High-Performance Computing

Fix Your OpenMP Problem by Using (the right) Tools

Bernd Dammann

Assoc. Professor Scientific Computing,
DTU Compute
&
Scientific Lead HPC,
DTU Computing Center (DCC)

Intro – the tool used here: gprofng

gprofng – short overview

What is gprofng?

- a next generation (‘ng’) GNU profiling tool
- part of GNU binutils since 2022
- Two step process:
  1. Collect application level performance data
     - gprofng collect app ./your_app your_args
  2. Display the recorded profiling data
     - gprofng display [text | html | gui] recordedprofile.er

gprofng – short overview

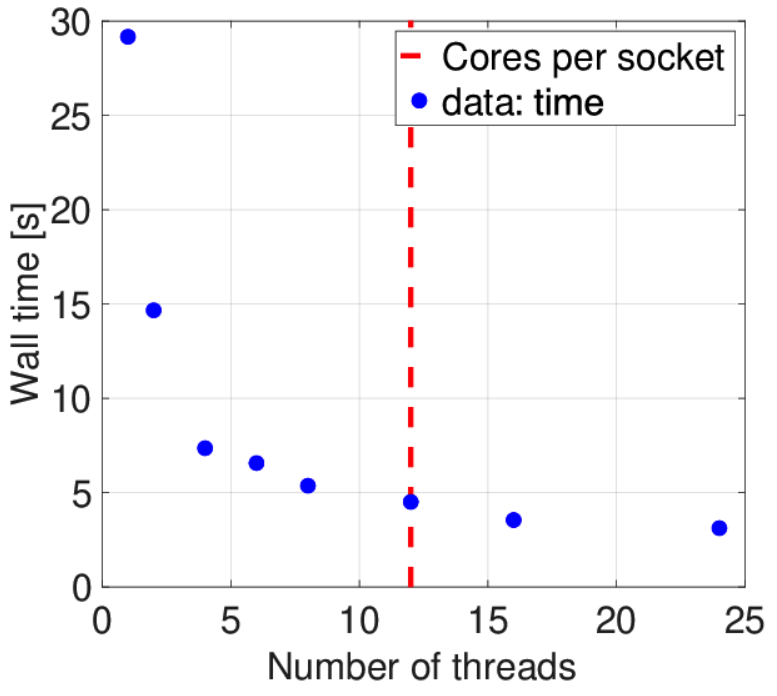
- Languages supported: C/C++, Fortran, Java, ...
- Full support for gcc compilers
- Various processors from Intel, AMD, and Arm
- No need to recompile the code
  - Works with production binaries
- Supports Multithreading
  - Posix Threads, OpenMP, and Java Threads
- GUI with a timeline (recommended add-on)
  - http://savannah.gnu.org/projects/gprofng-gui/

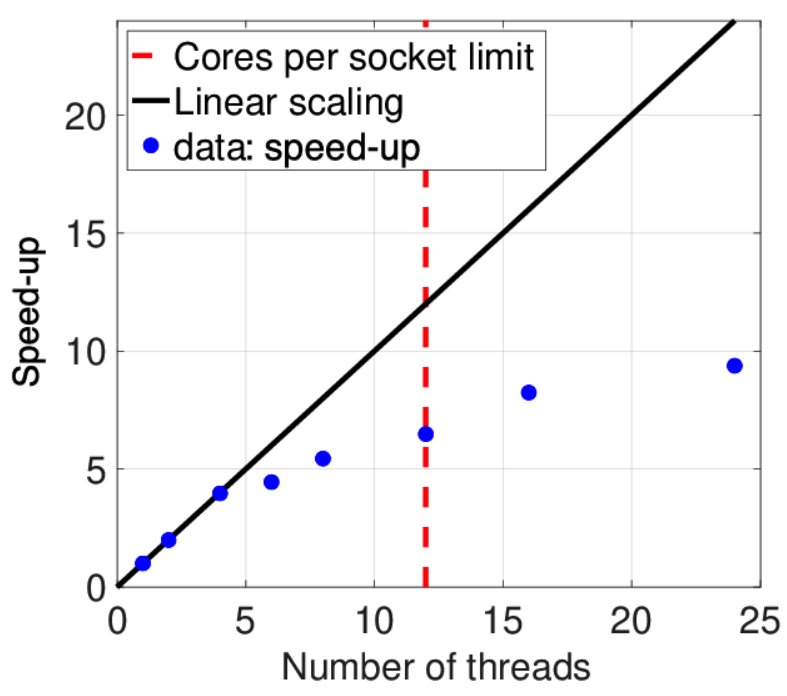
Episode I:

The problem and the single letter performance fix

Episode I: the problem

- a Monte Carlo simulation, OpenMP version
- embarrassingly parallel, but ...

