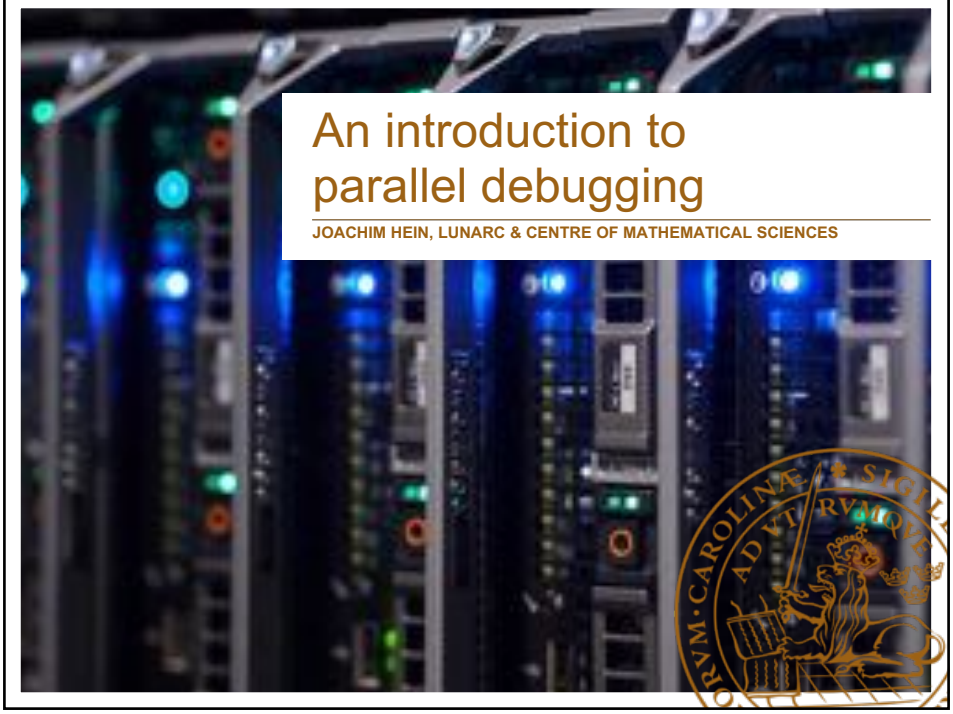


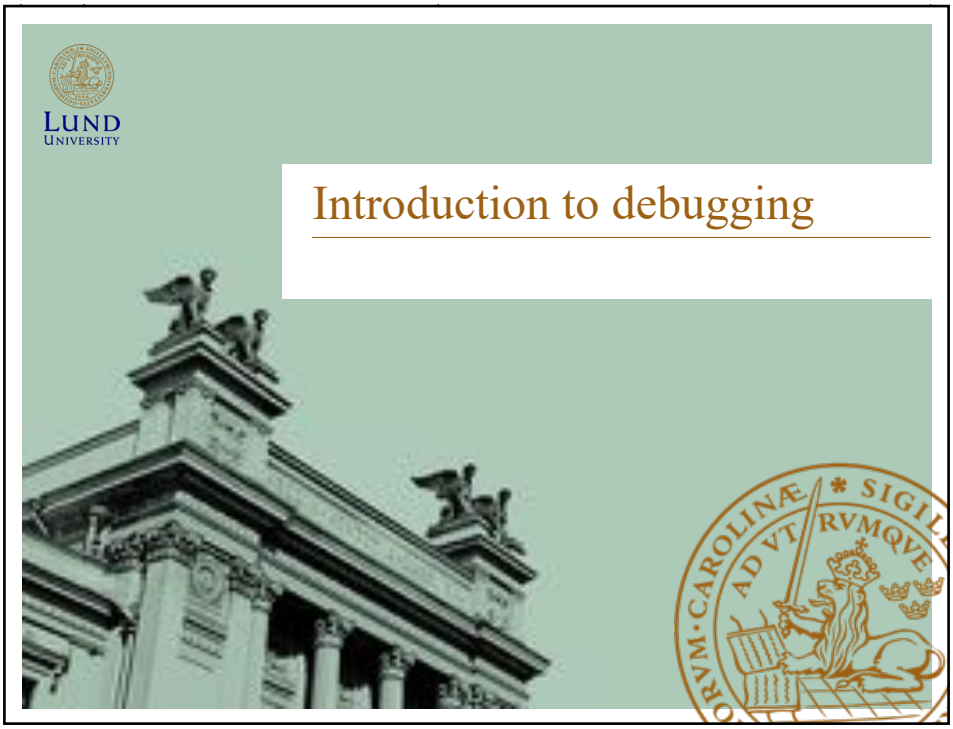
An introduction to parallel debugging

JOACHIM HEIN, LUNARC & CENTRE OF MATHEMATICAL SCIENCES

Overview

- Introduction to debugging and parallel debugging
- Running the ARM DDT parallel debugger

LUND UNIVERSITY



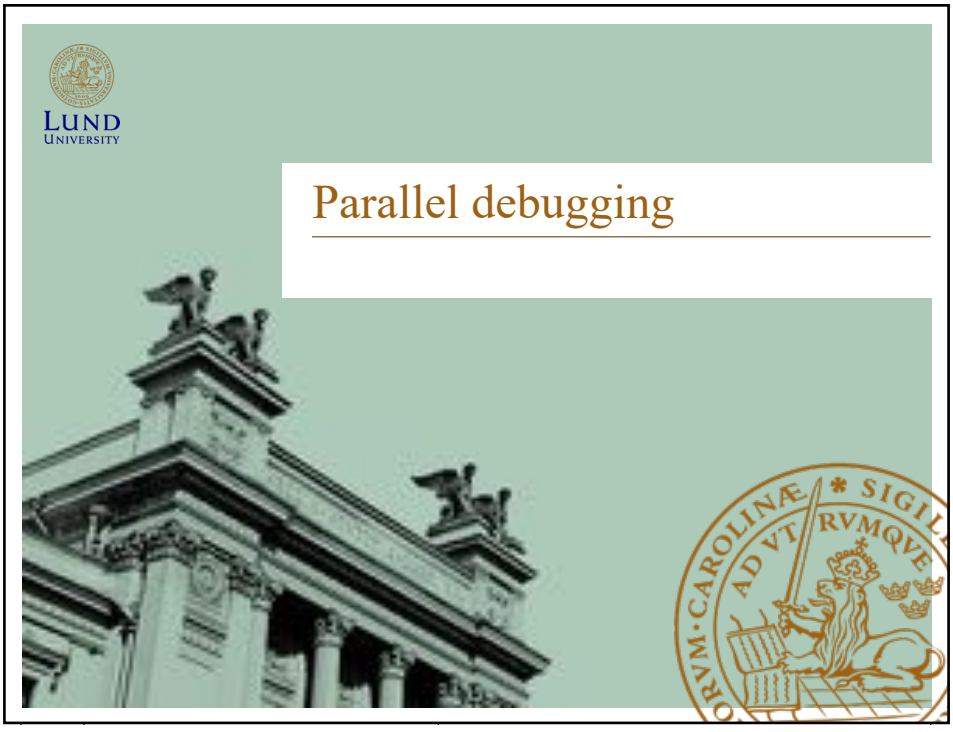
Traditional standard way to debug: "`printf` debugging"

- Add extra print statements to the code
  - Indicate whether the code reaches a certain stage
  - Print the values of key variable
- Issues with this approach
  - Need to modify the source code, recompile
  - Iterative approach, frequent recompiles
- Debuggers are more convenient
  - Allows working with unmodified source
  - Allows line by line execution

LUND
UNIVERSITY

Debuggers

- Linux system come with **gdb** as a debugger
  - Command line execution
  - GUIs exist
  - Often integrated into development platforms

