr.cpp
drwxrws--- 2 sleak staff     4096 Jun 16 14:49 tau_headers_8916de098aa624b118b62af74570dfd1
-rw-rw---- 1 sleak staff     5016 Jun 16 14:49 utils.inst.cpp
-rw-rw---- 1 sleak staff  3598913 Jun 16 14:49 utils.pdb
-rw-rw---- 1 sleak staff    89672 Jun 16 14:49 param_utils.o
-rw-rw---- 1 sleak staff     2456 Jun 16 14:49 param_utils.inst.hr.cpp
drwxrws--- 2 sleak staff      512 Jun 16 14:49 tau_headers_0f736bd7a5a00e53677e5a2223357365
-rw-rw---- 1 sleak staff     2445 Jun 16 14:49 param_utils.inst.cpp
-rw-rw---- 1 sleak staff  1922045 Jun 16 14:49 param_utils.pdb
```

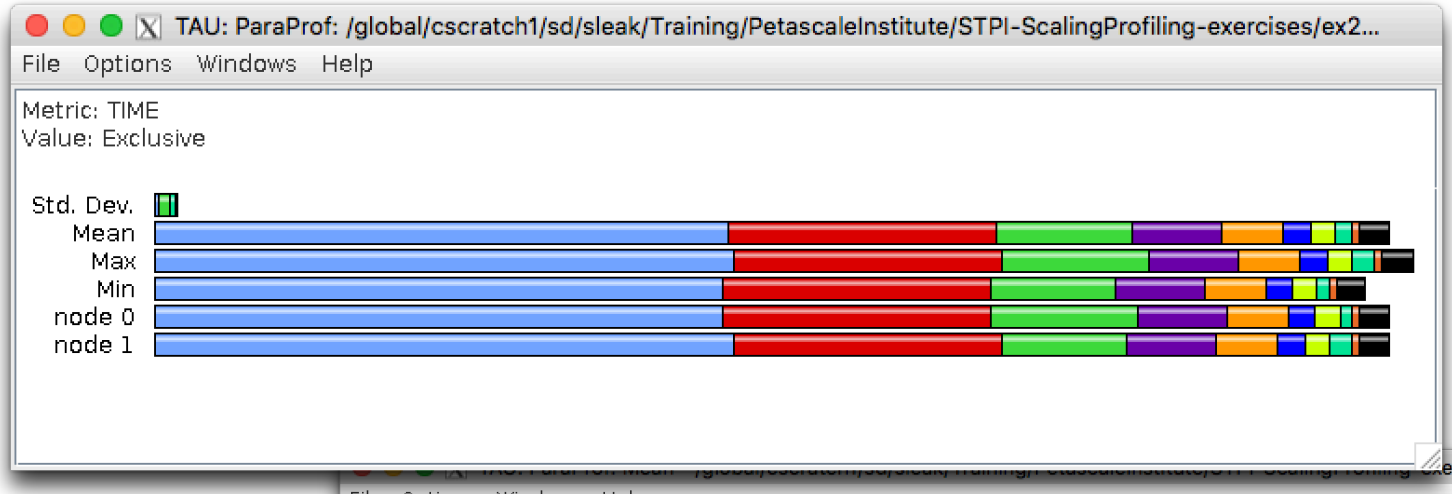
Exercise 2

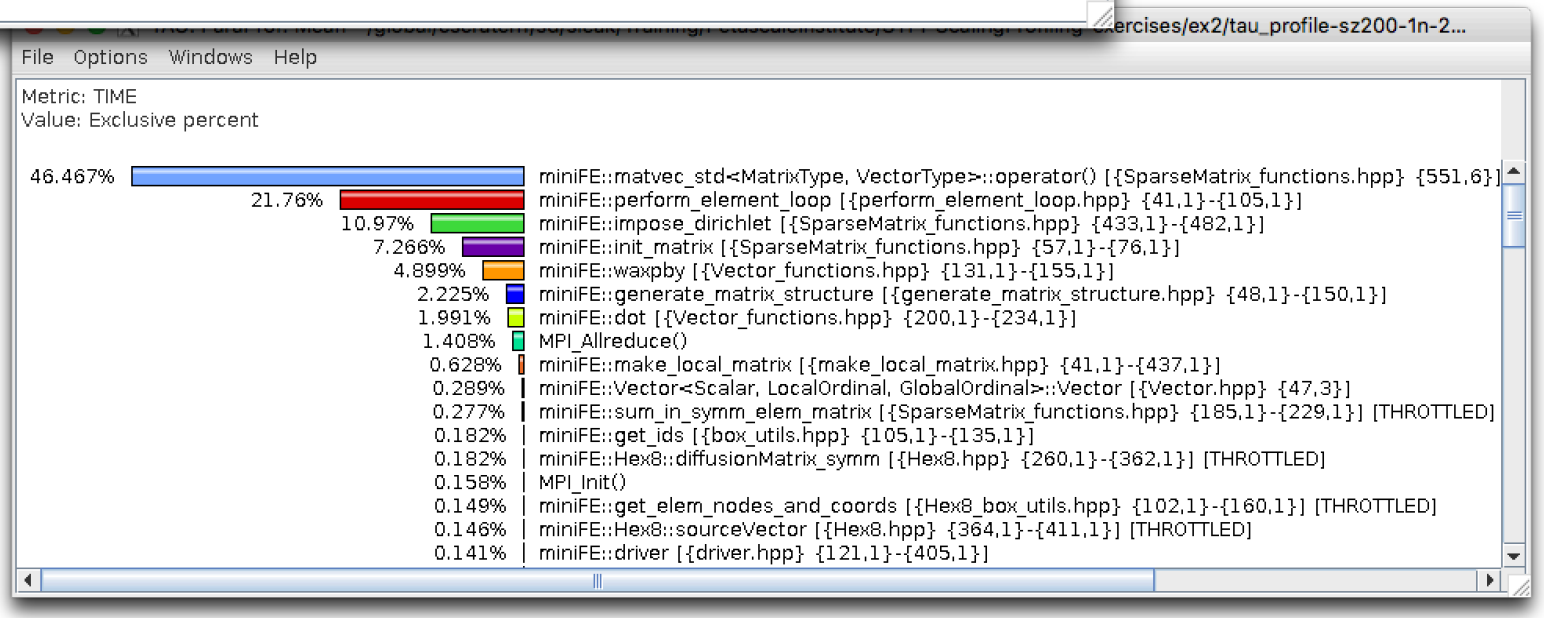
- **After the job runs, you'll see a number of profile.* files in the run directory:**

```
22:44 sleak@cori08:tau_profile-sz200-1n-2mpi-5399296$ ls -lt
total 44
-rw-rw---- 1 sleak sleak  1551 Jun 16 17:20 miniFE.200x200x200.P2.2017:06:16-17:20:19.yaml
-rw-rw---- 1 sleak sleak 20248 Jun 16 17:20 profile.0.0.0
-rw-rw---- 1 sleak sleak 13388 Jun 16 17:20 profile.1.0.0
-rw-rw---- 1 sleak sleak   832 Jun 16 17:20 stdout
-rw-rw---- 1 sleak sleak     0 Jun 16 17:19 stderr
```

- **Make sure TAU is loaded in your environment, and run paraprof**

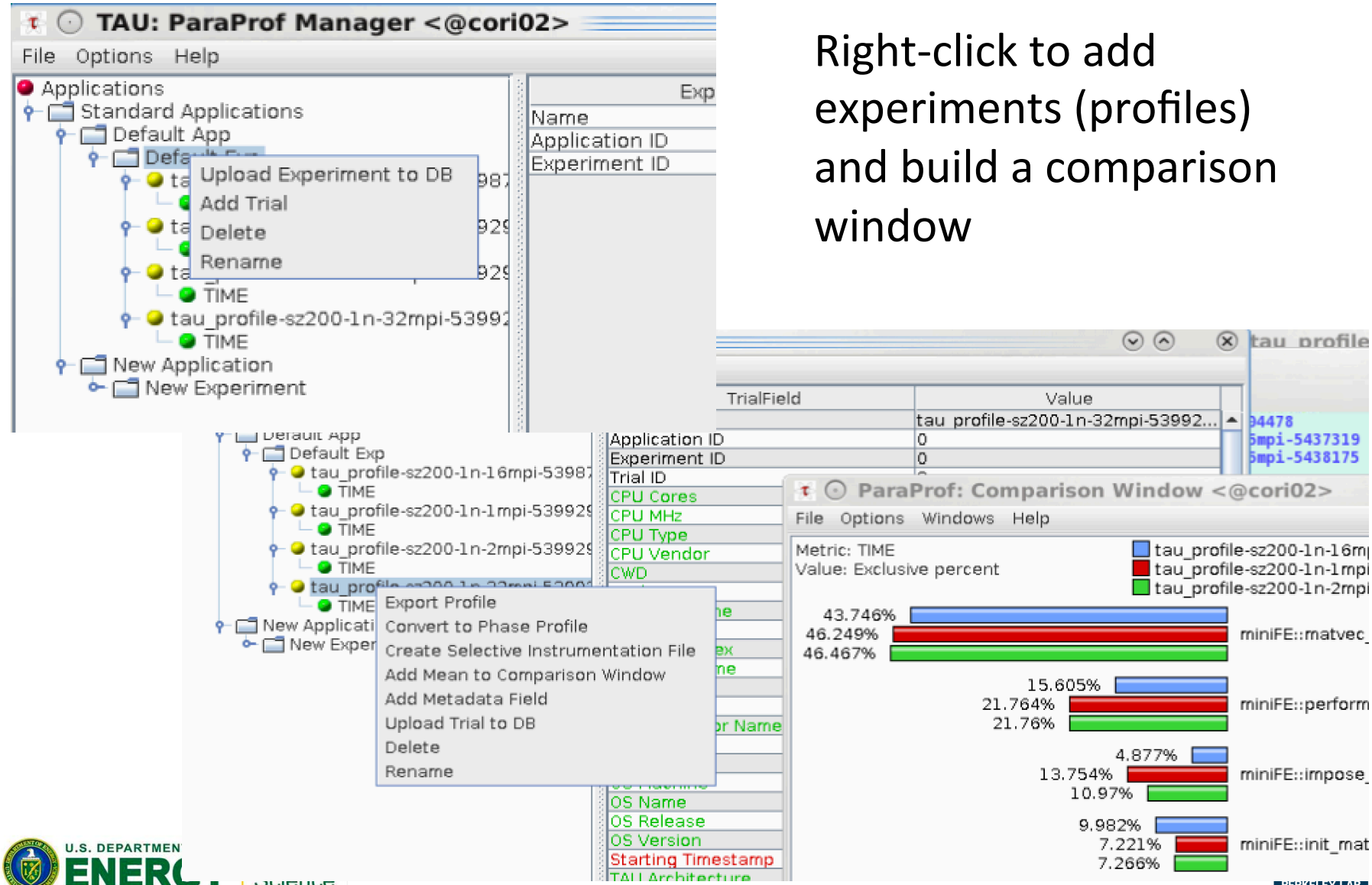
Some paraprof views





Click a bar label to see details:

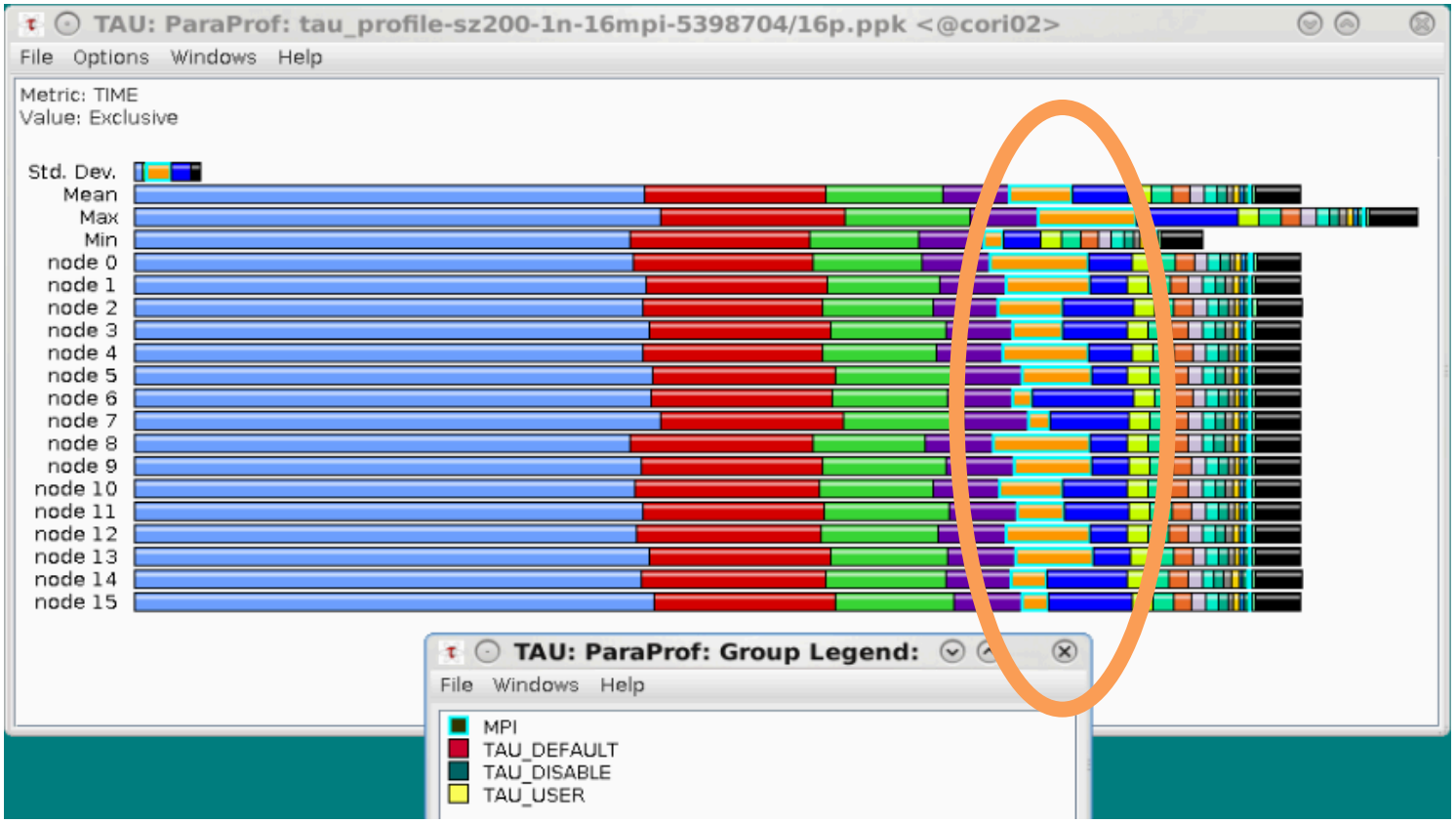
Comparing profiles



Right-click to add experiments (profiles) and build a comparison window

Where are the MPI calls?

- **Windows -> Group Legend allows you to highlight call groups (such as MPI)**



Comparing profiles

- **What does this suggest?**
- **Gotcha: colors don't necessarily correspond to same function in different profiles!**

Non-GUI option: pprof

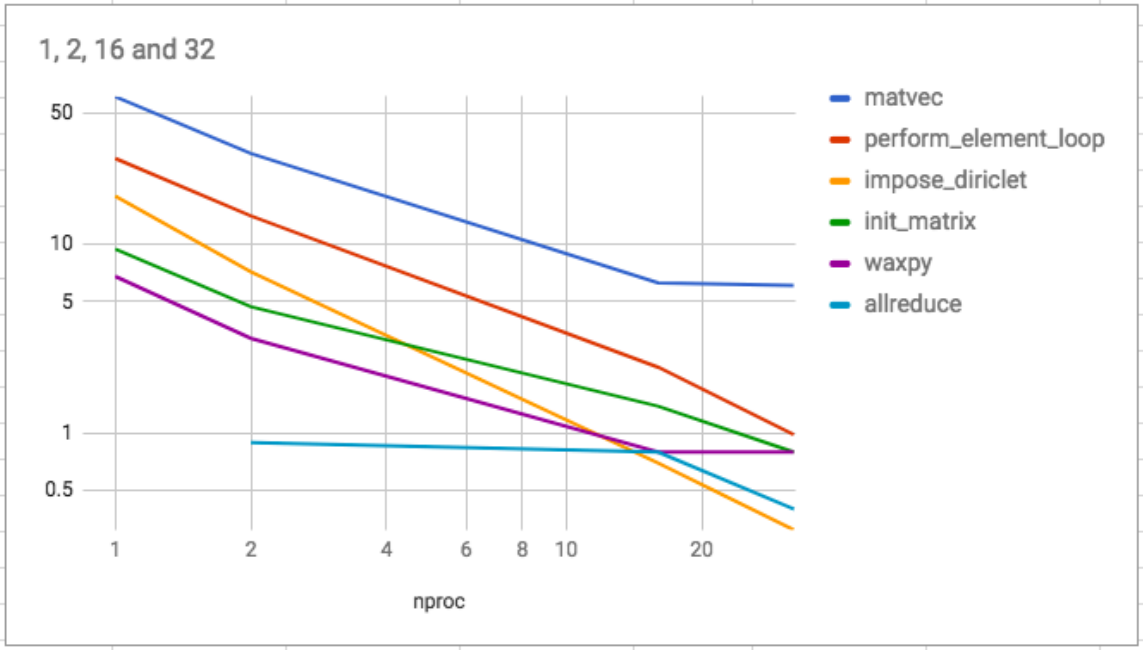
- **"pprof" writes a text-format profile to stdout**
  - Section-per-MPI-rank: might get long!