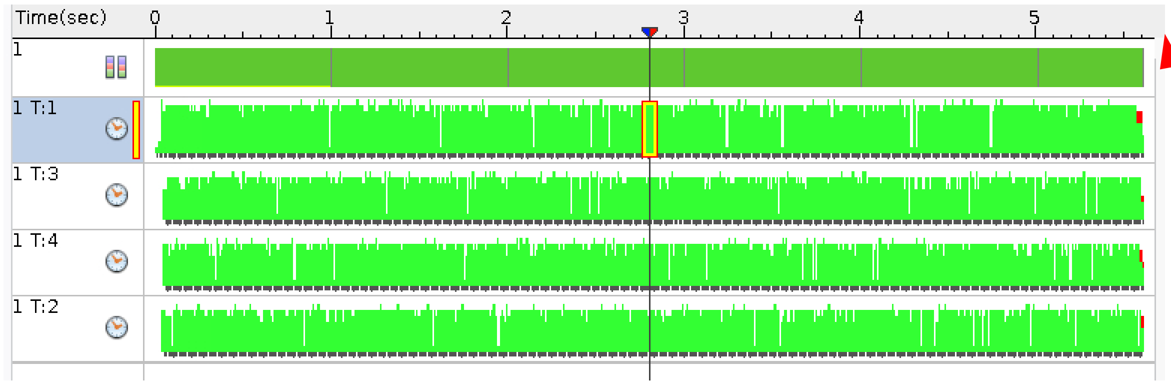




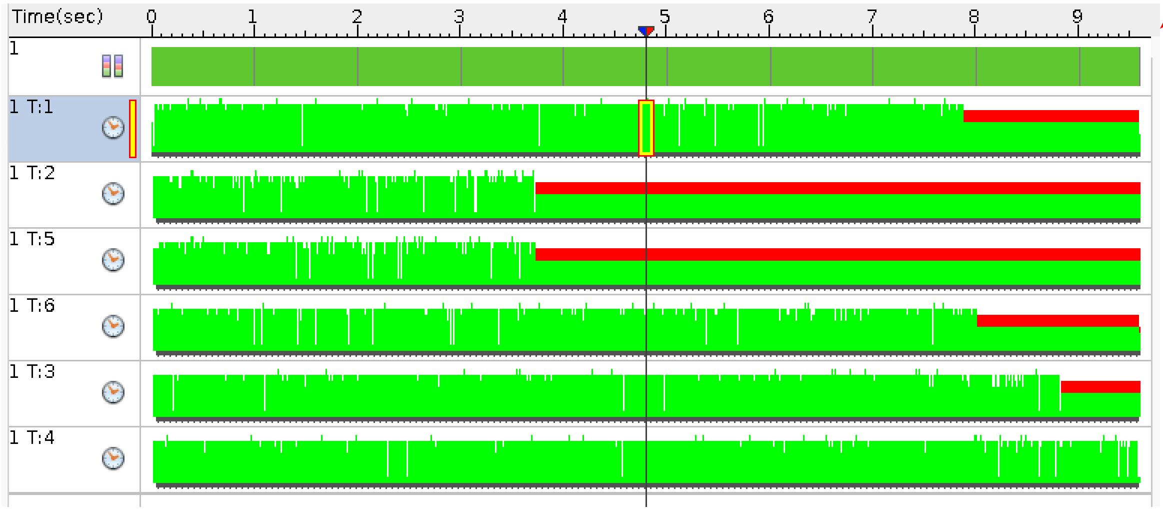
nice scaling up to 4 threads,
but not for 6 or more threads

Episode I: the analysis

- a first analysis, using the gprofng timeline
  - color coding: green is good, red is bad (barrier)



total run time increases, too!



severe load imbalance, when using 6 threads!

Episode I: the analysis

- the load imbalance displayed in the profile suggested to use a ‘dynamic’ loop schedule
  - yes – better balancing (the red parts in the timeline get reduced)
  - ... but the overall runtime, as well as the scaling, did not improve!
  - ... and why is there no imbalance for 4 threads and less?