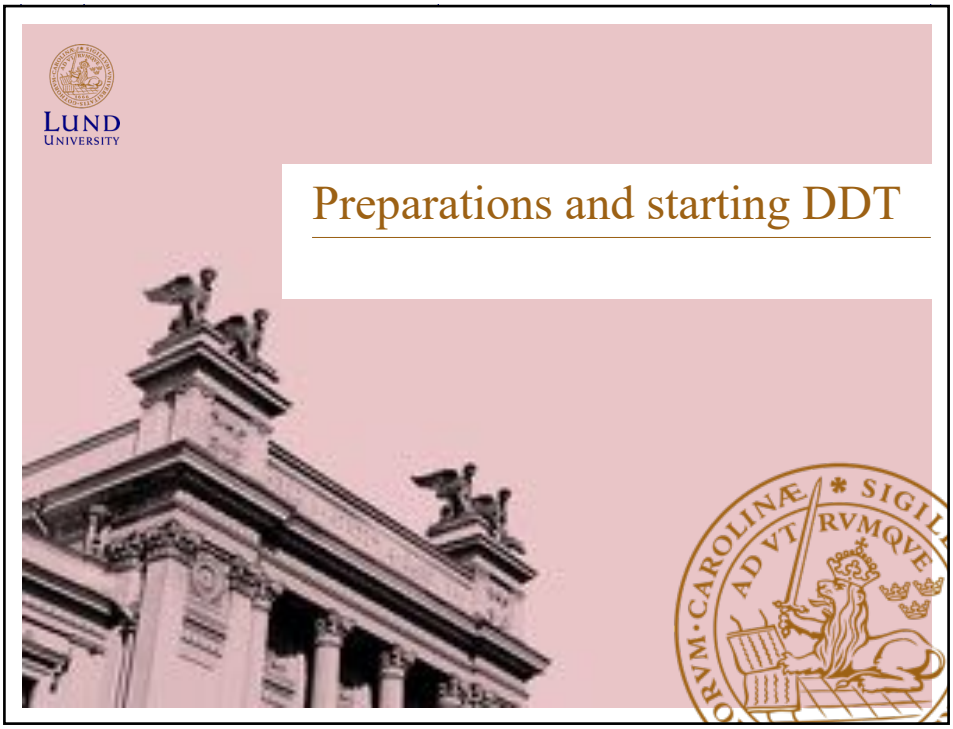


Parallel debugging

- Parallel applications offer new levels of complexity
- Before starting, try to simplify the task
  - Problem still there if you **reduce the problem size**?
  - Problem still there if you **reduce the task/thread count**?
- "printf debugging" even more problematic than in serial
  - More output (different tasks/threads printing)
  - Identification of task/thread printing required
  - UNIX `grep` helpful to filter output

Parallel debuggers

- Licenses are expensive
  - Being able to do "`printf`" is an essential skill
- Parallel debuggers more convenient to use over the years
- I am aware of two products
  - Totalview for HPC (https://www.roguewave.com/)
  - ARM DDT - part of ARM FORGE
    - » Formerly known as ALLINEA DDT/FORGE
    - » There is a SNIC wide license



Preparations

- HPC system needs to display the gui on your monitor
  - VNC solution (e.g. ThinLinc based)
  - Connect with X-forwarding (`ssh -X ...`)
- Recompile your application with the flags: `-g -O0`

```
mpif90 -g -O0 -o hello_mpi hello_mpi.f90
```

- Comments:
  - Code might run very slow (in particular C++)
  - Problem might disappear – hint for overrun array
  - You can use optimisation
    - » Though match code line to instruction might not work

LUND UNIVERSITY

Start the gui

- Best to start the gui on the login node and keep it running

```
module load allinea-forge
ddt &
```

ARM FORGE gui

Starting code on the compute nodes

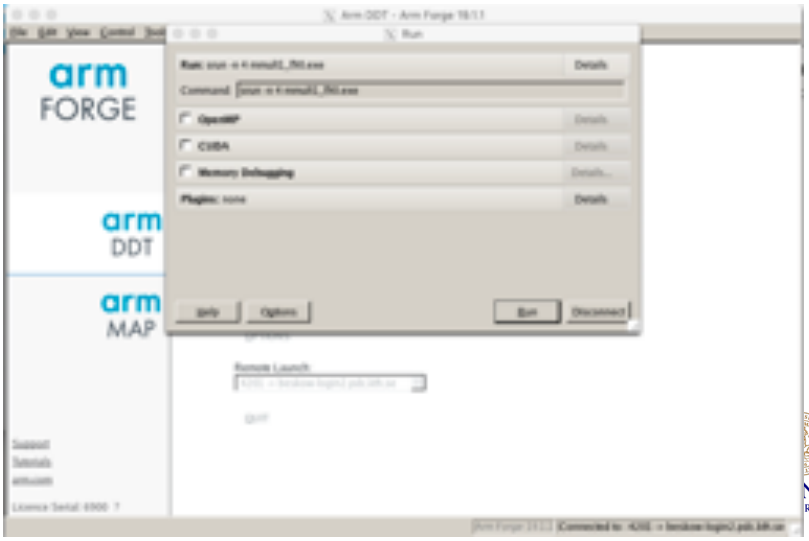
- Transfer to the backend node
  - Jobscript (currently not working on Beskow)
  - Interactive allocation (`salloc … `)
- Make sure relevant modules are loaded
  - compiler, MPI lib, other libs, ARM DDT/FORGE

- Prefix job launcher with: `ddt --connect`

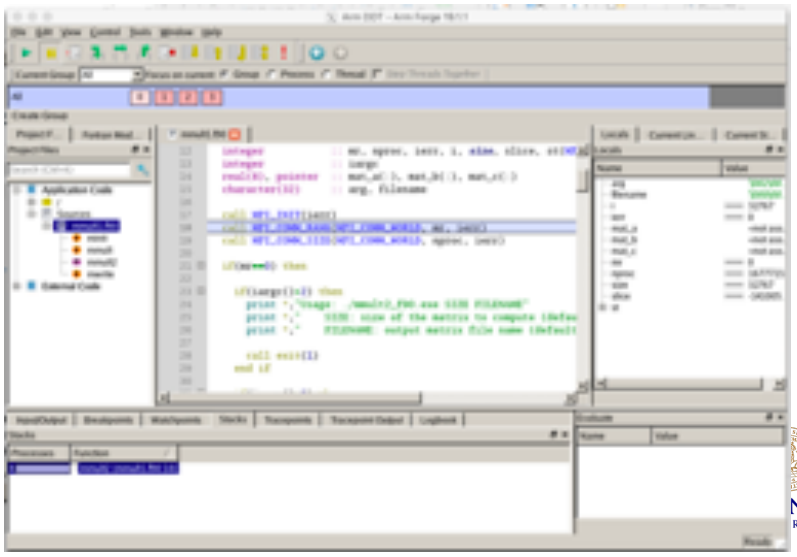
```
ddt --connect srun –n 4 hello_mpi
```

Accept the “Reverse Connect request”

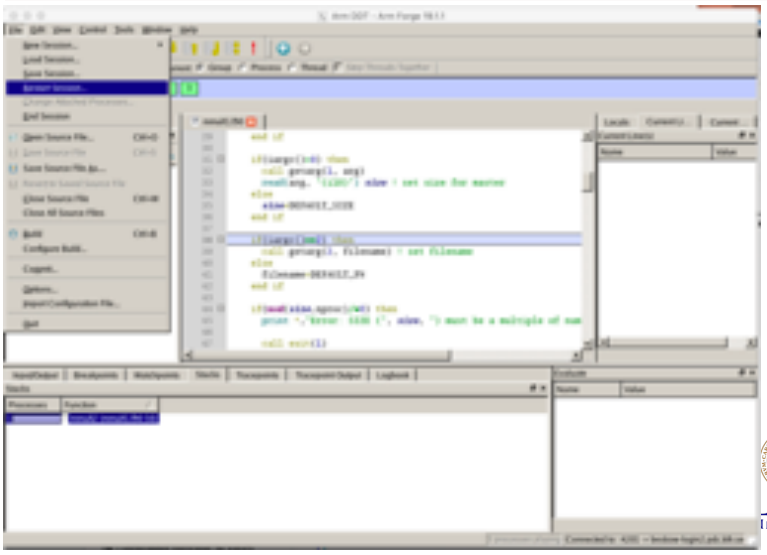
Start running your program



Start debugging in the gui



Starting over – frequently required
Use “Restart session option”