```
23:00 sleak@cori08:tau_profile-sz200-1n-2mpi-5399296$ pprof -m -p -n5 -s
Reading Profile files in profile.*

FUNCTION SUMMARY (total):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
 46.5        60717        60789         402         402     151217 miniFE::matvec_std<MatrixType
 23.0        28433        30081           2      800000   15040670 miniFE::perform_element_loop
 11.0        14334        14386           4      200002    3596509 miniFE::impose_dirichlet
  7.3         9495         9523           2      200002    4761448 miniFE::init_matrix
  4.9         6402         6402        1204           0       5317 miniFE::waxpby
396.1        11286       517606 5.20423E+06 4.00544E+06         99 -others-

FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
 46.5        30358        30395         201         201     151217 miniFE::matvec_std<MatrixType
 23.0        14217        15041           1      400000   15040670 miniFE::perform_element_loop
 11.0         7167         7193           2      100001    3596509 miniFE::impose_dirichlet
  7.3         4747         4761           1      100001    4761448 miniFE::init_matrix
  4.9         3201         3201         602           0       5317 miniFE::waxpby
396.1         5643       258803 2.60212E+06 2.00272E+06         99 -others-
```

Plotting scaling by routine

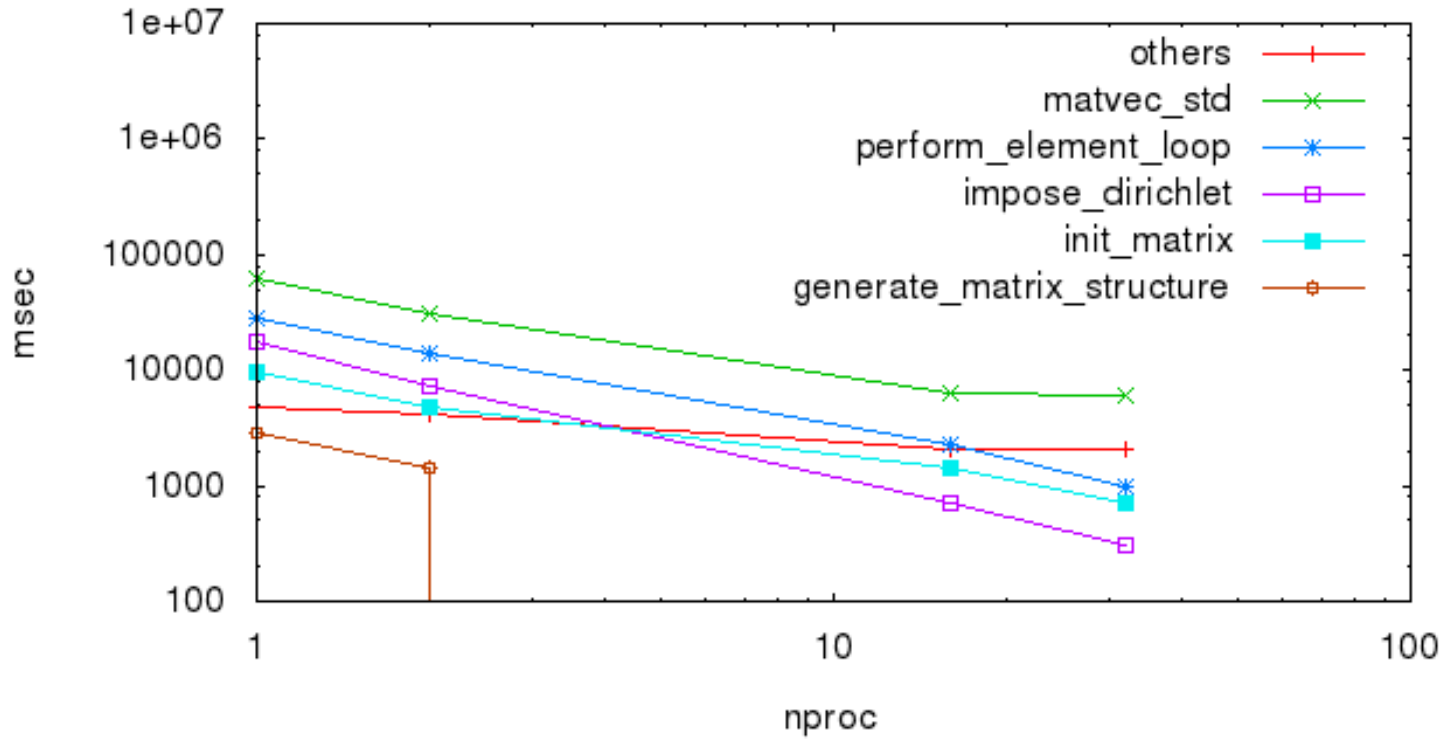
- **We could transcribe timings for individual routines into a spreadsheet and look at how they scale:**

nproc	matvec	perform_element	impose_diriclet	init_matrix	waxpy	allreduce
1	60.7	28.6	18.1	9.5	6.8	0
2	30.4	14.2	7.2	4.7	3.2	0.9
16	6.3	2.25	0.7	1.4	0.8	0.8
32	6.1	0.99	0.31	0.8	0.8	0.4



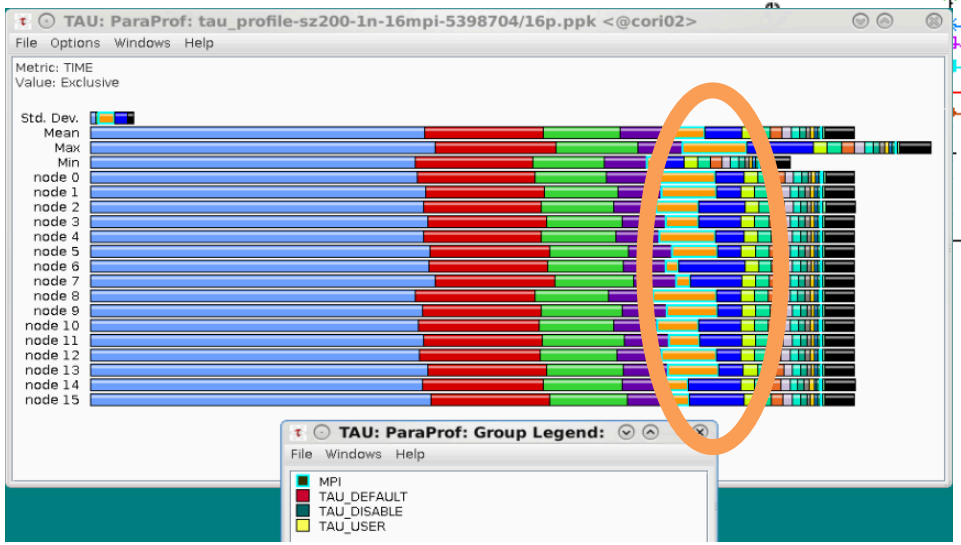
Plotting scaling by routine

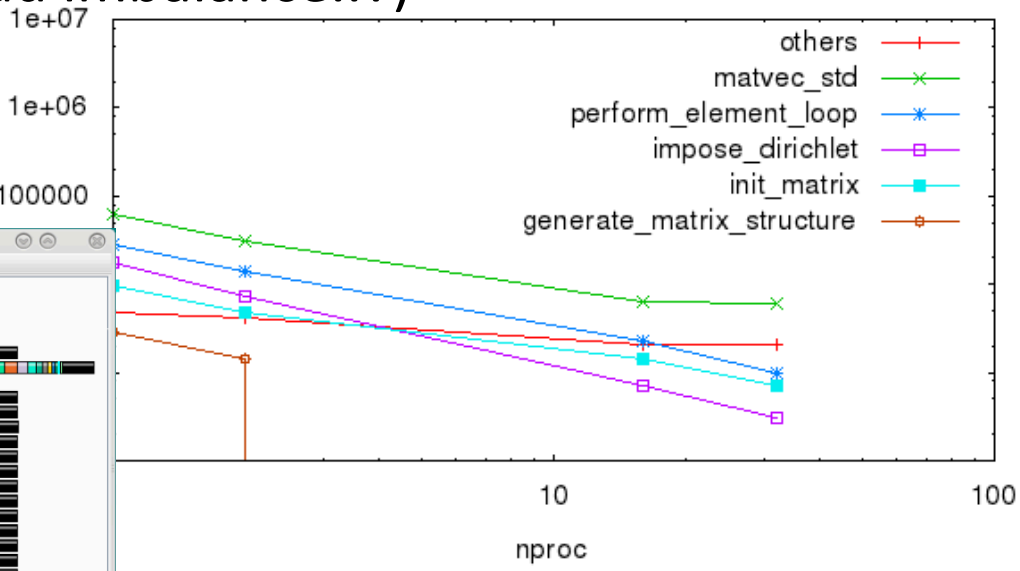
- **.. But we're programmers**
  - python ./parse_pprofs.py … > routine_scaling.dat



Plotting scaling by routine

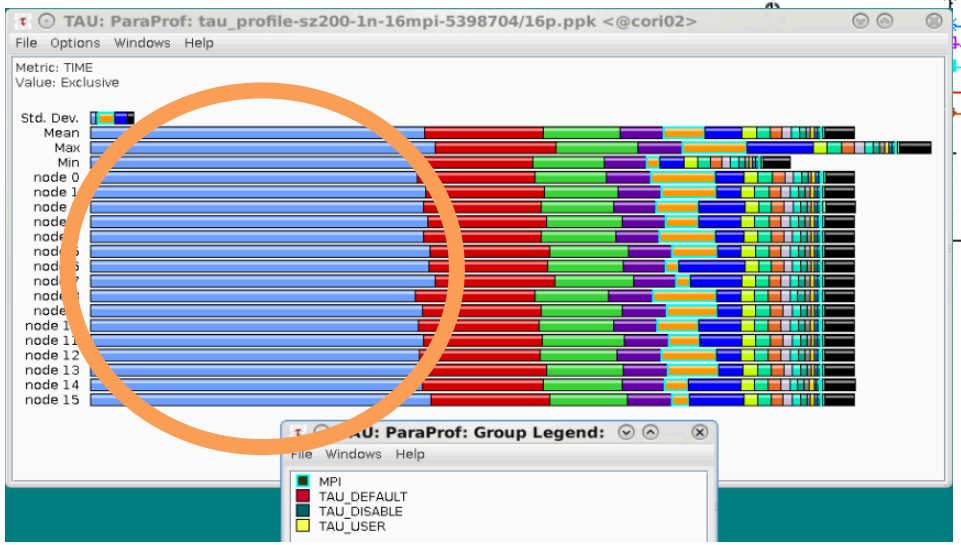
- **Caveat: this is using mean time**
  - impose_dirichlet (gnuplot purple) looks like it scales well!
  - (but remember that load imbalance..?)

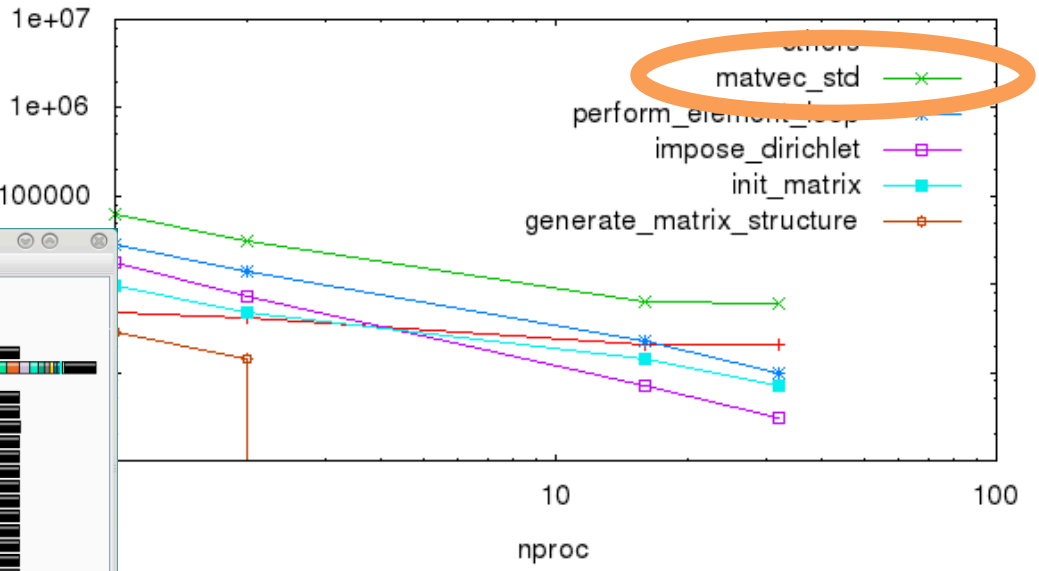




Plotting scaling by routine

- **This one looks like a candidate for a closer look**





Where is the bottleneck?

- **We found a couple of candidates..**
  - Load imbalance in impose_dirichlet
  - Poor scaling in matvec_std

```
void
impose_dirichlet(typename MatrixType::ScalarType prescribed_value,
                 MatrixType& A,
                 VectorType& b,
                 int global_nx,
                 int global_ny,
                 int global_nz,
                 const std::set<typename MatrixType::GlobalOrdinalType>& bc_rows)
{
  typedef typename MatrixType::GlobalOrdinalType GlobalOrdinal;
  typedef typename MatrixType::LocalOrdinalType LocalOrdinal;
  typedef typename MatrixType::ScalarType Scalar;

  GlobalOrdinal first_local_row = A.rows.size()>0 ? A.rows[0] : 0;
  GlobalOrdinal last_local_row  = A.rows.size()>0 ? A.rows[A.rows.size()-1] : -1;

  typename std::set<GlobalOrdinal>::const_iterator
    bc_iter = bc_rows.begin(), bc_end = bc_rows.end();
  for(; bc_iter!=bc_end; ++bc_iter) {
    GlobalOrdinal row = *bc_iter;
    if (row >= first_local_row && row <= last_local_row) {
      size_t local_row = row - first_local_row;
      b.coefs[local_row] = prescribed_value;
      zero_row_and_put_1_on_diagonal(A, row);
    }
  }

  for(size_t i=0; i<A.rows.size(); ++i) {
    GlobalOrdinal row = A.rows[i];

    if (bc_rows.find(row) != bc_rows.end()) continue;

    size_t row_length = 0;
    GlobalOrdinal* cols = NULL;
    Scalar* coefs = NULL;
    A.get_row_pointers(row, row_length, cols, coefs);