Episode I: the analysis

Looking at the function profiles (top 5):

4 threads

```
Excl.       Incl.       Name
Total       Total
CPU sec.    CPU sec.
20.815      20.815      <Total>
 5.244       5.244      fedbatchDrift
 4.783       5.064      matrix_vector_mul
 2.202      20.294      sdeClosedLoop
 2.011       2.011      vector_vector_add
```

6 threads

```
Excl.       Incl.       Name
Total       Total
CPU sec.    CPU sec.
51.006      51.006      <Total>
19.874      20.504      matrix_vector_mul
13.940      13.940      gomp_team_barrier_wait_end
 5.954       5.954      fedbatchDrift
 2.752       2.752      vector_vector_add
```

Episode I: the analysis

Looking at the function profiles (top 5):

- a 4.1x increase in CPU time for the matrix vector product, when going from 4 to 6 threads – for the same amount of floating point operations!
- single-threaded function, called in parallel
- time seems to increase when accessing the matrix and vector elements (from looking at the source level profile)

==> resource congestion – but why, and where?

Episode I: the analysis

Making use of hardware performance counters

- gprofng supports hardware counter profiling
- we investigated the “usual suspects”, like access to the different cache levels, etc
- one can also measure ‘stalls’ on resources
  - if supported as a counter on the HW
  - the counter of interest used here is ‘resource_stalls.any’

Episode I: the analysis

Looking at the function profiles again:

4 threads

```
Total       resource_stalls.any    Name
CPU sec.    Events sec.
22.196      4.530                  <Total>
 5.674      1.950                  fedbatchDrift
 2.071      0.974                  fedbatchDiffusion
 5.764      0.303                  matrix_vector_mul
 2.162      0.171                  vector_vector_add
 1.051      0.063                  sdeClosedLoop
 0.981      0.003                  vector_scalar_mul
=====================================================
57.240      20.015                 <Total>
20.895      12.463                 matrix_vector_mul
 6.525       2.480                 fedbatchDrift
 4.533       1.355                 vector_vector_add
 2.252       1.192                 fedbatchDiffusion
 1.611       0.417                 vector_scalar_mul
 1.041       0.320                 sdeClosedLoop
```

6 threads

42x increase!

Episode I: the analysis

Having a closer look:

- a 42x increase in waiting time, when accessing matrix and vector elements
- all data is local to the threads: dynamically allocated in the threads, before calling matrix_vector_mul()
- the amount of data here is really small: 96 bytes per thread!

==> could this be ‘false sharing’?

Intermezzo: What is False Sharing?

False sharing checklist:

- Be alert when all of these 3 conditions are fulfilled:
  - Shared data is modified by multiple cores
  - Multiple cores operate on the same cache line
  - This update occurs very frequently
- Use local data where possible
- Shared read-only data does **not** lead to false sharing!

Episode I: the analysis

Having a closer look:

- a 42x increase in waiting time, when accessing matrix and vector elements
- all data is local to the threads: dynamically allocated in the threads, before calling matrix_vector_mul()
- the amount of data here is really small: 96 bytes per thread!

==> could this be 'false sharing'?

===> not according to the checklist!

Episode I: the analysis

The smoking gun?

- are the 96 bytes workspaces the problem here?
- cache line size of the system: 64 bytes
- 96 bytes are 1.5 cache lines
- there is a theoretical chance for something that looks like ‘false sharing’
- ... but the allocation happens in different threads!?!?!
- is our malloc() broken?

Episode I: the ‘one letter fix’

- how to move the allocations further apart?
- there is valloc(): “*The allocated memory is aligned on a page boundary.*” (from the man page)
- typical page size: 4 kBytes (64x the cache line size)
- let’s replace the malloc() call for the workspace data by valloc(), and ...