Demo

- hello world
- matrix matrix multiply
- OpenMP code

Memory debugging

A KILLER FEATURE OF DDT

Problematic memory access

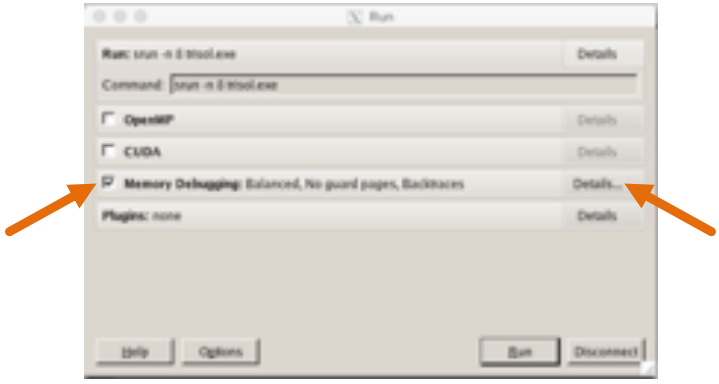
- Codes often suffer from memory problems
  - Writing in memory locations they shouldn't
  - Illegal deallocation (double, bad pointer position, ...)
  - Memory leaks
- Typical signatures of memory problems
  - Seg-faults
  - Code behaviour changes when:
    - » Editing (e.g. printf debugging)
    - » Changing compilers or optimisation flags

LUND UNIVERSITY

Activating memory debugging in DDT

- Replace the malloc library with ARM's dmalloc
- Comes in 4 versions:
  - C/Fortran no threads
  - C/Fortran threads
  - C++ no threads
  - C++ threads

Select memory debugging



Selecting Memory Debugging Option

LUND UNIVERSITY

Dynamic linking

LUND UNIVERSITY

Static linking

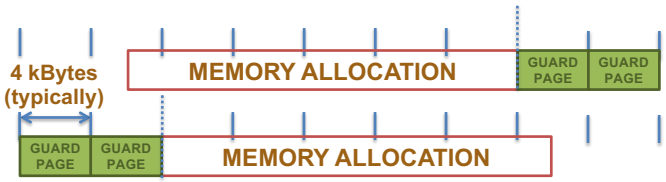
- If you link statically or if dynamic linking fails
- Add a line like (check user guide)

```
-Wl,--allow-multiple-definition,--undefined=malloc /path/lib/64/libdmalloc.a
```

to the link line **before** anything else

– Often required on CRAYs

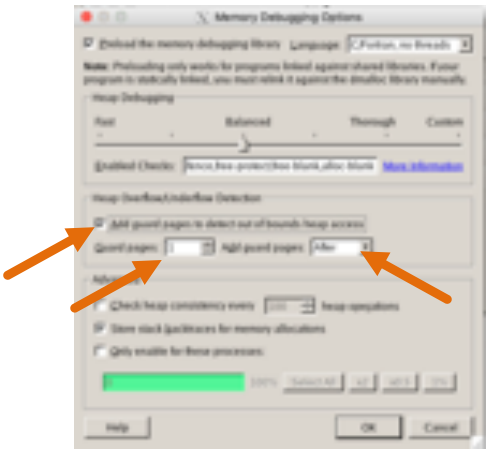
Guard pages (aka “electric fences”)



- **A powerful feature…:**
  - **Forbids read/write on guard pages throughout the whole execution**

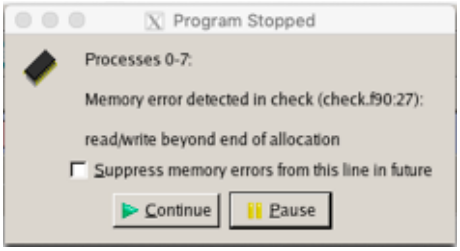
    (because it overrides C Standard Memory Management library)
- **… to be used carefully:**
  - **Kernel limitation: up to 32k guard pages max**
  - **Beware the additional memory usage cost**
- **Choice of before/after**

Activate guard pages



LUND UNIVERSITY

When it finds something you get:



LUND UNIVERSITY

Demo

- Locating memory issue

Recap/Summary

- Starting the gui
- Demonstrating how to run it
- Memory debugging feature
  - This saved me so much time in the years

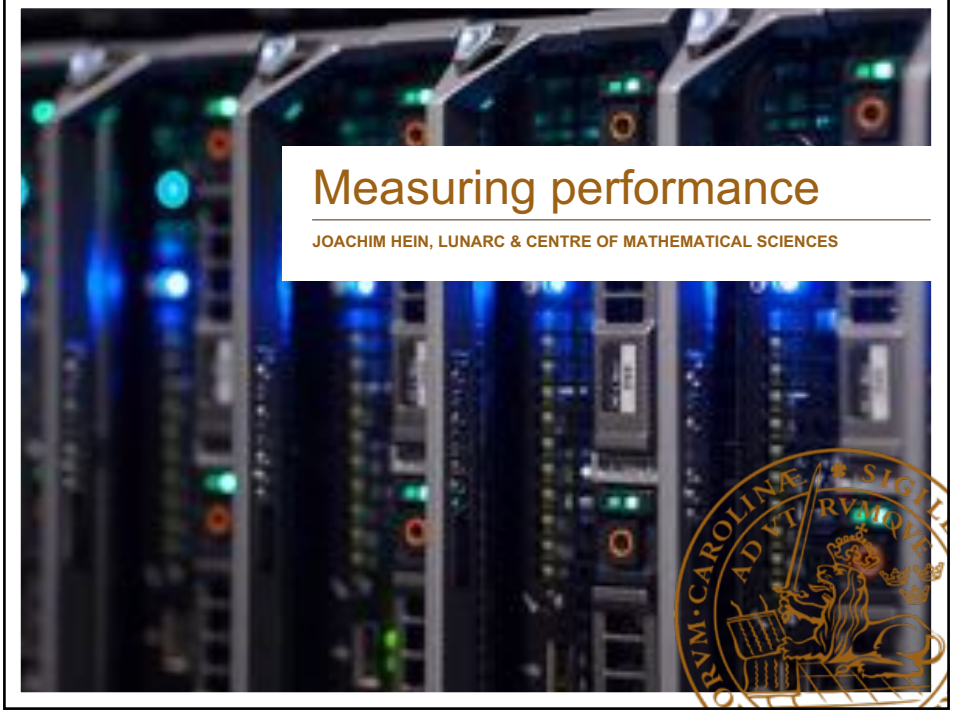
Acknowledgements

- Juan Gao (ARM)
- Patrick Wohlschlegel (ARM)

- Thor Wikfeldt (KTH/PDC)