    Scalar sum = 0;
    for(size_t j=0; j<row_length; ++j) {
      if (bc_rows.find(cols[j]) != bc_rows.end()) {
        sum += coefs[j];
        coefs[j] = 0;
      }
    }

    b.coefs[i] -= sum*prescribed_value;
  }
}
```

```
//-----------------------------------------------------------------------
//Compute matrix vector product y = A*x where:
//
// A - input matrix
// x - input vector
// y - result vector
//
template<typename MatrixType,
         typename VectorType>
struct matvec_std {
void operator()(MatrixType& A,
                VectorType& x,
                VectorType& y)
{
  exchange_externals(A, x);

  typedef typename MatrixType::ScalarType ScalarType;
  typedef typename MatrixType::GlobalOrdinalType GlobalOrdinalType;
  typedef typename MatrixType::LocalOrdinalType LocalOrdinalType;

  int n = A.rows.size();
  const LocalOrdinalType* Arowoffsets = &A.row_offsets[0];
  const GlobalOrdinalType* Acols      = &A.packed_cols[0];
  const ScalarType* Acoefs            = &A.packed_coefs[0];
  const ScalarType* xcoefs = &x.coefs[0];
        ScalarType* ycoefs = &y.coefs[0];
  ScalarType beta = 0;

  #pragma omp parallel for
  for(int row=0; row<n; ++row) {
    ScalarType sum = beta*ycoefs[row];

    for(LocalOrdinalType i=Arowoffsets[row]; i<Arowoffsets[row+1]; ++i) {
      sum += Acoefs[i]*xcoefs[Acols[i]];
    }