Episode I: the ‘one letter fix’

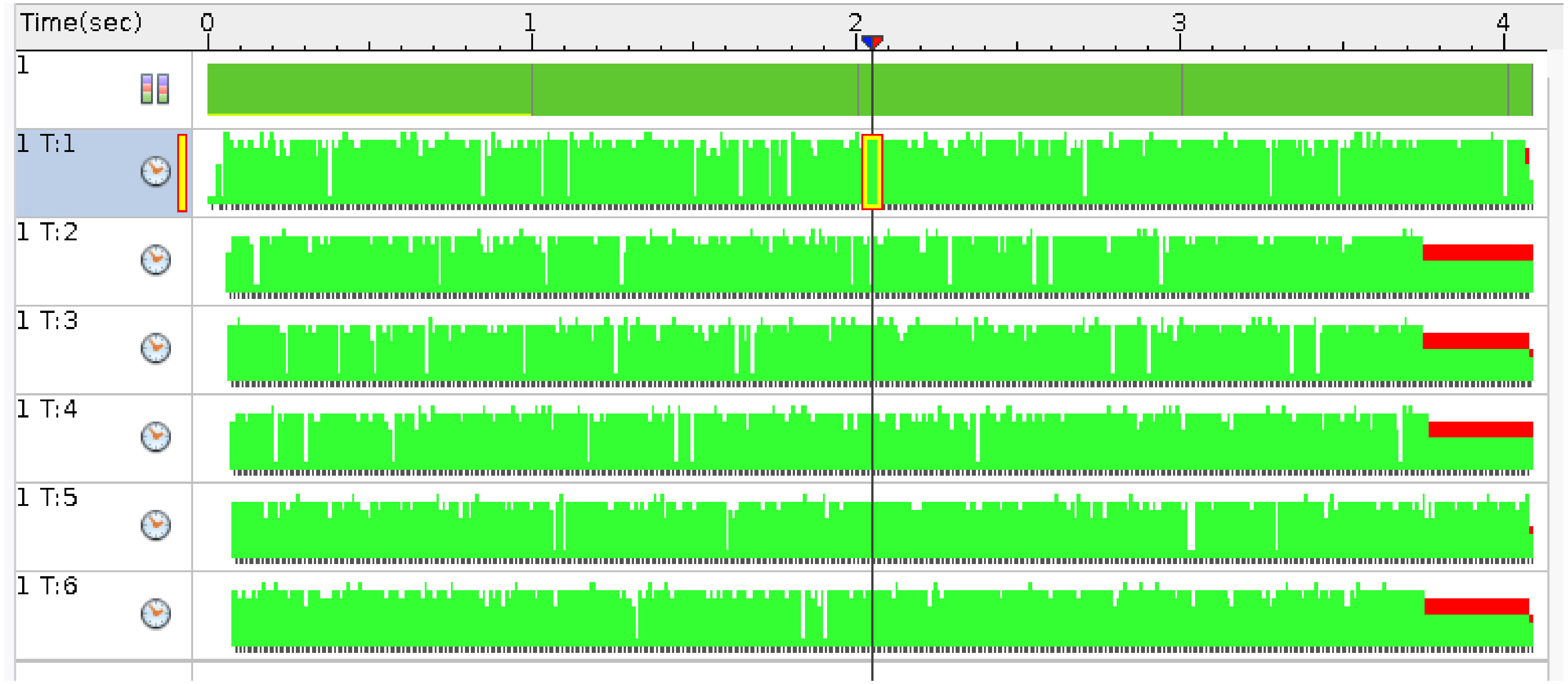
```
Excl.      Excl.                        Name
Total      resource_stalls.any
CPU sec.   Events sec.
57.240     20.015                       <Total>
20.895     12.463                       matrix_vector_mul
 6.525      2.480                       fedbatchDrift
 4.533      1.355                       vector_vector_add
 2.252      1.192                       fedbatchDiffusion
 1.611      0.417                       vector_scalar_mul
 1.041      0.320                       sdeClosedLoop
=====================================================
24.027      6.162                       <Total>
 6.124      2.380                       fedbatchDrift
 2.141      1.096                       fedbatchDiffusion
 5.344      0.981                       matrix_vector_mul
 0.911      0.104                       sdeClosedLoop
 2.302      0.080                       vector_vector_add
 0.871      0.007                       vector_scalar_mul
```