LUND UNIVERSITY

LUND UNIVERSITY



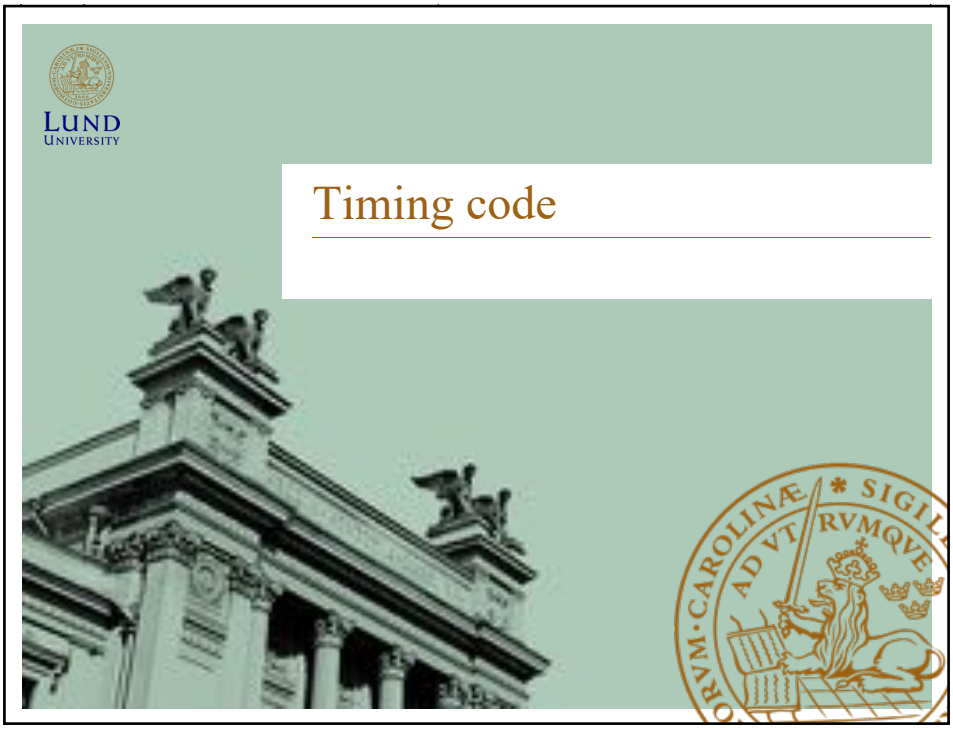
Measuring performance

JOACHIM HEIN, LUNARC & CENTRE OF MATHEMATICAL SCIENCES

Overview

- CPU and wall clock timers
- Best practices when using timers

LUND UNIVERSITY



Timers

Performance assessment requires timing

Broadly speaking: two different types of timers:

- CPU time
- Wall clock time

CPU timer

CPU time is

- Time spend by CPU on behalf of a process (or thread)
- May be useful of timing on a shared cpu core
- Often subdivided into
  - User time (time spend on user code)
  - System time (time spend in system calls)
  - Else
- Often not clear what goes to user time and to system time
- For an HPC (exclusive access) typically not required

Wall timer

- Time after a specified or unspecified point in the past
- Timing similar to a stop watch extenal to the computer
  - *”What a clock on the wall would report”*
- Useful when having exclusive access to resource
  - Typically given in an HPC scenario
- No uncertainty about time being ignored
  - E.g.: Attributed to system time

Return data from a good timer

- Return type should be larger than

  > 4 Byte

- Rationale:
  - 4 byte unsigned integer: up to 4294967295
  - Require resolution of a 1 µsec
  - Timer will turn over after about 4295 sec
    - » Just over one hour !!!
- Timer is either not precise or turns over to quickly

A few timers

Timer	Comment
Fortran `system_clock`	Wall timer F95: returns a default `integer` (4 byte on x86) Fortran 2003: longer integer types possible
UNIX `gettimeofday()`	Returns time from OS Returns seconds and µs since EPOCH (1 Jan '70) Affected by discontinuities and NTP
UNIX `clock_gettime`	Returns time from the OS Choice of timers (`CLOCK_REALTIME` required) • `CLOCK_REALTIME:` sec and ns since EPOCH, affected by discontinuities and NTP • Other optional timers: `CLOCK_MONOTONIC`, `CLOCK_MONOTONIC_RAW`, `CLOCK_PROCESS_CPUTIME_ID`
MPI `MPI_Wtime()`	Wall timer, returns double.
OpenMP `omp_get_wtime()`	Wall timer, returns double.

Wrap your timer if needed

- There might not be a universal timer
- Wraping a timer makes it easier to select another one

- Example from STREAM benchmark:

```
#include <sys/time.h>
double mysecond()
{
   struct timeval tp;
   struct timezone tzp;
   int i;
   i = gettimeofday(&tp,&tzp);
   return ((double) tp.tv_sec + (double) tp.tv_usec*1.e-6);
}
```

Using Timers in a Benchmark

HOW TO USE THEM

Limited precision

- Clocks have a tick, which sets a resolution
- They count 1, 2, 3, … ticks
- Measured period should be many ticks
  - Judgement call, what is many
- Most timers come with query functions for the tick size
  - Examples: `MPI_Wtick()`, clock_getres()
  - Return type is not a good indication of precision

Issues when benchmarking

- Computer systems are typically noisy
  - Even when having exclusive access to hardware
    - » E.g.: system daemons
  - Worse on partially shared systems (network, I/O)
- First calls are often slower
  - Initialisation
  - Cold caches
- MPI codes can have issues with task wake-up
- **Small tests and benchmarks particularly affected**

Case study: MPI Ping-Pong

```
call MPI_Barrier(paircomm, merror)
start_time = MPI_Wtime()
    do irep = 1, repetitions
       if (rankworld .eq. 0) then
          call MPI_Send(sendbuf, probsize, MPI_DOUBLE_PRECISION, &
               1, 0, paircomm, merror)
          call MPI_Recv(recvbuf, probsize, MPI_DOUBLE_PRECISION, &
               1, 0, paircomm, mstatus, merror)
          sendbuf = recvbuf+1
       elseif (rankworld .eq. 1) then
         ...
       endif
    enddo
end_time = MPI_Wtime()
av_time = (end_time - start_time) / (2.0D0*repetitions)
av_time = (end_time - start_time) / 2.0d0
```

Testing of parallel jobs

- Synchronise tasks/threads by utilising barriers
  - **BEFORE** starting the timer
- Each task/thread will have their own timing
- Use reductions (sum and/or min) to create a single result
- Looking at the spread between tasks can be useful
- Alternatively difference between average and min

- Comparing the sum of the times reported to the feedback from the UNIX time is an important sanity check

Further comments on performance testing

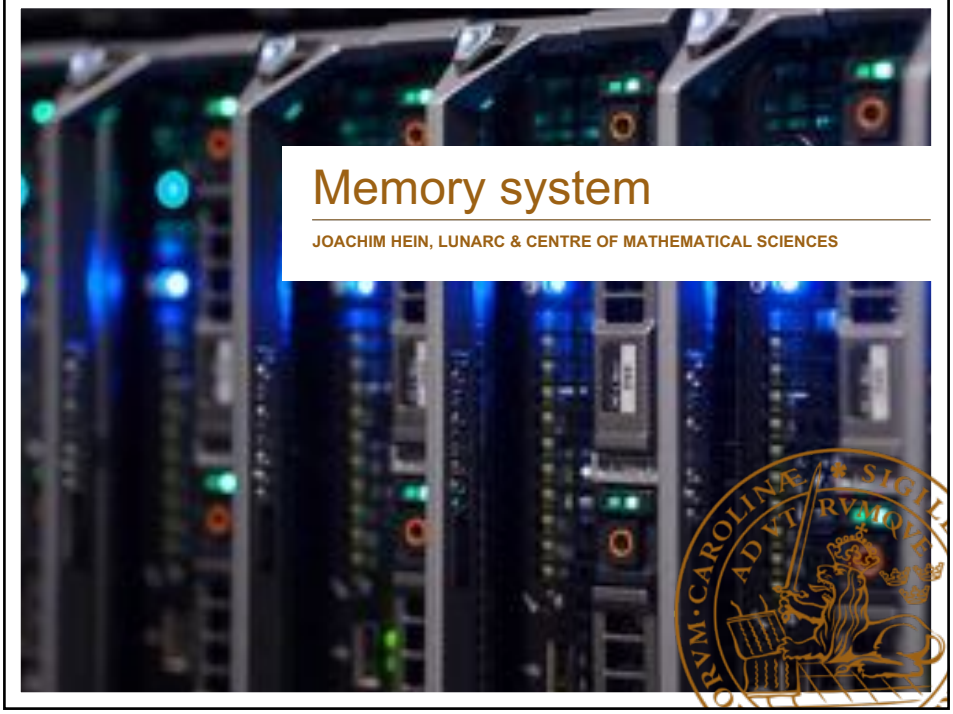
- Most testing is done during user service
  - Shared resources
  - Nodes might be in a sub-optimal state
- Run your tests repeatedly (separate batch scripts)
- When comparing `test_A` and `test_B`, run:

  `test_A, test_B, test_A, test_B, test_A, test_B`
  - Info on stability of timings
  - Handicaps will most likely affect both codes similarly
- **Always question your results!**

Summary

- Overview on timers available
- Best practice recommendation on
  - How to user timers
  - How to conduct benchmarking


SIGILLVM · VNIVERSITATIS · GOTHORVM · CAROLINÆ
AD VTRVMQVE
1666
LUND
UNIVERSITY



Memory system

JOACHIM HEIN, LUNARC & CENTRE OF MATHEMATICAL SCIENCES

Overview

- Memory subsystem
- Shared resources caches and memory bus
- Vector instructions
- How caches work and possible side effects

LUND UNIVERSITY

The Stream Triad test

- Stream: standard benchmark regarding memory performance
- Website: https://www.cs.virginia.edu/stream/
- Contains 4 tests (Copy, Scale, Sum, Triad)
- For illustrations use a modified version (FORTRAN) of Stream Triad test:

```
do i = 1, n
    a(i) = b(i) + s*c(i)
end do
```

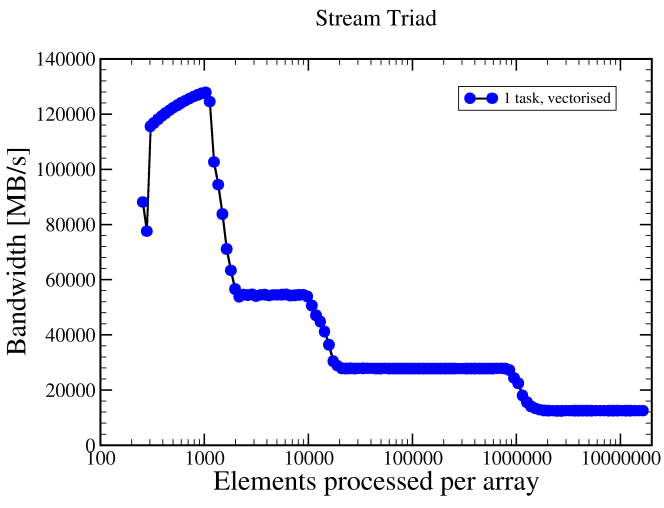
LUND
UNIVERSITY

Benchmarking system

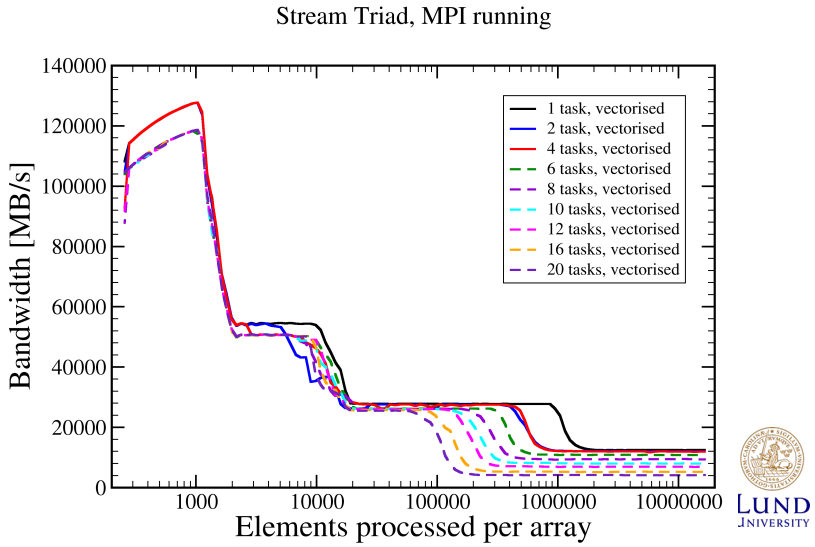
- 2 Intel Xeon E5-2650 v3 (Haswell)
  - 2.3 Ghz
  - 10-core/processor, 20 cores/node
  - Cache/core: 32 kB L1, 256 kB L2, 1.5 MB LLC
- 64 GB RAM per node
  - DDR4-2133
- 4xFDR InfiniBand

- GCC 7.3.0
- OpenMPI 3.1.1

Triad performance for different array sizes (1 task per node)



Running more then one task per node
Tasks synced with `MPI_Barrier()`



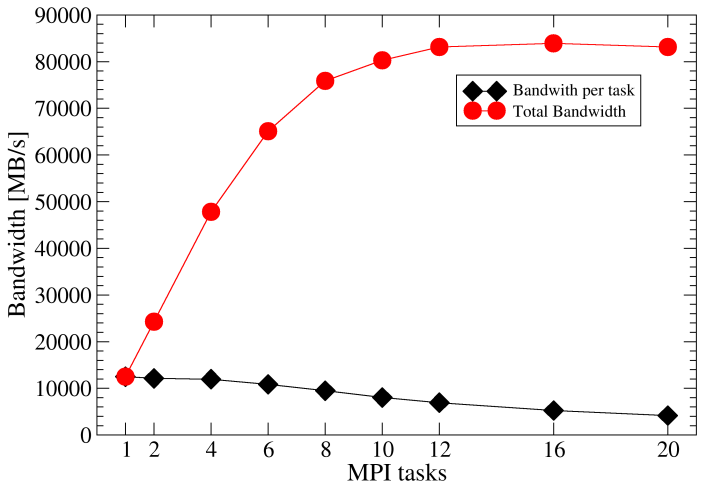
LUND UNIVERSITY

Comments on MPI running

- For 4 or fewer tasks CPU clock up (L1 cache result)
- L1 core private
- L2 not widely shared,
  - Performance sustained
- Low Level Cache (L3) accessible from all 10 cores on Processor
  - Performance sustained
- Main memory performance decreases when more cores used ➛ next slide

LUND UNIVERSITY

Band width on Main memory for different task counts



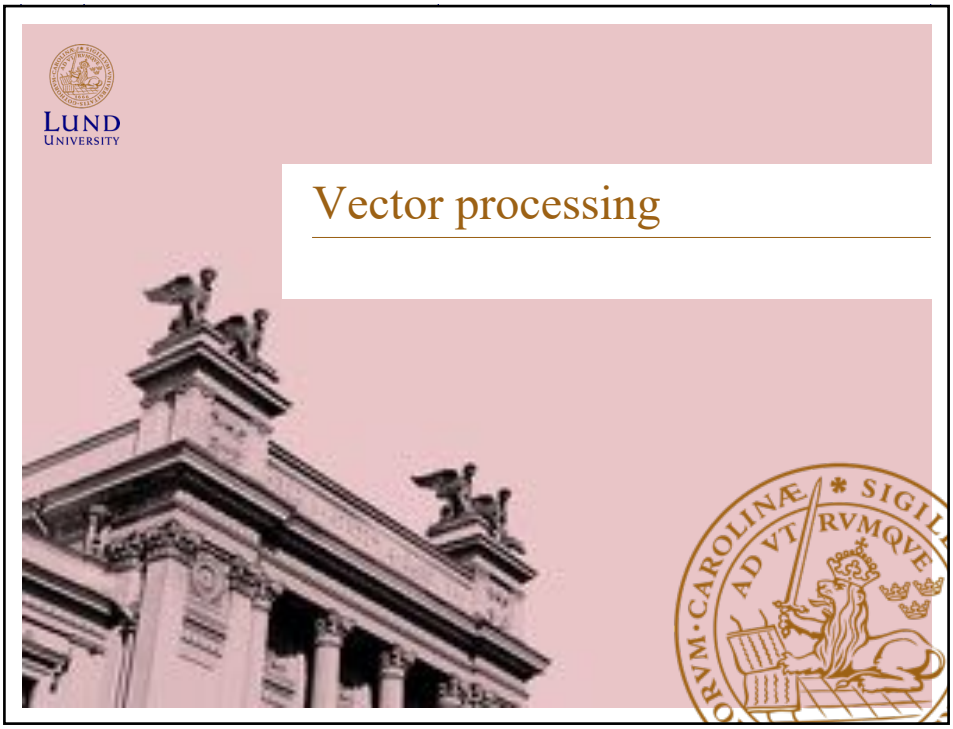
LUND UNIVERSITY

Comments on memory bandwidth

- System can sustain 4 cores
- Utilising up to 8 cores still gets more data from memory
- From 12 cores: the memory busses are completely saturated