    ycoefs[row] = sum;
  }
}
```

Some caveats…

- **Identifying bottlenecks in real code is complex!**
  - Don't expect to solve poor scaling on the first try
  - Exploration and experimenting is required
- **"Ease of use" is not a strong point of many tools**
  - Case in point: selecting TAU options, application settings for this tutorial took many experiments
  - Familiarity and/or support helps
    - With your application
    - With whichever profiling tool you use (ask your HPC center)
- **Don't be discouraged if your first attempt fails to make the bottleneck clear**

Key ideas

- **Different parts of an application scale differently**
- **Many tools are available to help you find which parts to optimize**
  - Accessibility, familiarity and support are significant considerations
- **Start with a lower-overhead approach, such as profiling or sampling**
  - Can look with higher resolution at key areas later

Q & A

Agenda

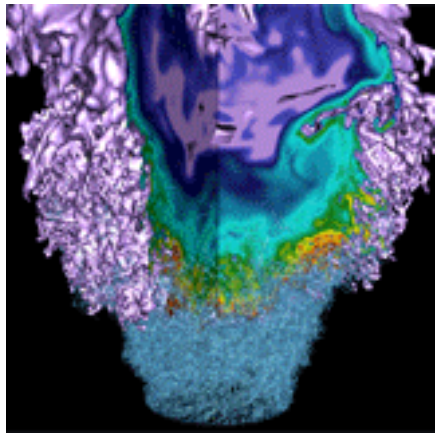
- Amdahl's law
- Where is the bottleneck?
- **Why is *this* a bottleneck?**
- OpenMP scaling

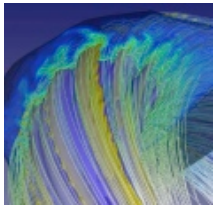
---- 1.45pm PDT: 15 minute break ----

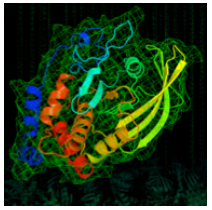
- Weak scaling
- Debugging at scale
- Finding the sweet spot
- Summary and Conclusions

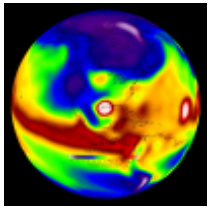
Why is this the bottleneck?

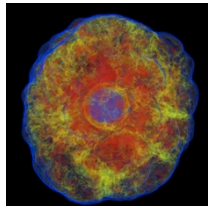
Tracing MPI calls

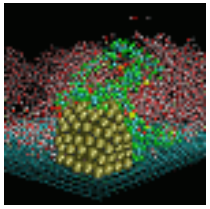






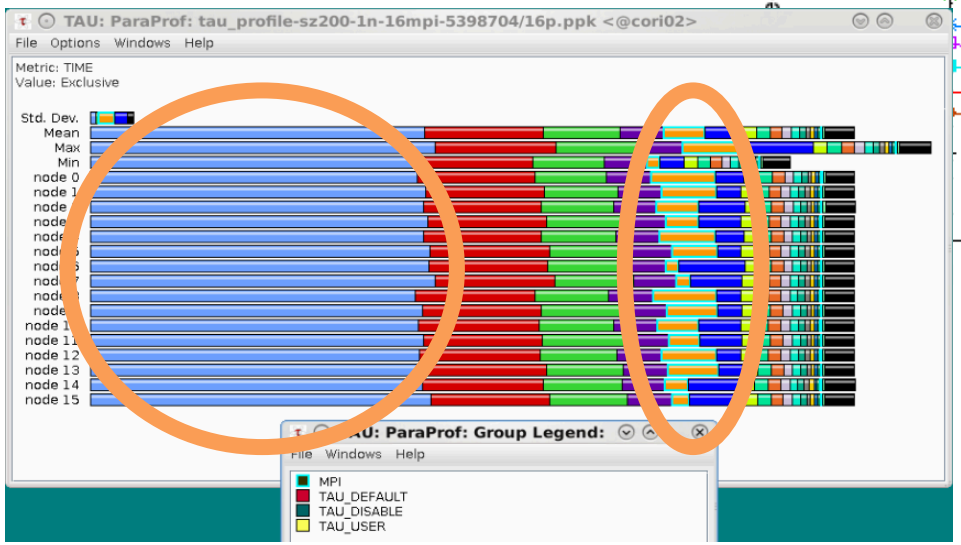


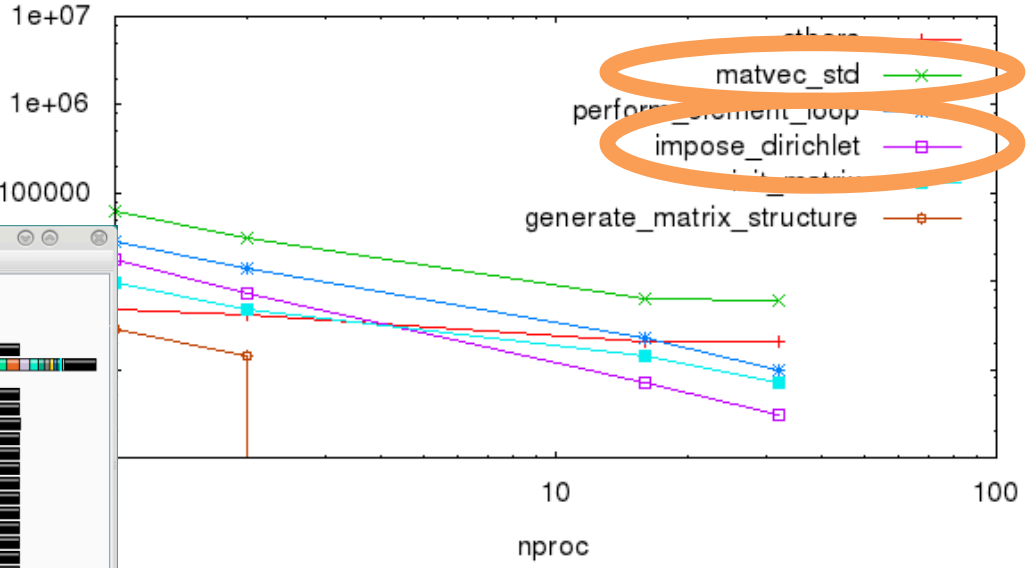




What did the profile tell us?

- **We found a routine that scales poorly, another with load imbalance, and that MPI_Allreduce is (relatively) expensive**





So what next?

- **Poorly-scaling routine**
  - not necessarily the same problem as a poorly-*performing* routine!
  - Look for regions in the routine where current decomposition doesn't reduce work/iteration count
  - Consider manually instrumenting sections of the routine to identify poorly scaling parts

So what next?

- **Load imbalance within routine**
  - Is it actually a problem?
    - Does it lead to spin waiting / load imbalance in an MPI call later?
    - (Maybe a different routine counter-balances it)
  - Identify MPI calls showing cost of imbalance
    - (or OpenMP barriers)

So what next?

- **Expensive MPI calls (why is it expensive?)**
  - Load imbalance? ..probably not due to the MPI call itself, but in a routine before it
    - Maybe non-blocking calls can reduce syncronization