malloc()
6 threads

valloc()
6 threads

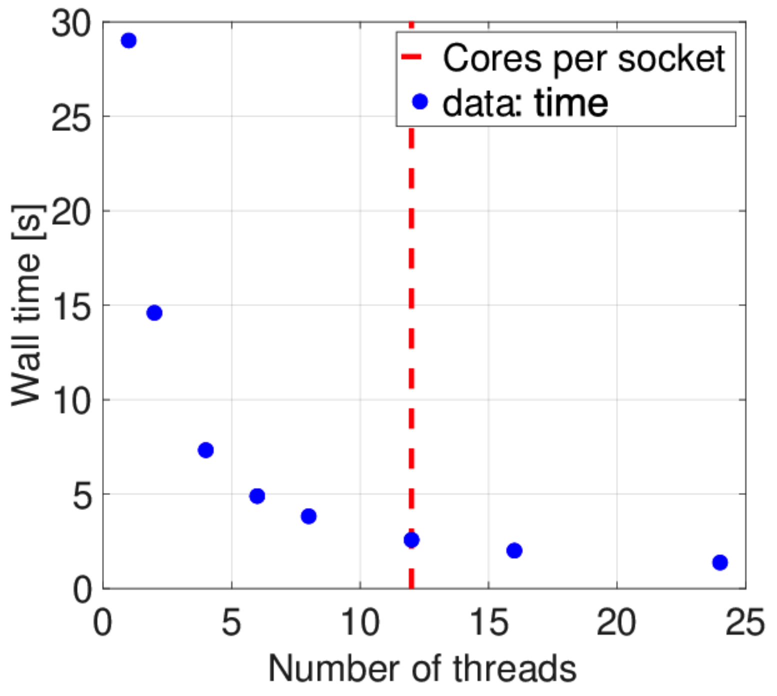
Episode I: the ‘one letter fix’

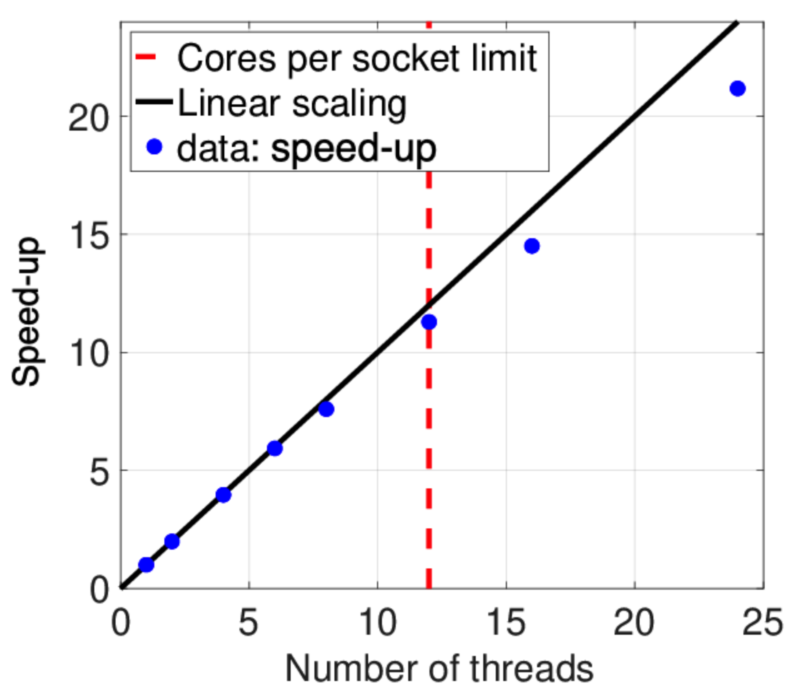
Timeline after the fix:



Episode I: the ‘one letter fix’

Runtime and scaling:





Episode II:

The mysterious five

Episode II: the story continues

Problem solved – so what? Lean back ...