- Recommendation:
  - Work on cache memory
  - Avoid reading memory from all tasks/threads at the same time

LUND UNIVERSITY



Modern hardware has wide registers
Overview on x86 system

Instruction set	Register width	Single prec. words	Double prec. words	Typical hardware
SSE, SSE2	128 bit	4	2	modern x86
AVX, AVX2	256 bit	8	4	x86 since 2011
AVX-512	512 bit	16	8	Skylake Knights Landing

- Concept also exists in non-x86 hardware, examples:
  - ARM: NEON
  - IBM Power: VSX

Example: AVX2 FMA instruction

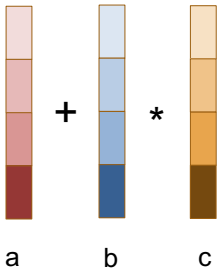
- AVX: 256 bit registers - 4 doubles
- Single instruction - 8 flops:

  $a_1 + b_1 * c_1$

  $a_2 + b_2 * c_2$

  $a_3 + b_3 * c_3$

  $a_4 + b_4 * c_4$



- Enable via compiler option:
  - Without cross compilation:
    - » GCC: `-march=native -O3`
    - » Intel: `-xHost -O3`
  - Cross compilation: explicit specification

Basic example for SIMD deployment

```
do i=1, n
   a(i) = b(i) + c(i)
enddo
```

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Iterations need to be independent

What the compiler will do for you (Simplified)

```
do i=1, n, 4
   a(i  ) = b(i  ) + c(i  )
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
enddo
```

- Execute multiple loop iterations simultaneously
- Iterations need to be independent
- Compiler might need to add a peel

Basic example for SIMD deployment

```
for (i=1; i<n; i++)
  {
   a[i] = b[i] + c[i]
  }
```

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Iterations need to be independent

What the compiler will do for you (Simplified)

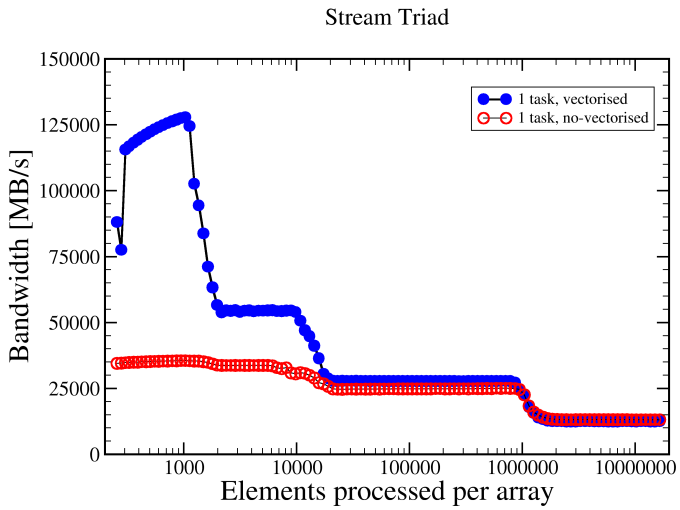
```
for (i=1; i<n; i+=4)
  {
   a[i  ] = b[i  ] + c[i  ]
   a[i+1] = b[i+1] + c[i+1]
   a[i+2] = b[i+2] + c[i+2]
   a[i+3] = b[i+3] + c[i+3]
  }
```

- Execute multiple loop iterations simultaneously
- Reduce loop count accordingly
- Compiler might need to add a peel

Automatic vectorisation

- Modern compilers vectorise many loops automatically
  - Choose right instruction set and optimisation level
    - » GNU: `-O3 -march=native`
    - » Intel: `-O3 -xHost`
  - Compilers can report on vectorisation
    - » GNU: `-fopt-info, -fopt-info-missed`
    - » Intel: `-qopt-report -qopt-report-phase=vec`
- Compiler needs help in complex situations
  - Compiler specific directive ☹
  - OpenMP SIMD construct: portable way to help ☺

Vectorisation of Stream Triad

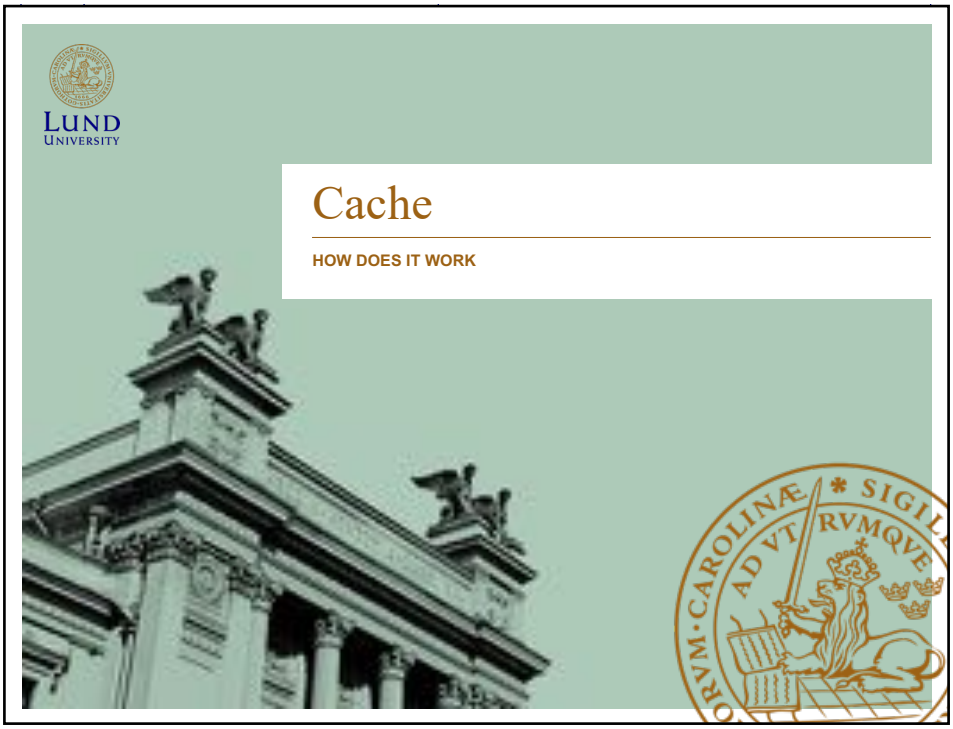


LUND UNIVERSITY

Discussion of vectorisation results

- Stream triad is simple to vectorise for the compiler
  - Specify architecture (GCC 7.3.0 defaults to SSE)
- Performance
  - 4x improvement on L1
  - Substancial boost on L2
  - Marginal boost on LLC
  - No effect on main memory
- To get benefit of vectorisation: work on a low level cache

LUND UNIVERSITY



Words and cacheline

- Cache organised in ”cachelines”
- Each line holds a number of words/double words
- System will always load entire cache lines

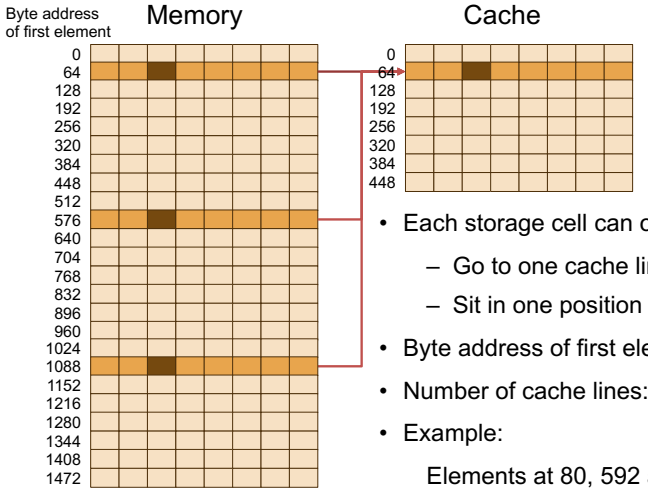
- System has no time to do complex decisions