  - High call count / latency dominated?
    - Consider MPI datatypes or packing messages into fewer calls
  - Collectives? .. Cost likely increases with process count
    - Can you use hints to relax ordering requirements?
- **Timeline-view of processes (tracing) around MPI calls can help diagnosis**

Tracing caveats

- **Very high overhead!**
  - Capturing timestamp of every event – high run-time overhead
  - Storing timestamp of every event, on every process – large trace files
- **Too much information**
  - Like looking through a microscope: great for biology, not so great for geography
- **Identify what you need to inspect, and just trace that**
  - Pause/resume API calls, if available

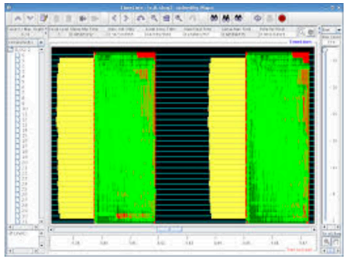
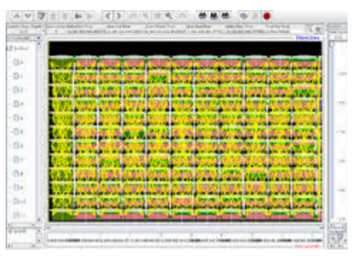
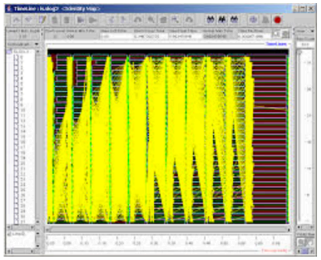
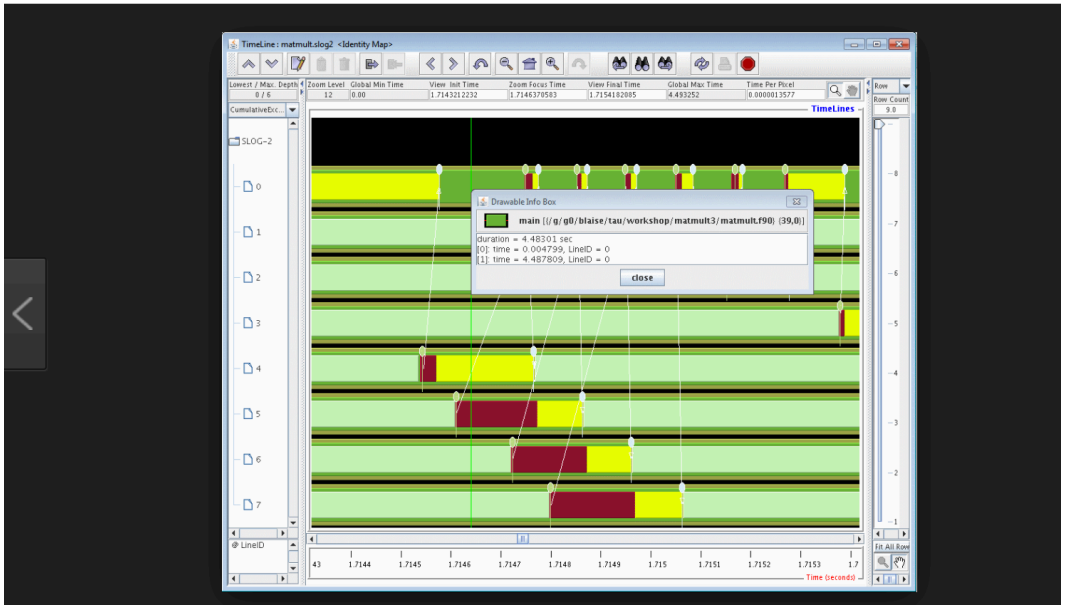
Hands-on exercise

- See ex3-tracing/README.rst
- In this exercise we'll run our TAU-instrumented executable with tracing enabled, and use jumpshot to explore the trace
- Jumpshot is X-intensive, so logging in via NX is recommended

- 5-10 minutes, and then we'll look at some results

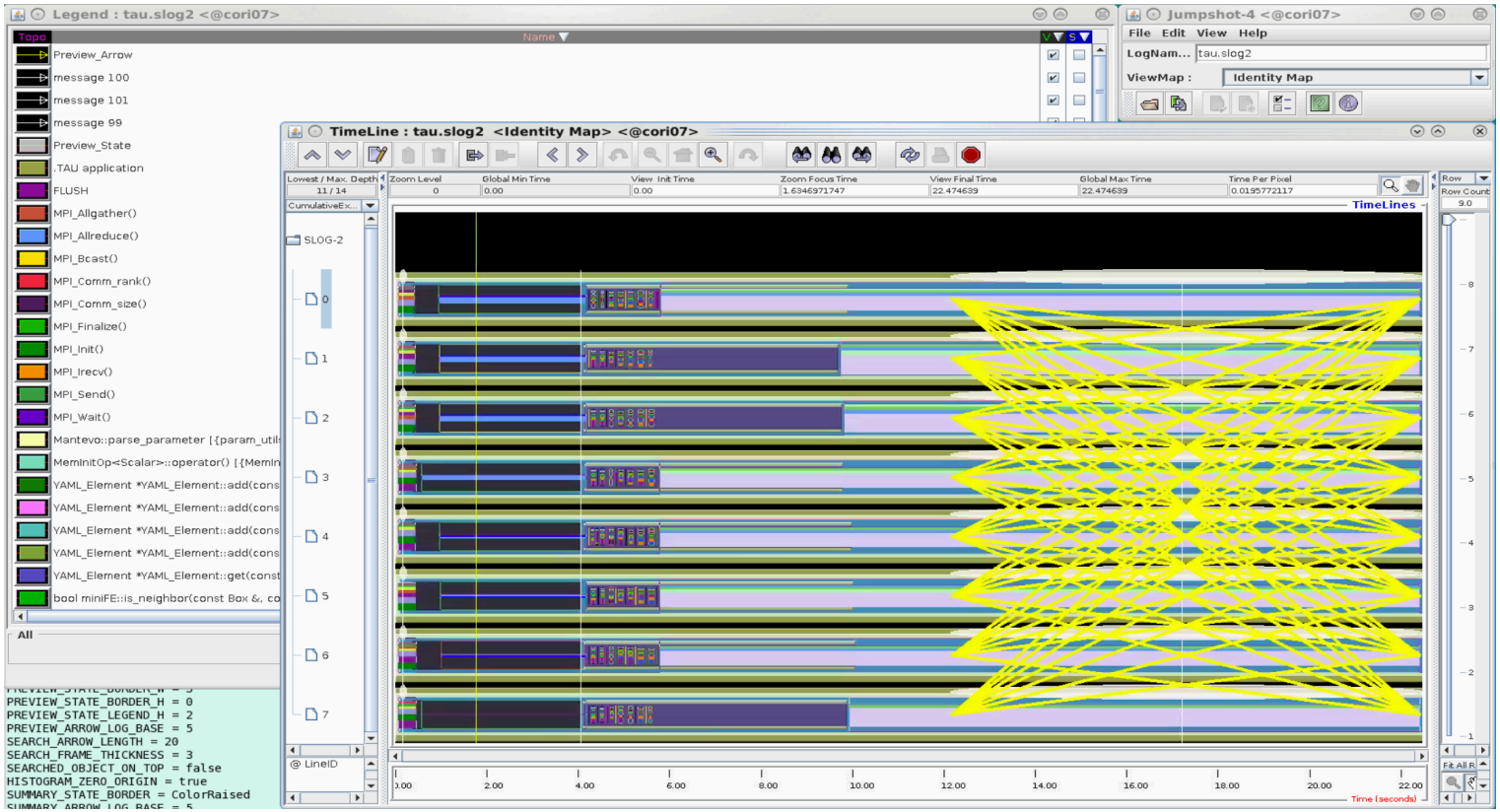
Tracing / Jumpshot first look

- **Google image search for “jumpshot” returns a lot of images like this:**
  - Looks nice and comprehensible
  - Processes, messages

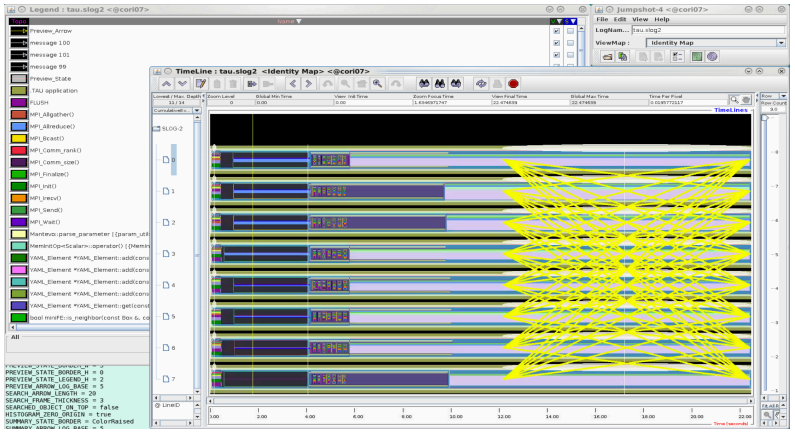





Tracing / Jumpshot first look

- .. And then when you opened it, you probably saw something like this:



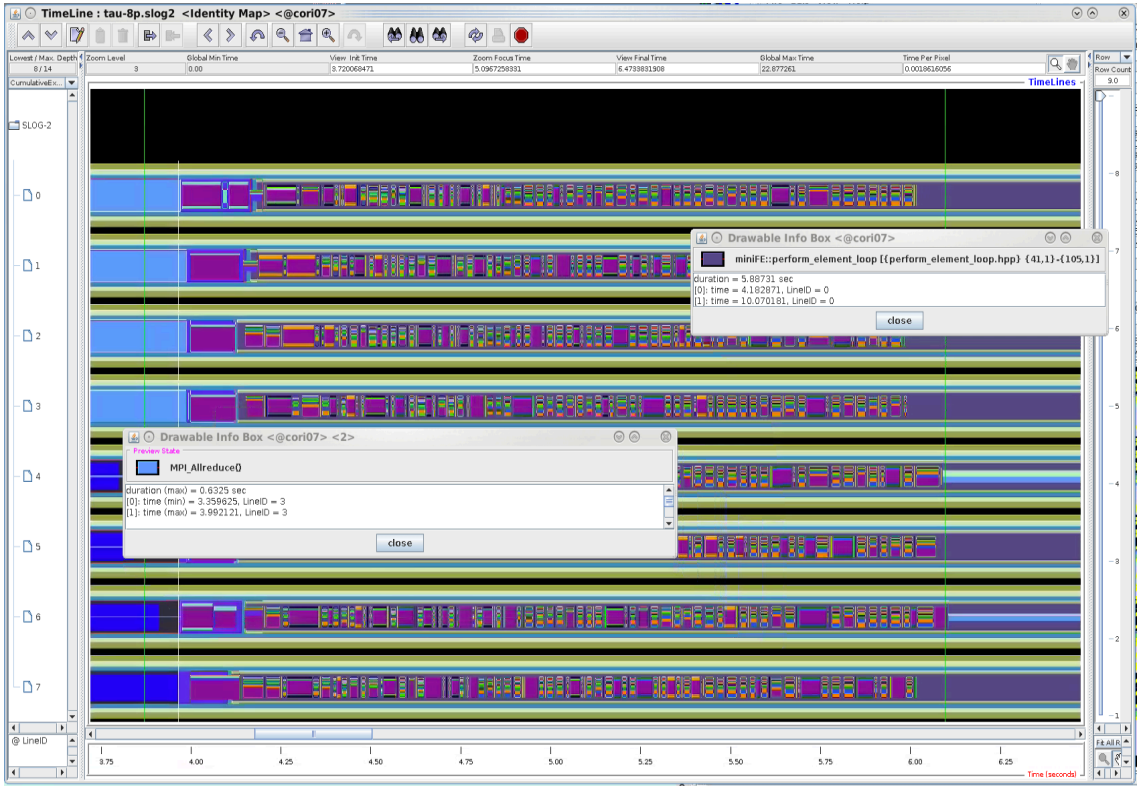
Understanding Jumpshot



- There's a lot going on here!
- Jumpshot uses "Preview States" and "Preview Arrows" to indicate regions with too many messages or events to draw
- Mouse to zoom in to see detail
- "v" column in Legend window toggles visibility of items, can use it to manage information overload

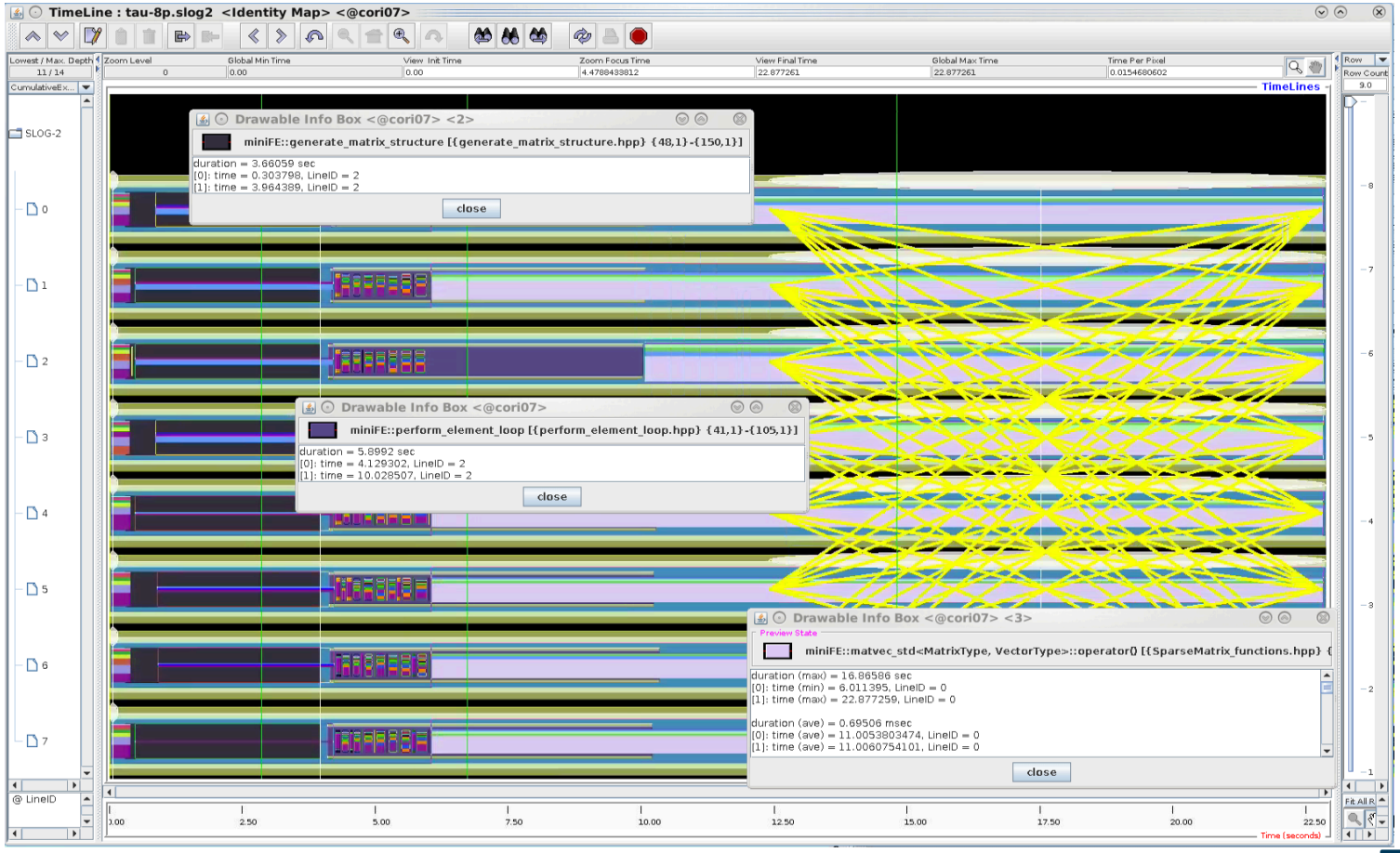
Understanding Jumpshot

- **Looking closer:**
  - Each major bar is one process
  - Stripes within bars represent call tree (stripe size indicates time relative to time in caller)
  - Light blue here is MPI_Allreduce



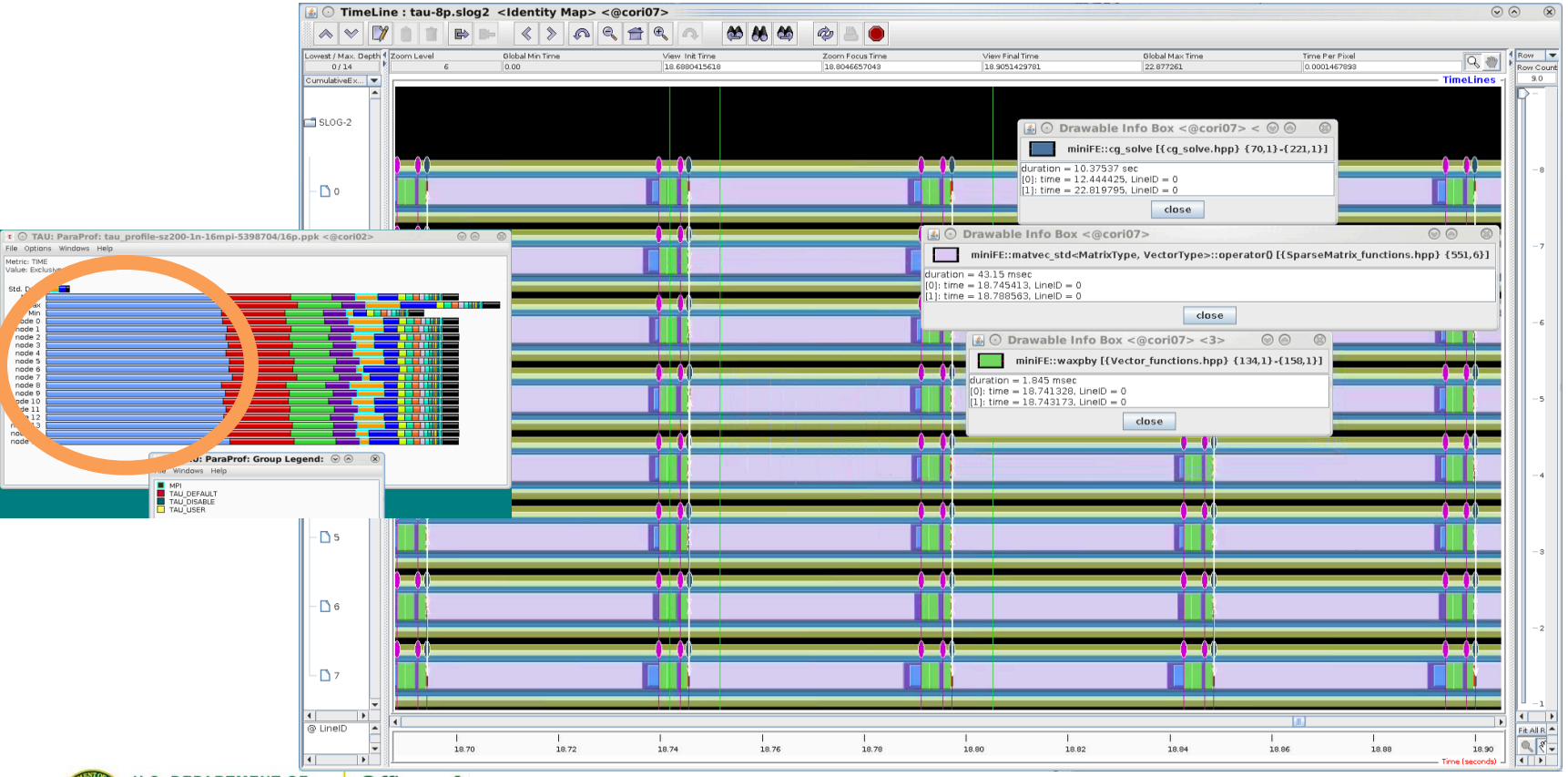
Understanding Jumpshot

- **Much of load imbalance appears to be startup**



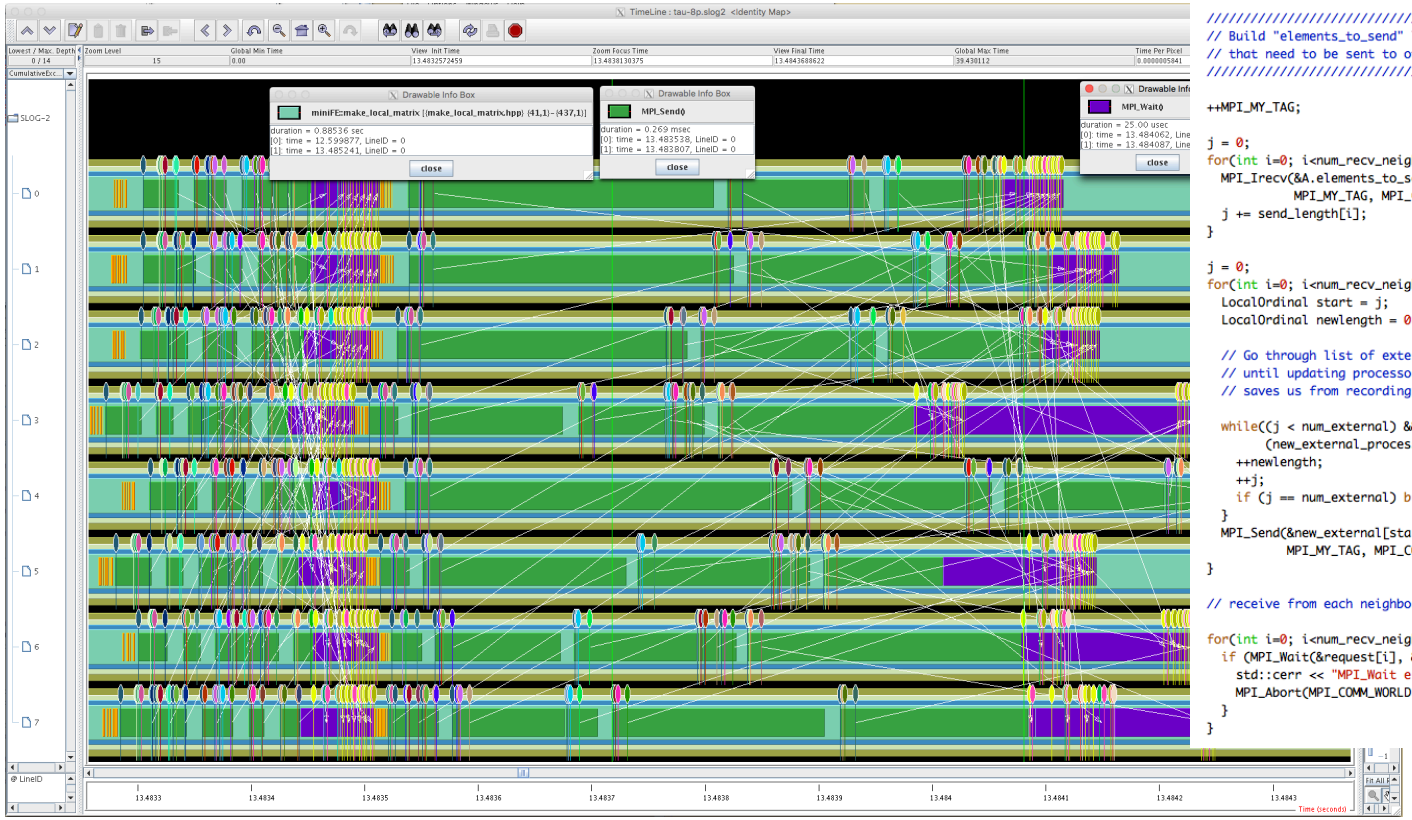
Understanding Jumpshot

- Zoom in on the preview area, can see matvec_std dominating cg_solve iterations



Understanding Jumpshot

- **Another example:**
  - Many send/irecv – can ordering be relaxed?