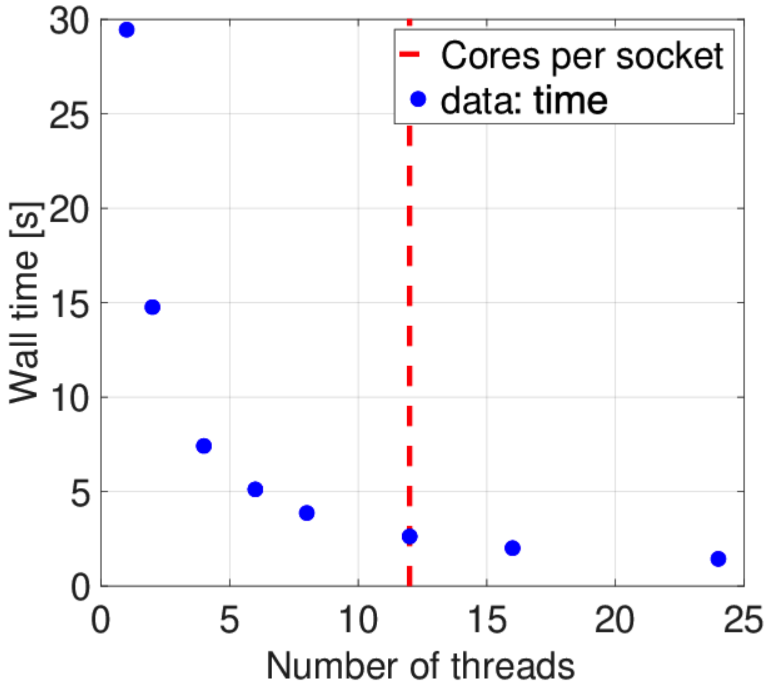
- Not really! Open questions/issues:
  - if this is (a kind of) false sharing, ...
    - ... can we make it go away, allocating a larger workspace? (cross-check/confirmation)
    - ... why do we first see it for 5 and more threads?
  - valloc() is obsolete (according to the man pages)
    - it solved our problem here, but we should not rely on it
    - what about indirect usages of malloc(), e.g. new[ ] in C++? There will be no simple fix!

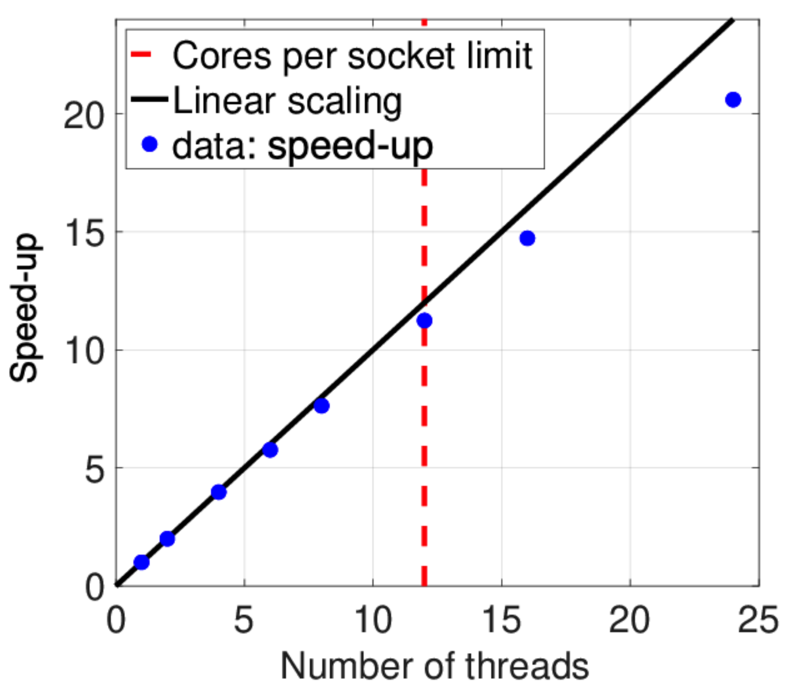
==> we need to dig deeper into this!

Episode II: false sharing cross check

Does it help to make the workspace larger, e.g. 1kB instead of 96 bytes?

- Yes ... it helps – which indicates false sharing





Episode II: malloc internals

GLIBC’s malloc behind the scenes

- *In order to efficiently handle multi-threaded applications, glibc's malloc allows for more than one region of memory to be active at a time. Thus, different threads can access different regions of memory without interfering with each other. These regions of memory are collectively called "arenas".* (from: MallocInternals glibc Wiki)
- Thus, there should be no overlap, as there can be enough arenas (max. 8x the number of cores), and therefore no false sharing

Episode II: malloc internals

GLIBC’s malloc behind the scenes (cont’d)