- Example x86 processor: 64 byte cache line holds
  - 16 words (4 byte each): float/real
  - 8 double words (8 byte each): double /double precision

double	double	double	double	double	double	double	double

LUND
UNIVERSITY

Direct mapped cache
Example: 8 lines of 64 Bytes



- Each storage cell can only:
  - Go to one cache line
  - Sit in one position on the cache line
- Byte address of first element modulo 512
- Number of cache lines: power of 2
- Example:

  Elements at 80, 592 and 1104

  2nd row, 3rd element in cache

LUND UNIVERSITY

Basic use of cache

- You always load a cache line – not an individual element
- Once a cache line is loaded, try to use all elements before loading another one

LUND UNIVERSITY

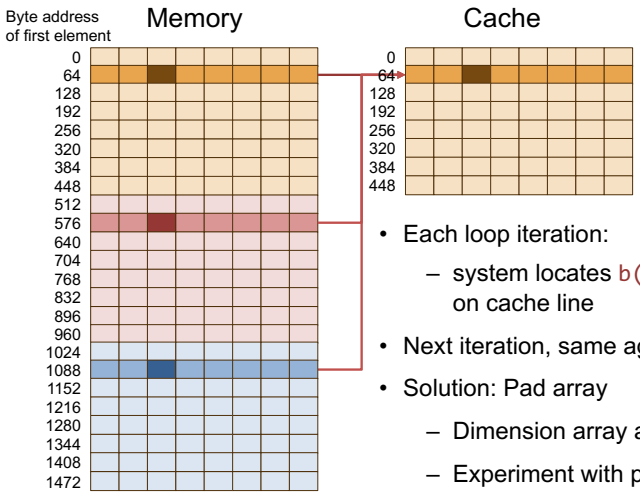
Problem with direct mapped cache

- Consider simple code

```
double precision, dimension(512) :: a, b, c
…
do i=1, 512
      a(i) = b(i) + c(i)
enddo
```

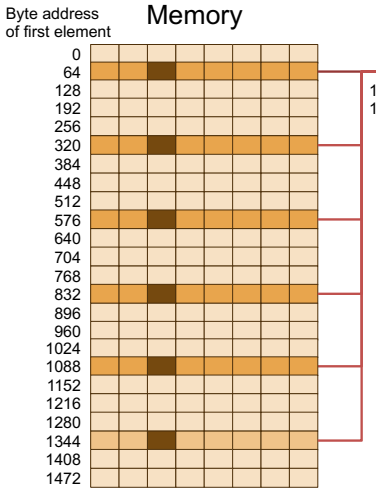
- Assume arrays a, b and c are are located back-to-back in memory
- Poor performance due to cache trashing!

Problem with direct mapped cache Visualisation



- Each loop iteration:
  - system locates `b(i)`, `c(i)` and `a(i)` on cache line
- Next iteration, same again
- Solution: Pad array
  - Dimension array as 512 + p * 8
  - Experiment with p = 1, 2, 3, …
  - Use first 512 elements only

Set associative cache
Example: 2-way set associative



- We have 4 sets - 2 cache lines each
- Every element can sit in one of 2 positions
- Byte address of first element modulo 256
- Example:

  Elements at 80, 336, …

  2nd set, 3rd element in the cache line
- Cache replacement policy:

  Random, Least Recent Use (LRU), First in First out (FIFO)

LUND UNIVERSITY

Comments on set associative cache

- Fixes the issue with vector code for 3-way or better
- Can still encounter poor cache results on matrices
  - E.g. rows in Fortran, columns in C/C++
  - Problem if leading dimension matches product of cacheline size and number of sets
    - » E.g: multiple of 256 elements on prev. picture

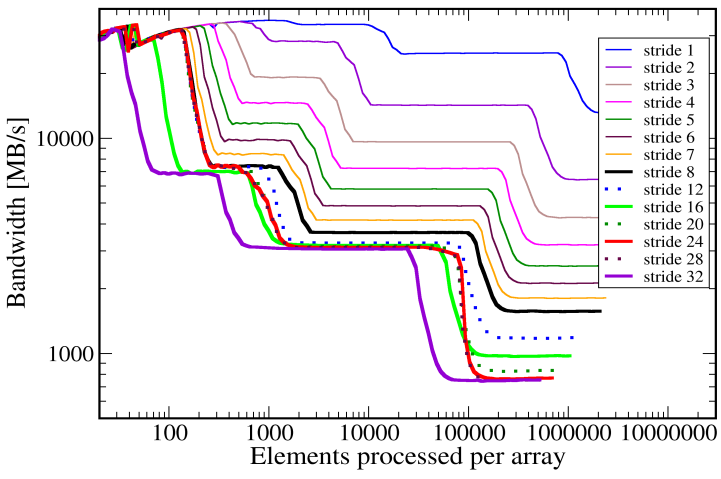
LUND UNIVERSITY

What to expect on current processors

- Modern Intel:

Processor	L1 Data	L2	L3
Haswell	32 kB/core 64 B/line 8-way	256 kB/core 8-way	1.5 MB/core
Broadwell	32 kiB/core 64 B/line 8-way	256 kiB/core 8-way	1.5 MB/core
Skylake (Server)	32 kiB/core 64 B/line 8-way	1 MiB/core 16-way	1.375 MiB/core 11-way
Cascade Lake	32 kiB/core 64 B/line 8-way	1 MiB/core 16-way	1.375 MiB/core 11-way

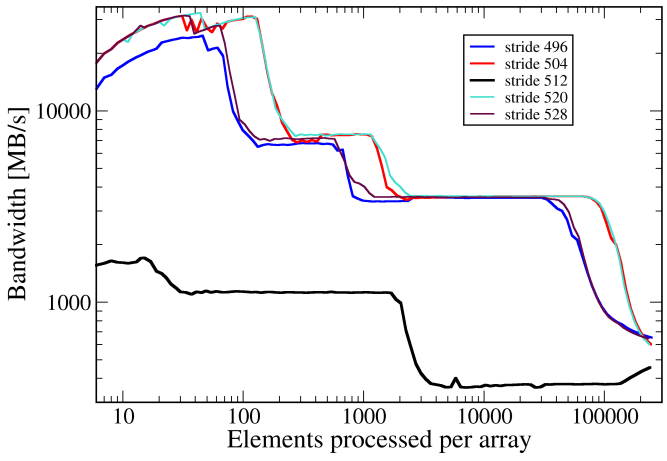
Strided data access



Comments on strided access

- L2, L3 and main memory loose performance
- The caches effectively shrink
- Process stops when the strides are beyond a cacheline
- Though bad things happen for strides of a power of 2

Stride of 512 elements



Observations

- Performance for stride 512 disastrous
  - Cache drops by a almost two orders of magnitude
  - Cache levels do not perform
- Adding/subtracting one or two cache lines improves
  - When adding/subtracting two lines, caches shrink by a factor of 2

Analysis

- On Haswell:
  - L1D of 32 kB, 8 way, 8 doubles per line
    - » L1D turns over after 512 words
    - » For stride 512 utilises only one set ot of 64
    - » The L1D stores only 8 words for stride 512
  - L2 of 256 kB, 8 way, 8 doubles per line
    - » L2 turns over after 4096 words
    - » For stride 512 we utilise eight sets out of 512
    - » The L2 stores only 64 words for stride 512
  - For strides 496, 528 we use every second line

Relevance

- Strided access patterns are common in matrix operations
  - E.g.: operation on
    - » rows in Fortran
    - » columns in C/C++
- Avoid badly strided access by
  - Avoid array leading dimension equal to a power of 2
  - Extending the leading dimension (example soon)
    - » This wastes a bit of memory
    - » Needs care in MPI functions