```
//////////////////////////////////////////////////////////////////////
// Build "elements_to_send" list. These are the x elements I own
// that need to be sent to other processors.
//////////////////////////////////////////////////////////////////////

++MPI_MY_TAG;

j = 0;
for(int i=0; i<num_recv_neighbors; ++i) {
  MPI_Irecv(&A.elements_to_send[j], send_length[i], mpi_dtype, neighbors[i],
            MPI_MY_TAG, MPI_COMM_WORLD, &request[i]);
  j += send_length[i];
}

j = 0;
for(int i=0; i<num_recv_neighbors; ++i) {
  LocalOrdinal start = j;
  LocalOrdinal newlength = 0;

  // Go through list of external elements
  // until updating processor changes. This is redundant, but
  // saves us from recording this information.

  while((j < num_external) &&
        (new_external_processor[j] == recv_list[i])) {
    ++newlength;
    ++j;
    if (j == num_external) break;
  }
  MPI_Send(&new_external[start], j-start, mpi_dtype, recv_list[i],
           MPI_MY_TAG, MPI_COMM_WORLD);
}

// receive from each neighbor the global index list of external elements

for(int i=0; i<num_recv_neighbors; ++i) {
  if (MPI_Wait(&request[i], &status) != MPI_SUCCESS) {
    std::cerr << "MPI_Wait error\n"<<std::endl;
    MPI_Abort(MPI_COMM_WORLD, -1);
  }
}
```

Key ideas

- **Tracing is analogous to viewing with a microscope**
- **Some time, patience required!**
  - Locate the routines identified via profiling, examine behavior
- **In this case:**
  - We saw a lot of MPI_Allreduce imbalance in startup region, before CG solver begins
  - CG solver itself looks balanced, but has frequent communication overhead

Q & A

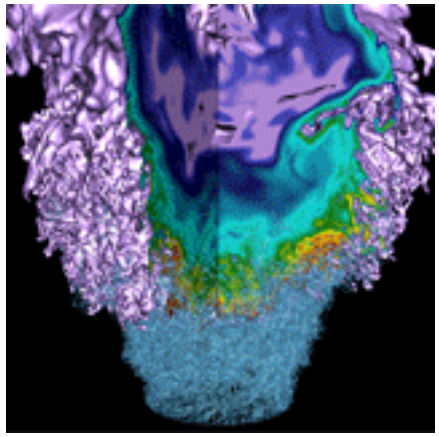
Agenda

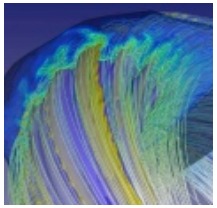
- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- **OpenMP scaling**

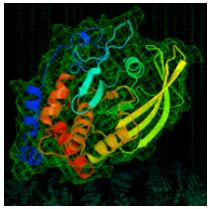
  ---- 1.45pm PDT: 15 minute break ----

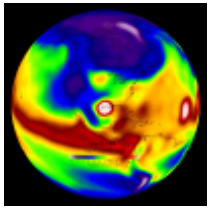
- Weak scaling
- Debugging at scale
- Finding the sweet spot
- Summary and Conclusions

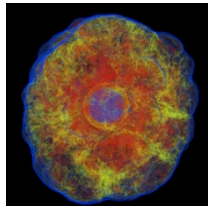
OpenMP scaling
What limits OpenMP scalability?

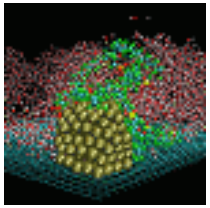






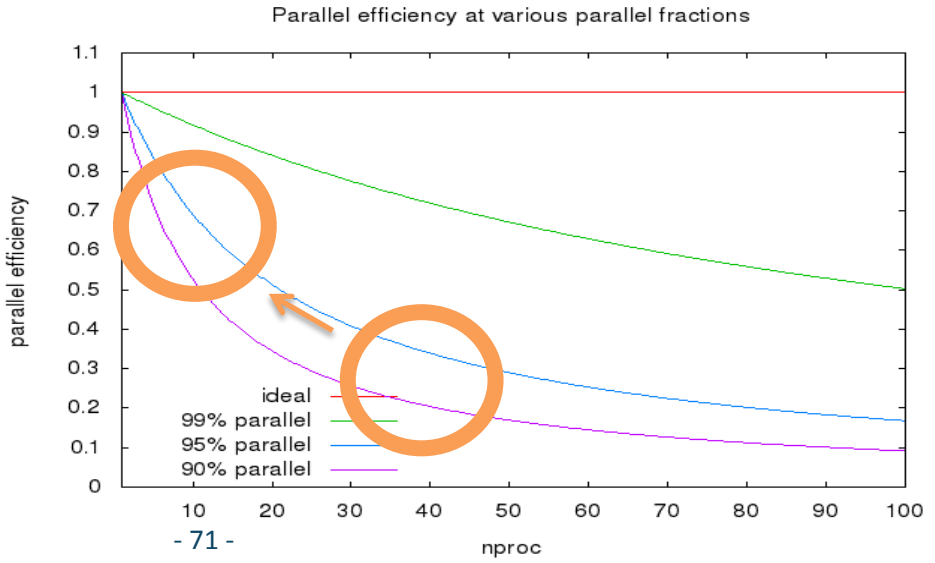






Why OpenMP?

- **Memory constraints**
- **Expensive MPI calls**
  - MPI communication == overhead
  - Reducing process count reduces communication
  - OpenMP: can use same #cores with smaller #ranks
    - But introduced OpenMP overheads must be smaller than removed MPI overheads!



Scaling limiters in OpenMP vs MPI

- **MPI**
  - Serial component
  - Load imbalance
  - **Communication:**
  - Latency
  - Collective operations
- **OpenMP**
  - Serial component
  - Load imbalance
  - Thread startup/cleanup
  - Memory affinity
  - **Resource Sharing:**
  - Cache lines
  - Data (Critical sections)

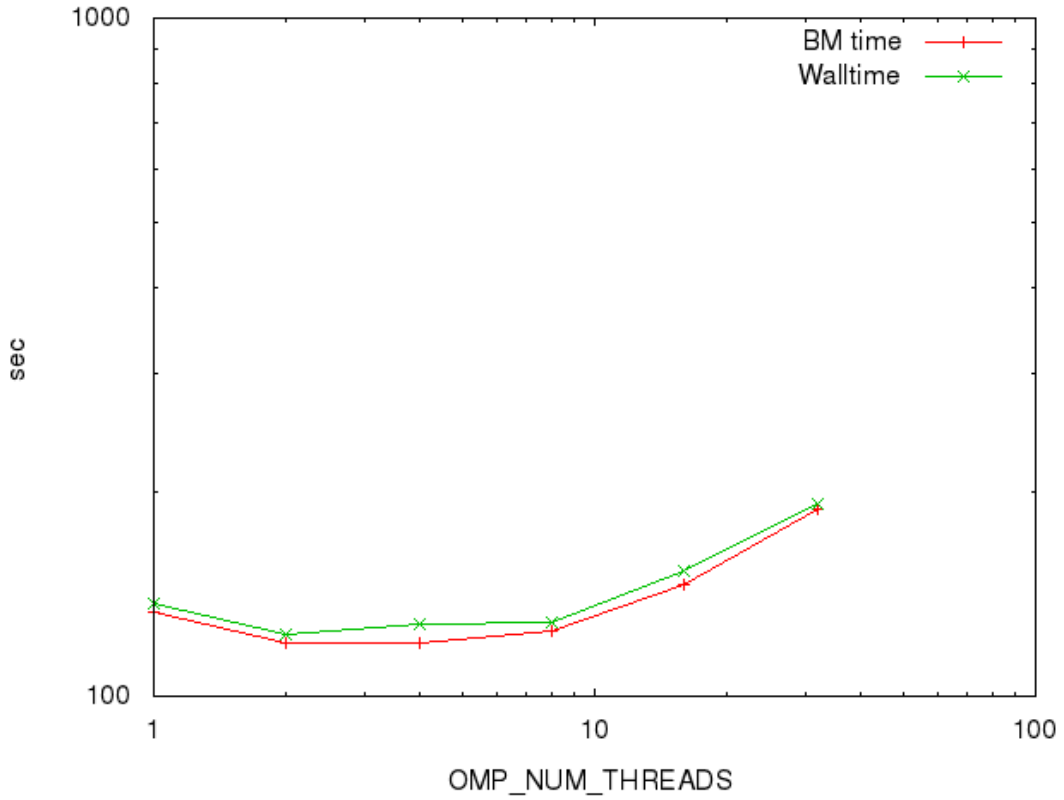
Hands-on exercise

- **See ex4-openmp/README.rst**
- **In this exercise we will build miniFE with OpenMP enabled, measure how it scales and use TAU to identify areas where we might improve OpenMP scaling**
  - For now, OpenMP only (no MPI / 1 rank)
- **5-10 minutes, then we'll look at some results**

- **(jobs may take a little long, but can look at first few profiles while later runs still in progress)**

Exercise 4

- **Some scaling issues here!**
  - We get some benefit with 2 threads, but increasing threads further only makes performance worse

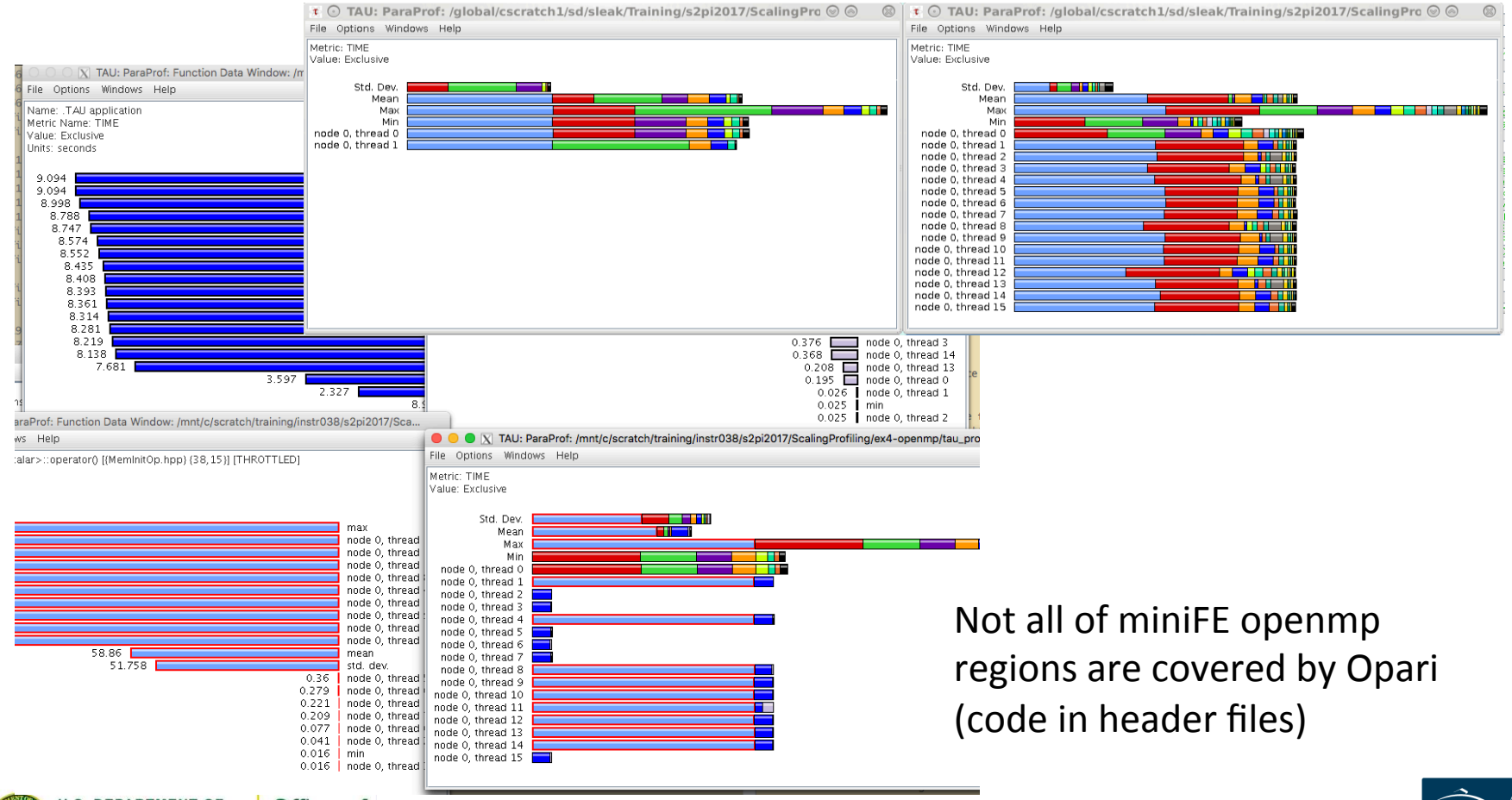


Exercise 4 - Paraprof

- **OpenMP vs MPI profiling**
  - MPI has PMPI_* interface
    - Each MPI routine has well-defined hooks for instrumentation
  - No OpenMP equivalent
    - Tools for OpenMP profiling have suffered from this lack of support
  - Some progress: OMPT interface
    - Not fully/widely supported (some support in Intel compiler)
- **TAU on Cori vs Blue Waters**
  - Cori version has Makefiles using OMPT interface, Blue Waters version uses Opari

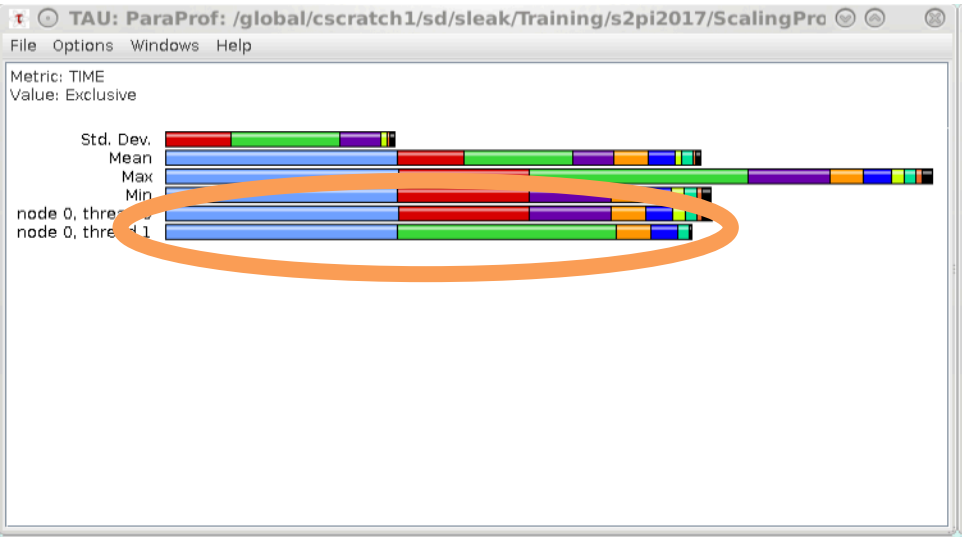
Exercise 4 - Paraprof

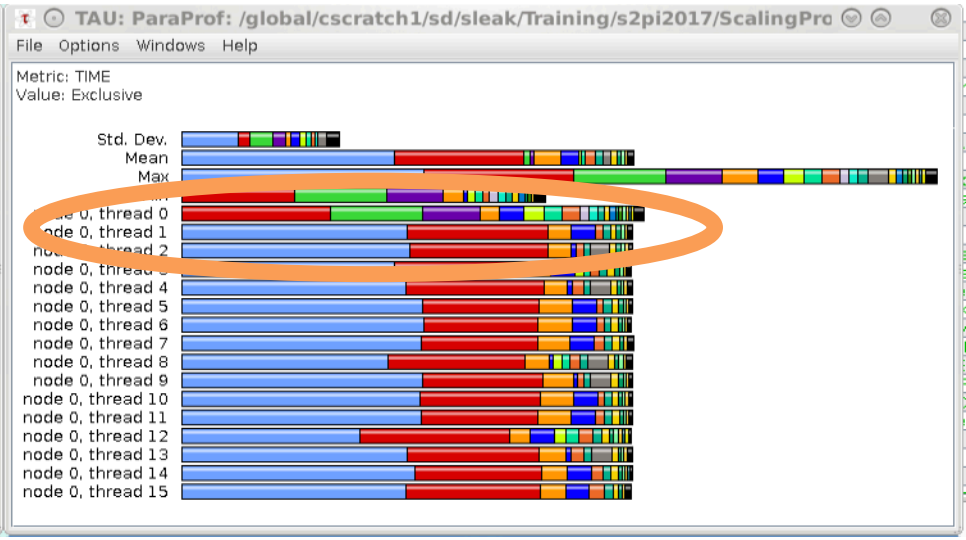
- **Caveat: Cori with OMPT vs BW with Opari**



Not all of miniFE openmp regions are covered by Opari (code in header files)

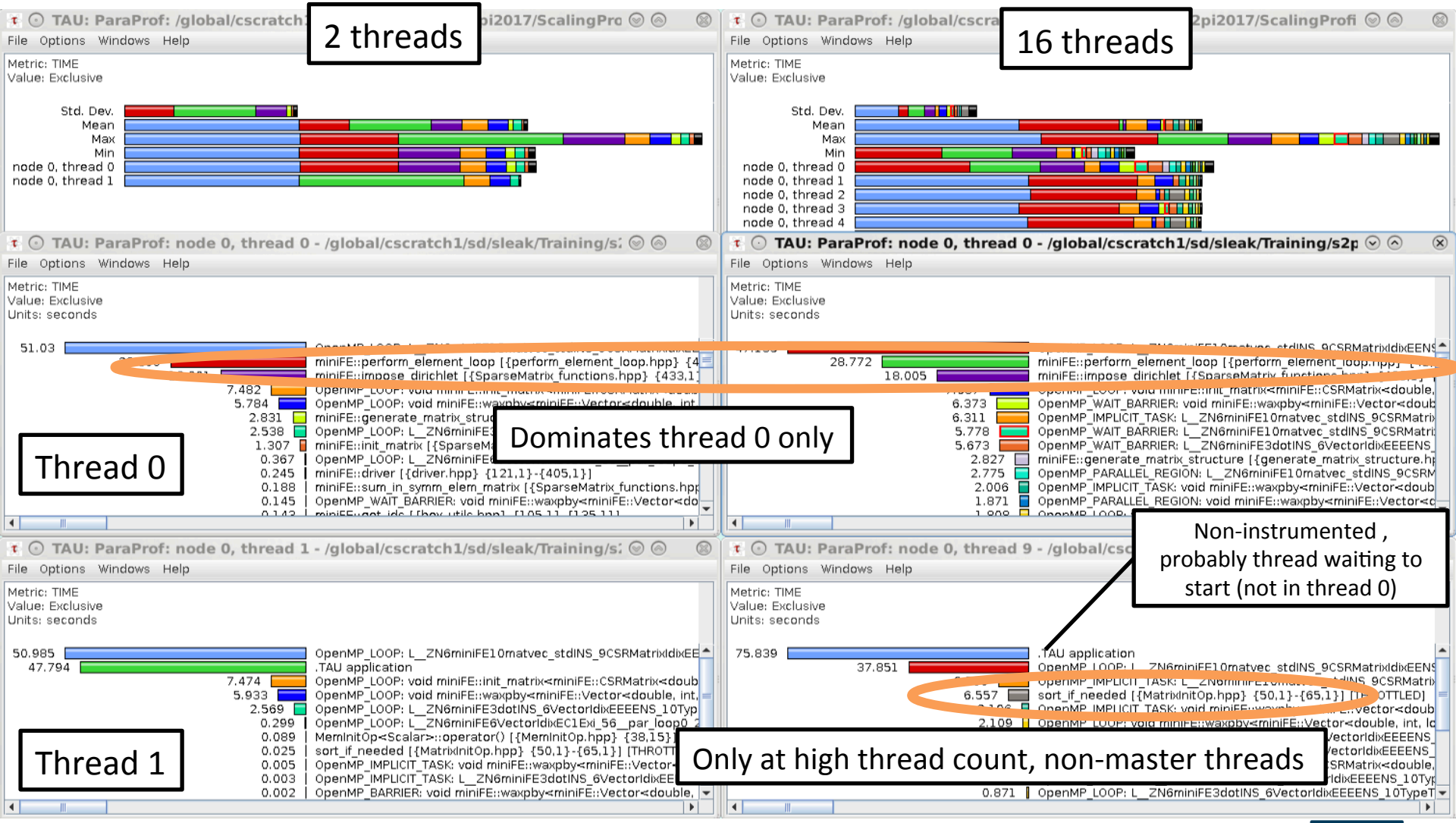
Exercise 4 - Paraprof





- **Some distinct differences between thread 0 and others!**
  - Suggests lack of coverage – large serial component

Exercise 4 - Paraprof



Exercise 4 – what did we find?

- **Lack of coverage probably biggest issue**