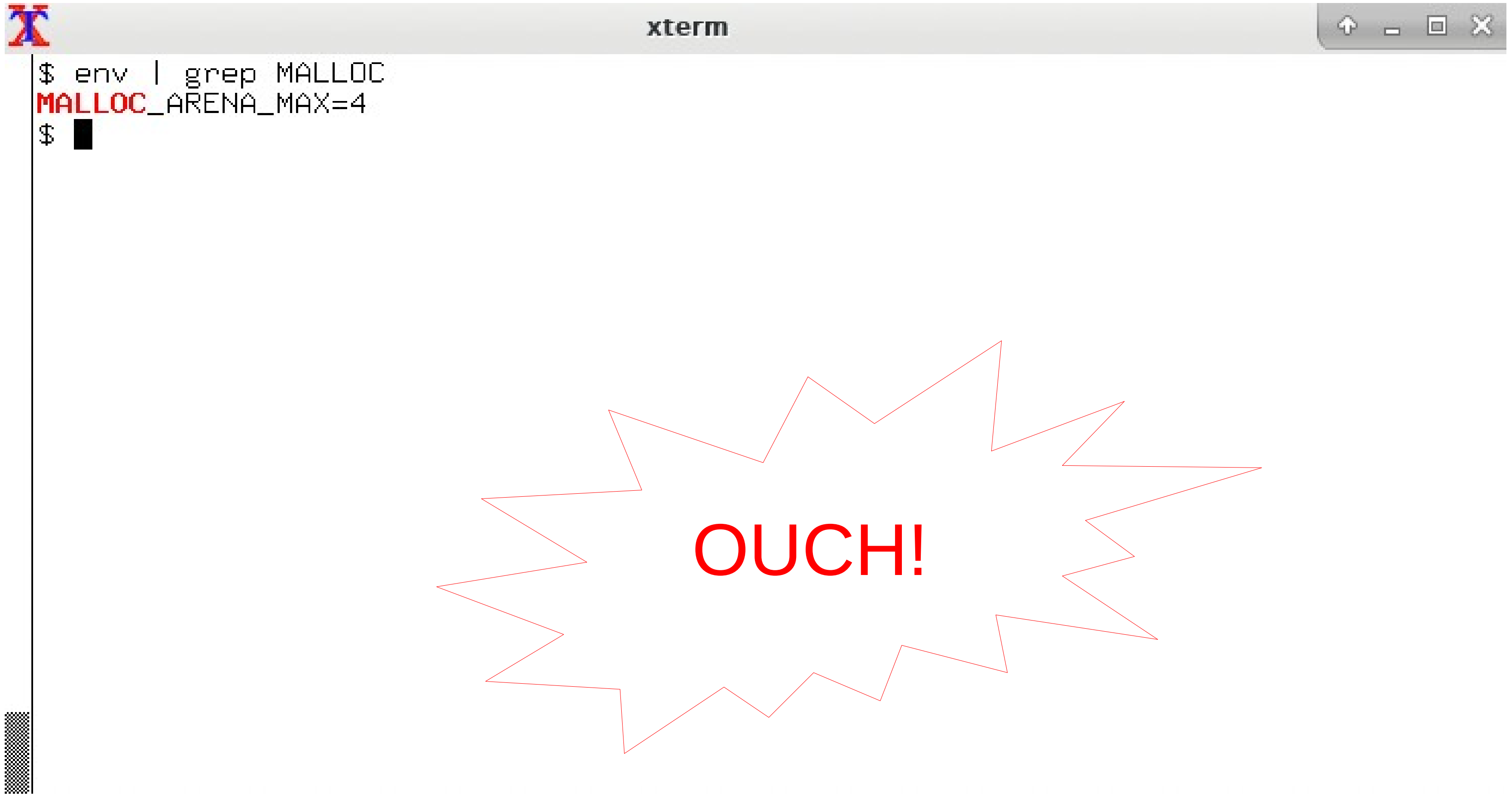
- one (nasty) side effect of the arena implementation in multi-threaded applications:
  - it creates an arena for each active thread ...
  - ... but it cannot distinguish between ‘good compute threads’ and other threads (e.g. from Java VMs)
  - this is a bomb under e.g. Java based applications, that create a lot of threads, as your virtual memory usage explodes ...
  - ... and if you operate with memory limits (e.g. on multi-user systems, or via resource managers)

Episode II: malloc internals

GLIBC’s malloc behind the scenes (cont’d)