Extended leading array dimension Fortran

```
integer, parameter :: clsz = 8
double precision, dimension(512+clsz, 1024) :: A

do j = 1, 1024
  do i = 1, 512
     A(i,j) = some_function(i,j)
  enddo
enddo

...
```

Cache line size as parameter

Leading dimension Extended by a cache line

Extended leading array dimension C

```
const int clsz = 8;
double A[1024][512+clsz];

for (int j=1; j<1024; j++)
  for (int i=1; i<512; i++)
     A[j][i] = some_function(j,i);

...
```

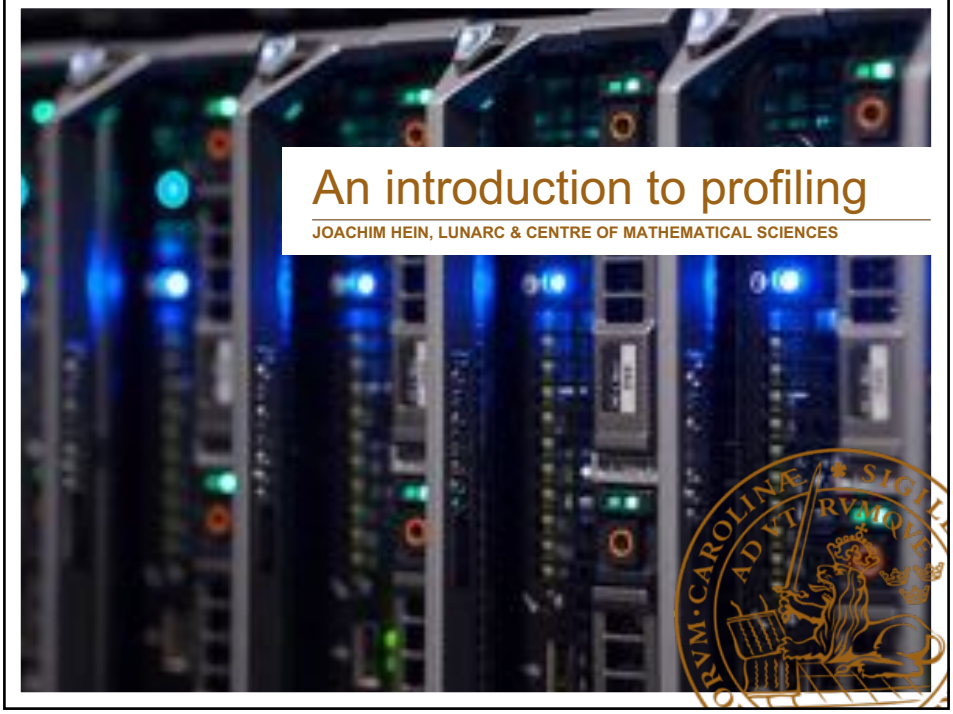
Cache line size as parameter

Leading dimension Extended by a cache line

Summary

- A high flop rate is typically a memory traffic problem:
  - Work on low level cache
  - Use vector instructions of architecture
  - Access all data on a loaded cache line
  - Strides/leading dimensions of a power of 2 are bad


SIGILLVM · VNIVERSITATIS · GOTHORVM · CAROLINÆ
AD VTRVMQVE
1666
LUND
UNIVERSITY



An introduction to profiling

JOACHIM HEIN, LUNARC & CENTRE OF MATHEMATICAL SCIENCES

Overview

- Overview on profilers
  - Timer based
  - Sampling based
- Getting started with ARM map

LUND UNIVERSITY

Measuring performance

Analysing performance

- Questions
  - Where is a code spending time?
  - Why is it spending time there?
  - Is the time spend justified?
  - If not, what can be done to reduce the time spent?

LUND UNIVERSITY

Timers

- Discussed before
- Insert explicitly into the code
- Often build into applications for performance “monitoring”

- The “printf” solution of performance analysis

- For real impact you need a profiling tool
  - Providing automatic instrumentation

Performance profilers

- Two kinds available:

  - Timer based profilers
  - Sampling based profilers

- Both have their strength and weaknesses

Timer based profilers

- Start a timer when entering:
  - Subroutine or function
  - OpenMP construct
- Stop the timer when exiting the above
- User can manually add instrumentation
- Gives very detailed information on task/thread activity
  - Can be utilised for detailed problem reports

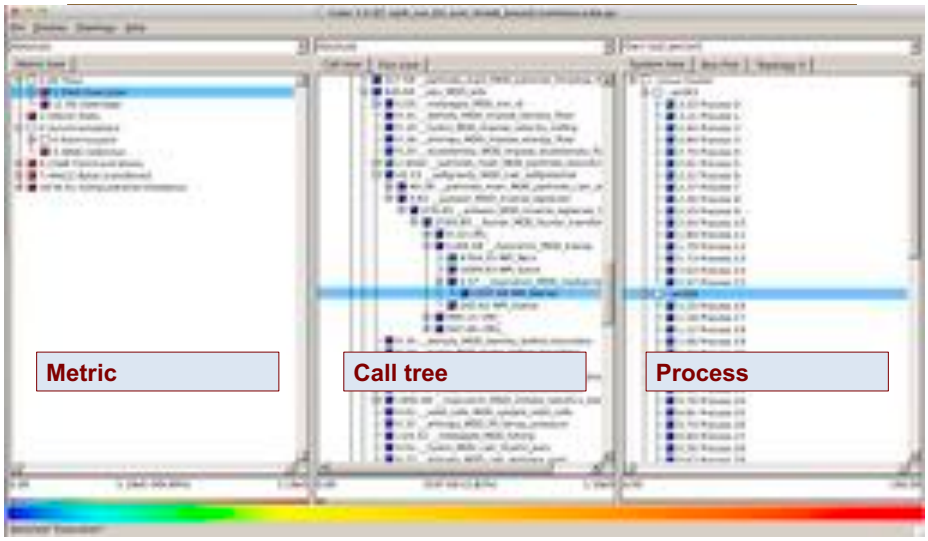
Timer based profilers (cont.)

- Starting/stopping timers gives overhead
- Works well when timed sections are long enough
  - Typical FORTRAN coding
- Often challenged with OO style coding (e.g. C++)
  - Methods are short
  - Overheads unacceptably large (10x in program time)
  - Need exclusion lists (subroutines/functions)
  - Alternatively files compiled without instrumentation
  - Time gets attributed to the caller

Timer based profilers

- The ones I have used (aimed at HPC):
  - Craypat – comes with a Cray
  - Scalasca from FZJ in Germany - free

Scalasca GUI



Sampling based profilers

- Regularly interrupts program execution
  - Typical value about 100 times per second
- Records the call stack
- Puts a “black mark” against the routine it is in
- In the end you get an overview on the black marks

- Information on individual code lines/instructions

Sampling based profilers

- The ones I have used (aimed at HPC):
  - Craypat – comes with a Cray
  - ARM map – requires an expensive license

ARM map

LUND UNIVERSITY

Three steps to a profile

1. Prepare your code
2. Run your application under the profiler
3. Analyse and display the results

LUND UNIVERSITY

Using ARM map

Doing a map analysis

- Compile the code with `-g` and performance optimisation

```
mpicc -g -O3 -march=native -o program program.c
```

- Run the code (jobscript) with

```
map --profile mpirun ./program
```

- View the resulting .map-file with the FORGE gui

LUND UNIVERSITY

What does it do

- MAP is designed and configured with massive parallelism in mind
- Keeping about 1000 evently spaced samples per task
  - Starting with a sample rate of 50Hz – every 20ms
  - Reducing the sample rate as the runtime increases
  - Control starting sample rate via env. variable, e.g.:

    ```
    export ALLINEA_SAMPLER_INTERVAL=5
    ```

  - Beneficial for short tests

Demo

Acknowledgements

- Juan Gao (ARM)
- Patrick Wohlschlegel (ARM)

- Thor Wikfeldt (KTH/PDC)
- Pekka Manninen (CSC in Finland)

LUND UNIVERSITY

LUND UNIVERSITY