  - Identified a couple of routines that appear to be outside of OpenMP regions
- **Routine "sort_if_needed" scales poorly**

- **Remember: profiling only shows *where*!**
  - Need to look at routines themselves, maybe detailed performance profiling, to understand *why*

Q & A

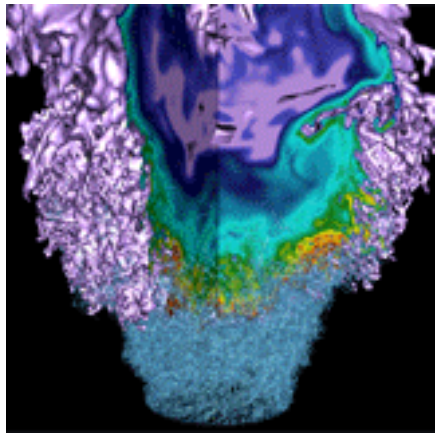
Agenda

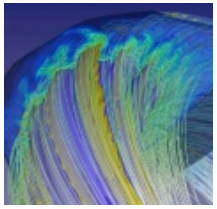
- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- OpenMP scaling
- **Before the break: look at ex5-weakscaling, submit job script**
  - Longer-running job, we'll use the break to give it a head-start

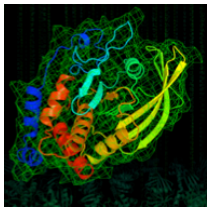
---- 1.45pm PDT: 15 minute break ----

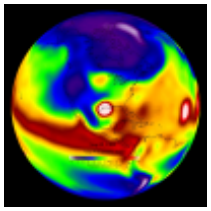
- Weak scaling
- Debugging at scale
- Finding the sweet spot
- Summary and Conclusions

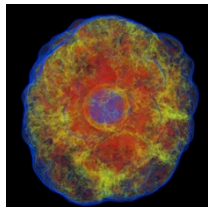
Weak scaling
The loophole in Amdahl's law

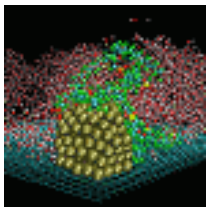










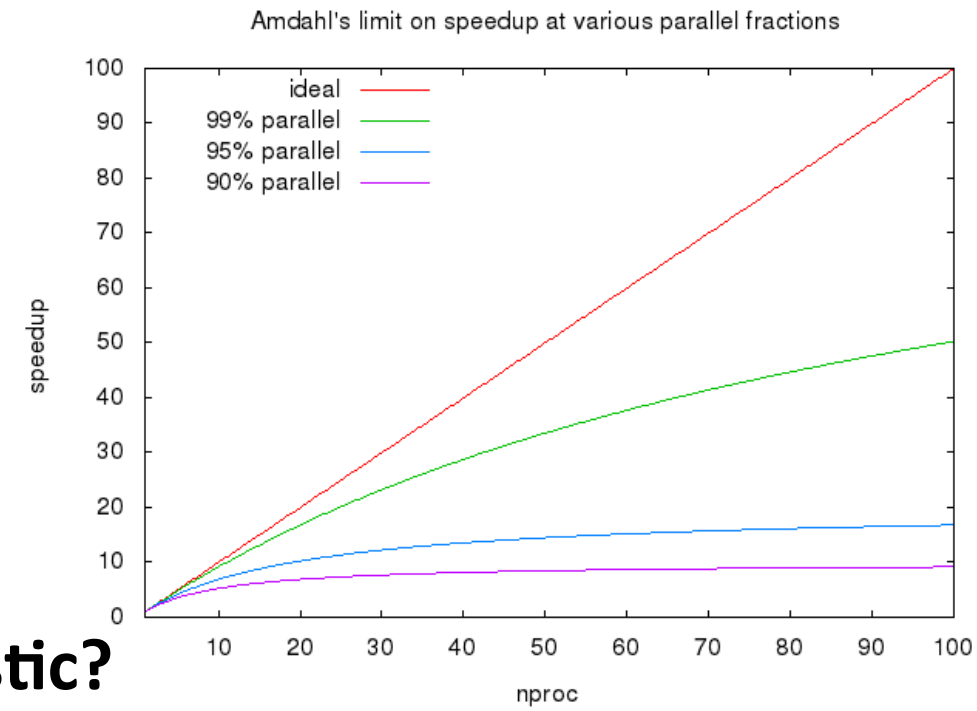


The problem with Amdahl's law

- **How to get 1000x speedup?**

$$Speedup \leq \frac{1}{\frac{p}{n} + (1-p)}$$

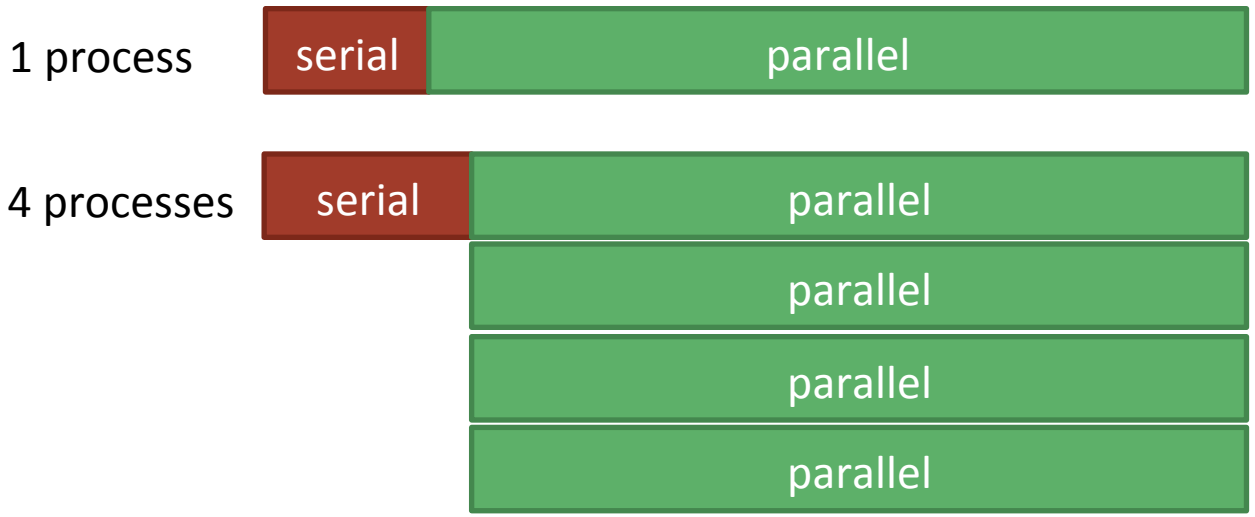
$$Speedup_{p \Rightarrow \infty} \leq \frac{1}{1-p}$$



- **Is 99.9% parallel realistic?**

Gustafson's law

- **The problem you solve with 10 processors is not the same as the problem you solve with 100 processors**
  - Fixed time, not fixed work
  - Increasing problem sizes mostly increase parallel part



Terminology

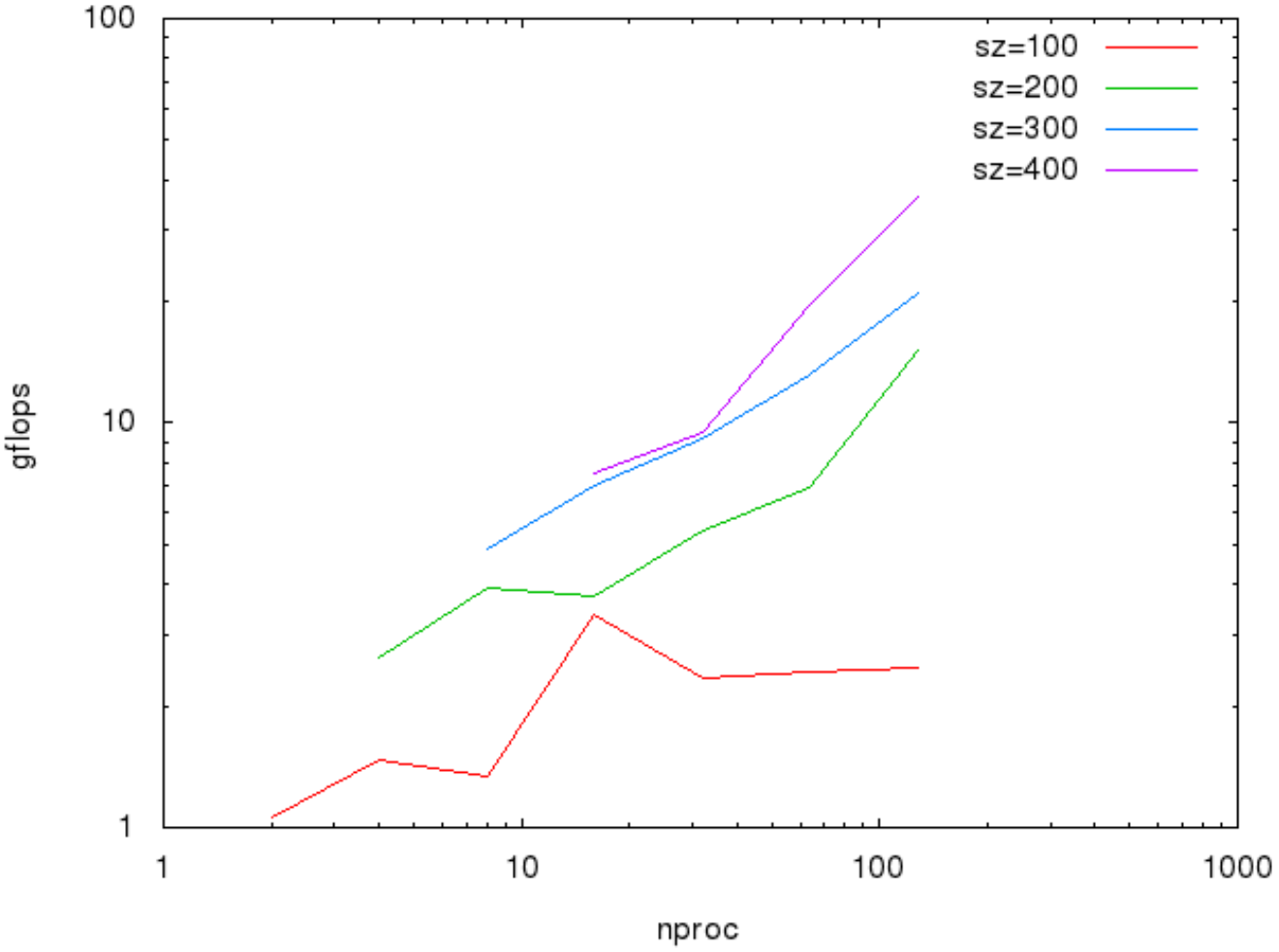
- **"Strong scaling"**
  - How well does application scale for fixed problem size?
- **"Weak scaling"**
  - How well does application scale when problem size is increased proportional to process count?

Hands-on exercise

- **See ex5-weakscaling/README.rst**
- **In this exercise we will run miniFE at a few different problem sizes, and a few different node counts**
  - Hopefully, some jobs have already run, and you can start plotting results
- **Plotting results:**
  - Time-to-solution is no longer a good comparison
  - Calculate work (Flops) and plot speed instead

- **5-10 minutes, then we'll look at some results**

Exercise 5 – Weak scaling



Key ideas

- **Amdahl's law is not a show-stopper**
  - The problems that most need extreme parallelism are the same problems best able to use it

Q & A

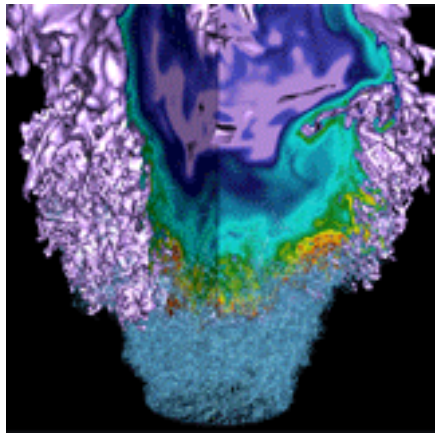
Agenda

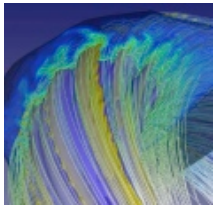
- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- OpenMP scaling

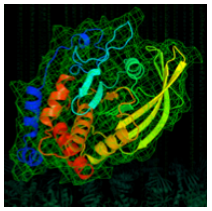
---- 1.45pm PDT: 15 minute break ----

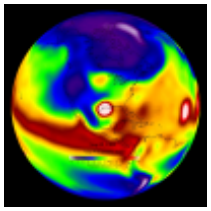
- Weak scaling
- **Debugging at scale**
- Finding the sweet spot
- Summary and Conclusions

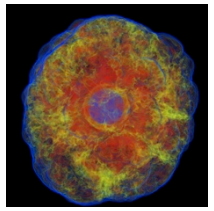
Debugging at scale

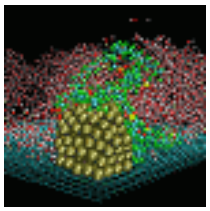












Debugging at scale

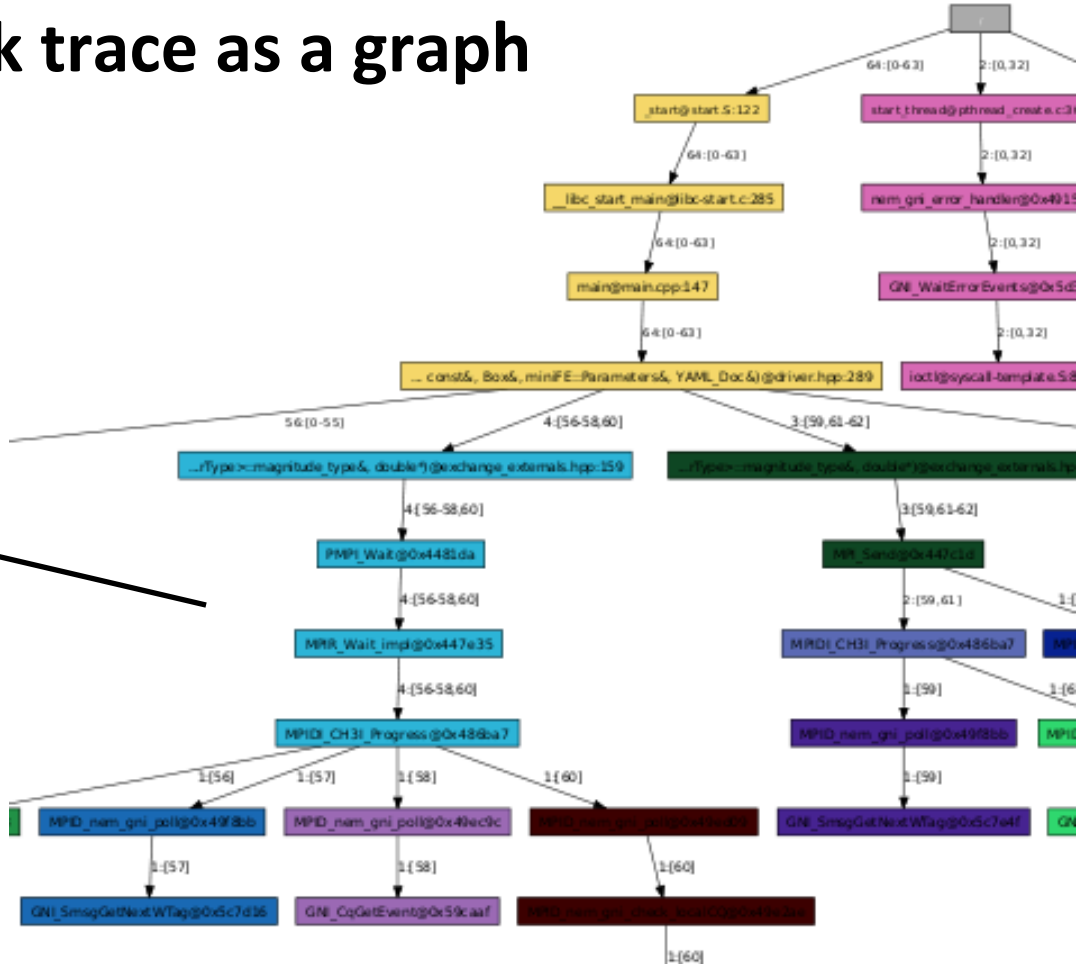
- **Debugging is hard!**
  - Even without parallelism
- **Parallel debugging is harder**
  - Additional error types (race conditions, deadlocks, numerical/order-of-operations problems)
  - Multiple processes/threads
    - Which to attach a debugger to?
  - Traditional methods are problematic
    - Multiple debuggers?
    - printf from dozens of processes?
    - Dump array to file – how many files?

Parallel debugging tools

- **Manage a set of processes, present single interface**
- **Full-featured commercial offerings:**
  - DDT, Totalview
  - Heavyweight, capable
- **Open Source:**
  - STAT "Stack Trace Analysis Tool"
  - Lightweight - Not a full-featured debugger
  - Captures stack trace for all processes in parallel application, draws tree of where each process is
  - **Finding the location of the error is often half of the solution!**

STAT

- **Parallel stack trace as a graph**

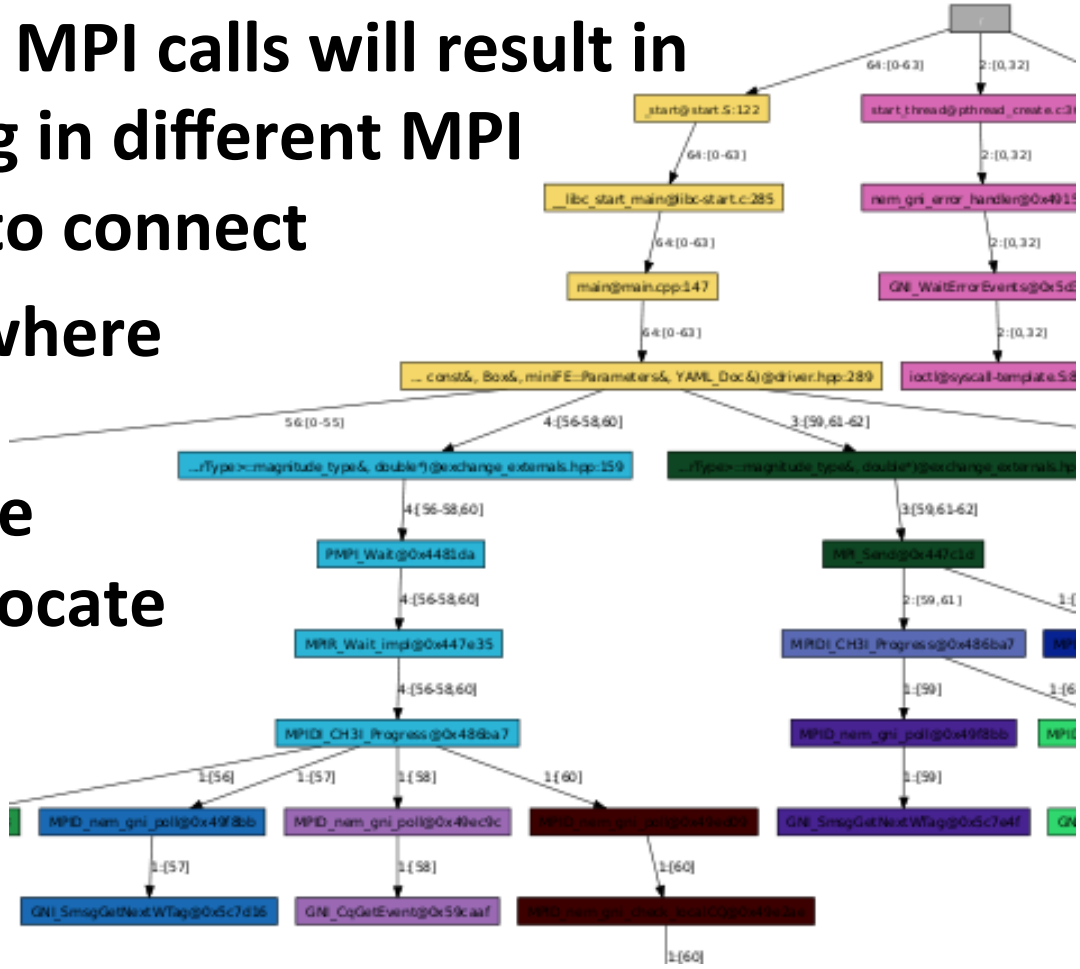


Most processes had this part of stack in common

Function each process was in

How does this help?

- **Mismatch in MPI calls will result in tasks waiting in different MPI calls, never to connect**
- **Identifying where different processes are stuck helps locate the bug**



ATP

- **In hands-on exercise we'll use ATP, part of Cray tool suite**
  - Invokes STAT on application crash
- **Two modes of use:**
  - If the application hangs, you can trigger ATP by sending a signal to all processes (via Slurm "scancel –s ABRT" or ALPS "apkill")
    - Roughly corresponds with STAT native use: attach to hung parallel job to capture stack traces