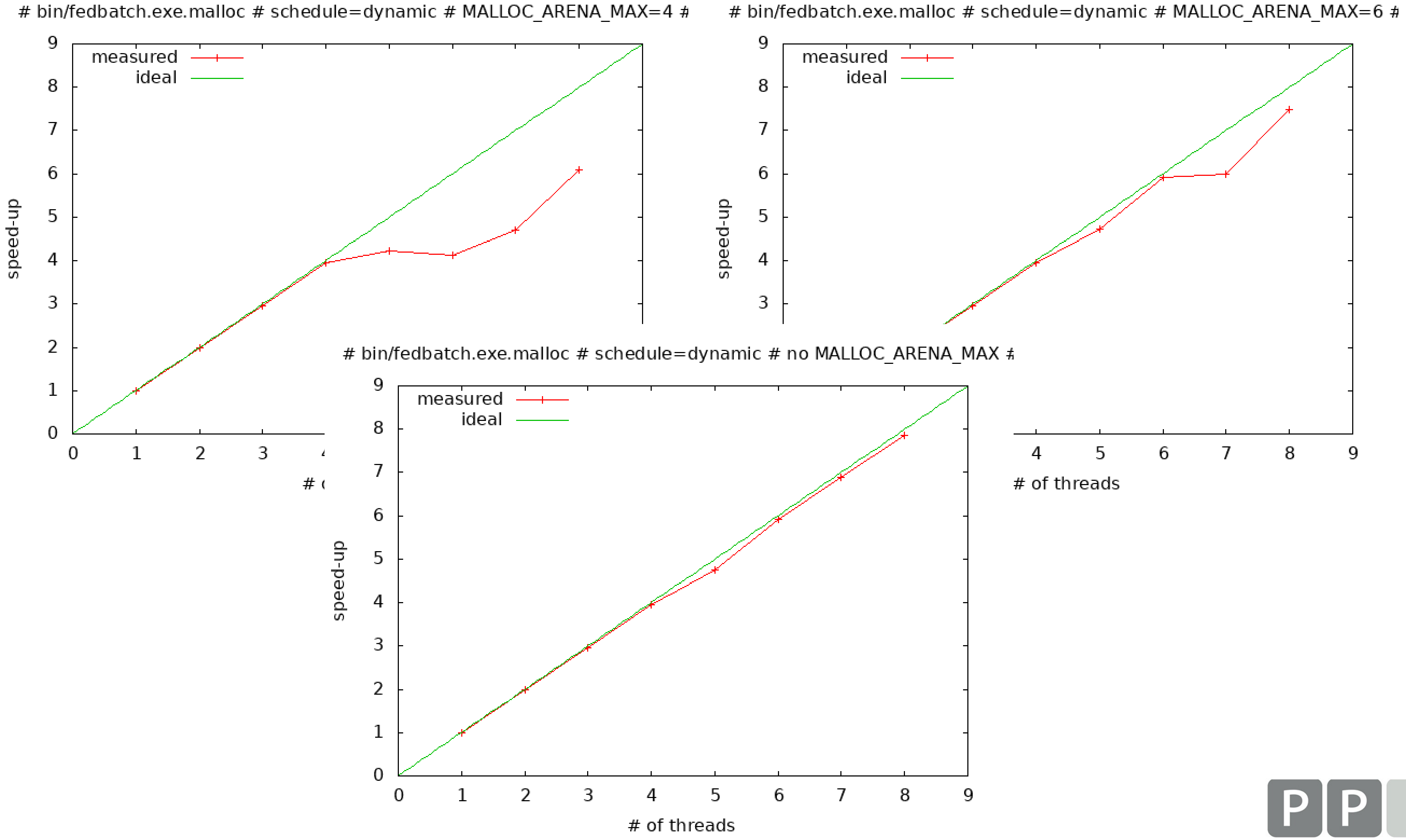
- to reduce this effect, one can limit the number of arenas via an environment variable
  - MALLOC_ARENA_MAX=n
- Hmm ....

Episode II: The mysterious five



```
$ env | grep MALLOC
MALLOC_ARENA_MAX=4
$ 
```

Episode II: The mysterious five



Episode II: The mysterious five

Conclusions:

- the reason behind all the problems was the ‘legacy’ setting of an environment variable
  - MALLOC_ARENA_MAX=4
  - needed to solve memory limit issues with MATLAB, when our resource manager didn’t support cgroups, yet!
- removing the variable made the application behave as it should – linear speed-up, with malloc(), beyond five threads!

Episode II: The mysterious five

Lessons learned – take home messages:

- using a tool (gprofng) has enabled us to pinpoint the problem ...
- ... which “should not exist”
- don’t accept the simple fix, if you still are in doubt
  - BTW: a few weeks after, we saw the same issue with a C++ code and new[ ]
- look up implementation details
- check your runtime environment
- ... and use your experience!

THE END