  - If the application crashes, ATP automatically captures the stack traces
- **Use STAT stat-view to view results**

Using ATP

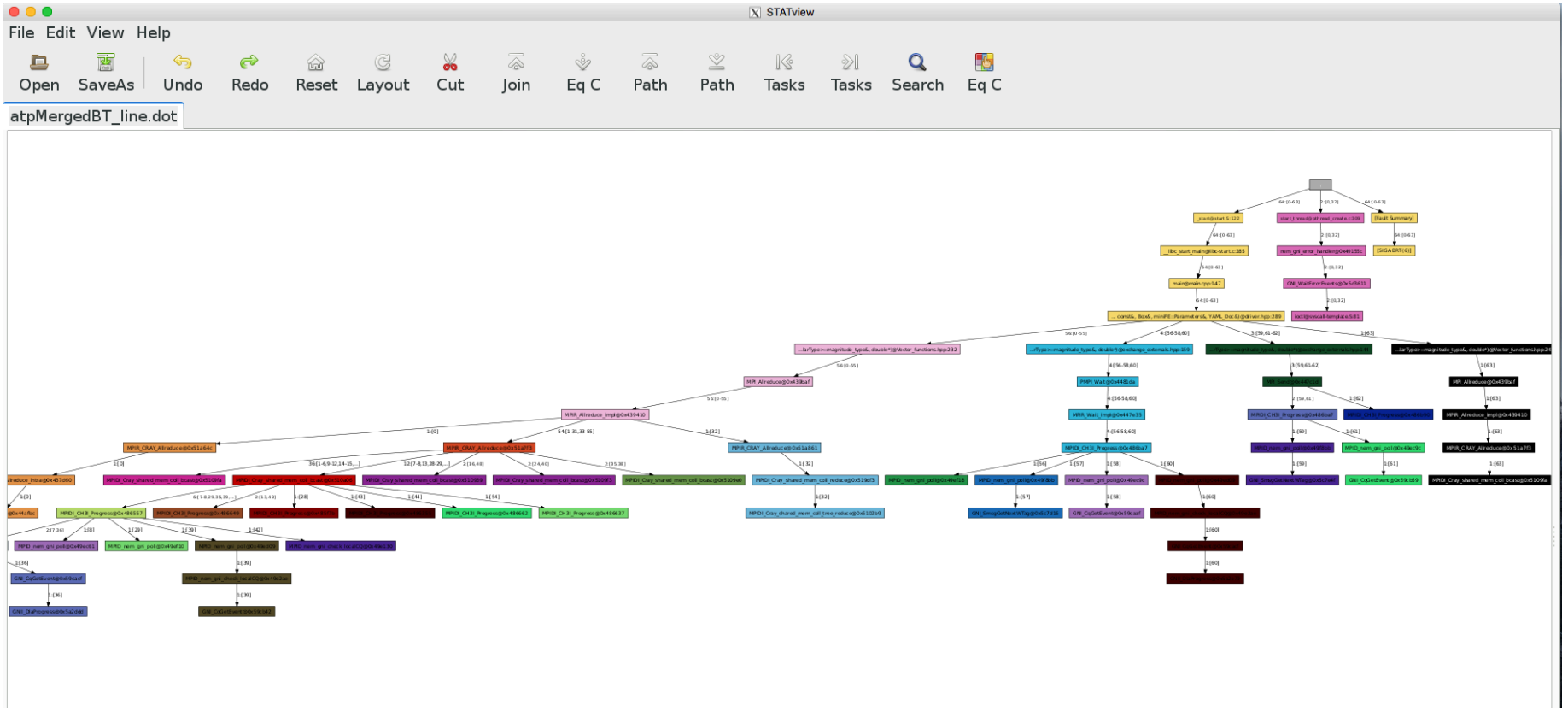
- **Requires libraries linked in to application**
  - Cray compiler wrappers do this if 'atp' module is loaded
- **Requires environment variable at run-time**
  - ATP_ENABLED=1
- **For hung processes:**
  - On Cori, "scancel" must be run on MOM node, not login node (run via "ssh cmom02 scancel –s ABRT <jobid>")
  - On Blue Waters, use "apstat | grep $USER" to find apid (first column), then "apkill <apid>"

Hands-on exercise

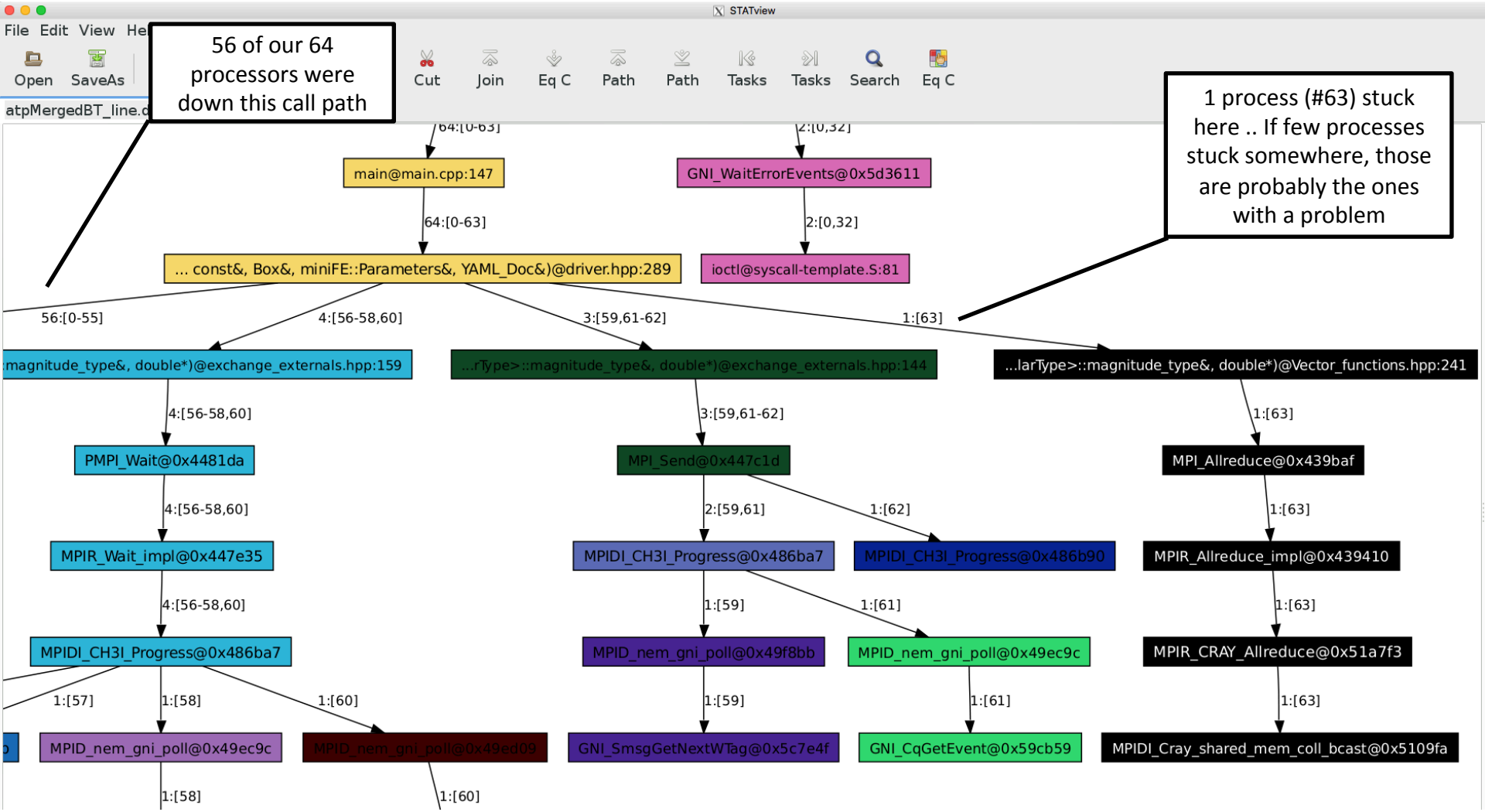
- **See ex6-debugging/README.rst**
- **This exercise requires a Cray system**
  - Cori or Blue Waters
- **We'll build a version of miniFE that has some bugs "enabled" (via preprocessor), then use ATP and STAT to try to identify the location of the bugs**
  - Application might crash, or might hang
  - For hung jobs, you might need to be quick (before walltime limit hits)
- **5-10 minutes, then we'll look at some results**

Hands-on exercise 6

- **If you caught a hung job, stat-view probably looked something like:**



Hands-on exercise 6



Key ideas

- Debugging at scale is hard .. But there are tools that can help
- STAT/ATP: lightweight capture of many stack traces
- DDT or Totalview (if available) for parallel-capable version of traditional debugger

Q & A

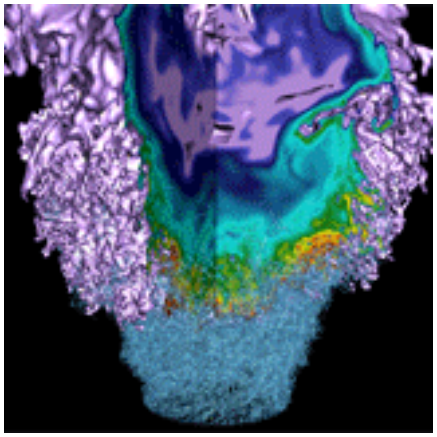
Agenda

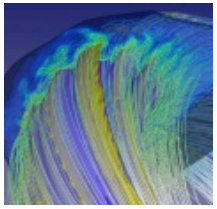
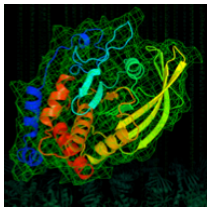
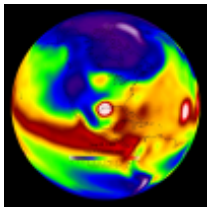
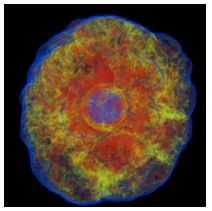
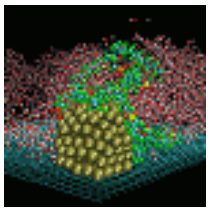
- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- OpenMP scaling

---- 1.45pm PDT: 15 minute break ----

- Weak scaling
- Debugging at scale
- **Finding the sweet spot**
- Summary and Conclusions

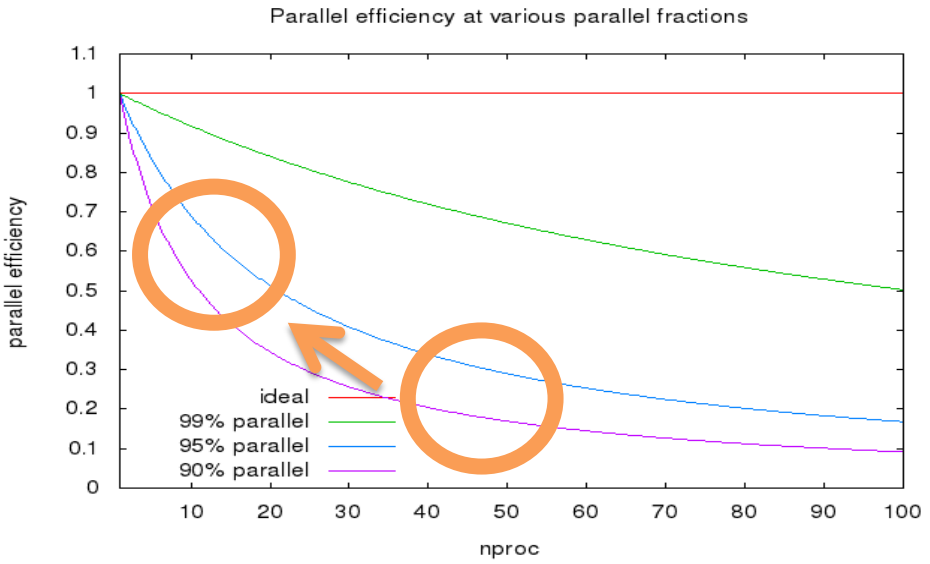
Finding the sweet spot
Hybrid parallelism



Hybrid MPI/OpenMP

- **MPI scaling limitations – communication at scale**
- **OpenMP scaling limitations – resource sharing**
  - Both together: benefits, costs of both
  - A well-optimized application will benefit from higher OpenMP thread count and reduced MPI process count .. up to a point
  - Move to left of efficiency curve!

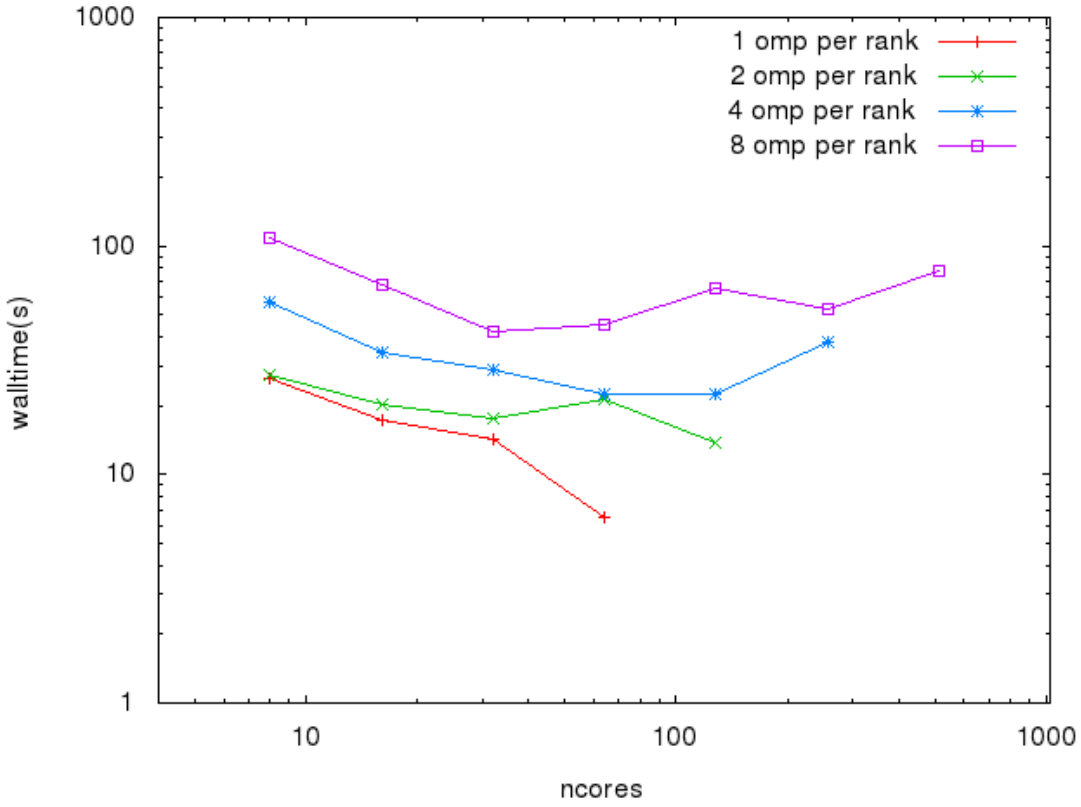


Hands-on exercise

- **See ex7-hybrid/README.rst**
- **We'll build miniFE for MPI+OpenMP, then run at a few combinations of OpenMP-threads-per-MPI-process, plot performance**
- **Jobs will take a little longer to run so we'll start them, then return to slides for some more discussion**
  - Can revisit in Open Lab session
- **5 minutes, then we'll look at some results**

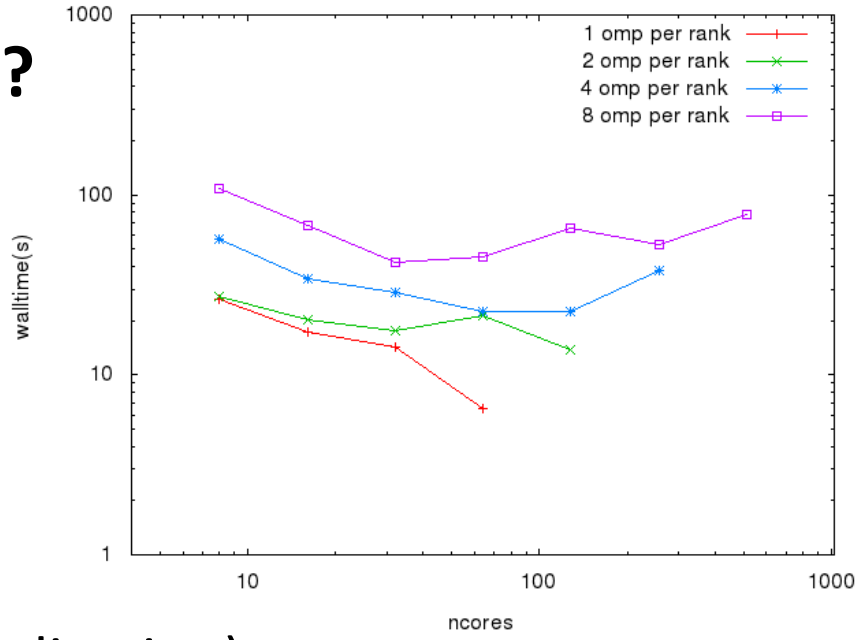
Hands-on exercise

- **Once you have plot of timings, you will probably see something like this:**



Hands-on exercise

- **What do we read from this?**
  - In this instance, at all scales, fewer OpenMP tasks (or MPI only) is fastest
    - miniFE OpenMP scaling is limited, as we saw in exercise 4
    - (not the case for every application)
  - Notice 2 vs 4 OMP per rank, at 64 cores
    - Higher thread count *almost* outperforms lower



Key ideas

- **Multi-level parallelism as a means of moving left along the parallel-efficiency curve**
- **Efficiency trade-offs at each level – experiment to find optimal runtime parameters**

Q & A

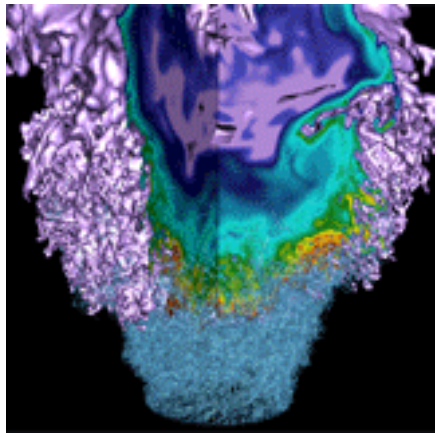
Agenda

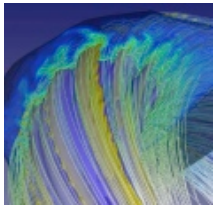
- Amdahl's law
- Where is the bottleneck?
- Why is *this* a bottleneck?
- OpenMP scaling

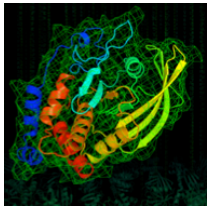
---- 1.45pm PDT: 15 minute break ----

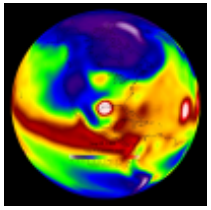
- Weak scaling
- Debugging at scale
- Finding the sweet spot
- **Summary and Conclusions**

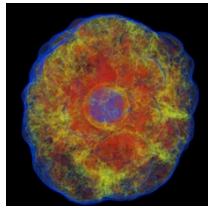
Summary and Conclusions

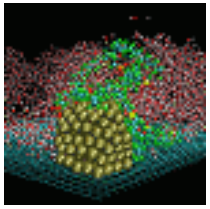












Summary, Conclusions

- **Amdahl's Law: serial fraction drastically limits scalability**
- **Gustafson's Law: the problems that need extreme scalability usually have higher parallel fraction**
- **Tools / Techniques for identifying / understanding scaling limiters in an application**
  - Profile, compare profiles
  - Trace for fine-grained look – be prepared for overhead!
- **Hybrid parallelism – experiment to find optimal balance for your application**

National Energy Research Scientific Computing Center