BSC Tools
Hands-On

Lau Mercadal, Sandra Méndez, Germán Llort
(tools@bsc.es)
Barcelona Supercomputing Center

Extrae features

- Platforms
  - Intel, Cray, BlueGene, MIC, ARM, Android, Fujitsu Sparc …
- Parallel programming models
  - MPI, OpenMP, pthreads, OmpSs, CUDA, OpenCL, Java, Python …
- Performance Counters
  - Using PAPI interface
- Link to source code
  - Callstack at MPI routines
  - OpenMP outlined routines
  - Selected user functions (Dyninst)
- Periodic sampling

No need to recompile or relink!

- User events (Extrae API)

How does Extrae work?

- Symbol substitution through LD_PRELOAD
  - Specific libraries for each combination of runtimes
    - MPI
    - OpenMP
    - OpenMP+MPI
    - CUDA+MPI
    - …

Recommended

- Dynamic instrumentation
  - Based on Dyninst (developed by U.Wisconsin / U.Maryland)
    - Instrumentation in memory
    - Binary rewriting

- Alternatives
  - Static link (i.e., PMPI, Extrae API)

Extrae on MareNostrum5 (I)

- Log into MareNostrum5 ACC:

```
laptop$ ssh -Y <USER>@alogin1.bsc.es
```

- Extrae is available via modules…

```
alogin1$ module avail extrae
```

- … as are the other BSCTOOLS:

```
alogin1$ module avail paraver
alogin1$ module avail clustering_suite
alogin1$ module avail dimemas
alogin1$ module avail basicanalysis
```

Getting your first trace

- Provided folder **bsctools** in `/gpfs/scratch/nct_362` contains:
  - Application compiled for MPI and MPI+OpenMP (`lulesh2.0_mpi, lulesh2.0_mpi+openmp`)
  - Jobscripts to execute and trace (`job64p.slurm, job8p+8t.slurm, trace.sh`)
  - Configuration of the tracing tool (`extrae.xml`)
  - Already generated tracefiles (`traces/*.{pcf,prv,row}`)

- Copy this folder to your **$HOME** and you are ready to follow this hands-on tutorial

```
alogin1$ cp -R /gpfs/scratch/nct_362/bsctools $HOME
```

Getting an MPI trace with Extrae

Using Extrae in 3 steps

1. **Adapt** your job submission scripts

2. **Configure** what to trace
   - XML configuration file
   - Example configurations at `$EXTRAE_HOME/share/example`

3. **Run** it!

- For further reference check the **Extrae User Guide**:
  - https://tools.bsc.es/doc/html/extrae
  - Also distributed with Extrae at `$EXTRAE_HOME/share/doc`

Step 1: Adapt the job script to load Extrae

- Example of a standard jobscript (without tracing)

job64p.slurm

```
!/usr/bin/env bash

#SBATCH --job-name=lulesh2.0_64p
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --ntasks=64
#SBATCH --time=00:10:00
#SBATCH --qos=acc_debug
#SBATCH --exclusive

ml gcc/11.4.0
ml openmp/4.1.5-gcc

srun ./lulesh2.0_mpi -i 10 -s 65 -p
```

Request resources

Load modules

Run program

Step 1: Adapt the job script to load Extrae

- Jobscript modified to load Extrae and helper script (`extrae/MPI/{job64p.slurm,trace.sh}`)

job64p.slurm

```
!/usr/bin/env bash

#SBATCH --job-name=lulesh2.0_64p
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --ntasks=64
#SBATCH --time=00:10:00
#SBATCH --qos=acc_debug
#SBATCH --exclusive
#SBATCH --constraint-perfparanoid

ml gcc/11.4.0
Ml openmpi/4.1.5-gcc

export TRACE_NAME=lulesh2.0_64p.prv

srun ./trace.sh ./lulesh2.0_mpi -i 10 -s 65 -p
```

Access to Hardware Counters

Trace name

Run with Extrae

trace.sh

```
#!/usr/bin/env bash

module load extrae

export EXTRAE_CONFIG_FILE=./extrae.xml
# For C apps
export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so
# For Fortran apps
#export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitracef.so

## Run the desired program
$*
```

What to trace

Choose a tracing library depending on the app type (see next slide)

Step 1: Which tracing library?

- Choose depending on the application type
  - Navigate to `$EXTRAE_HOME/lib` for all the combinations

Library	Serial	MPI	OpenMP	pthread	CUDA
libseqtrace	✓				
libmpitrace[f][1]		✓			
libomptrace			✓		
libpttrace				✓	
libcudatrace					✓
libompitrace[f] [1]		✓	✓		
libptmpitrace[f] [1]		✓		✓	
libcudampitrace[f] [1]		✓			✓

[1] add suffix "f" in Fortran codes

Step 2: Extrae XML configuration

extrae.xml

```xml
<mpi enabled="yes">
  <counters enabled="yes" />
</mpi>

<openmp enabled="yes">
  <locks enabled="no" />
  <counters enabled="yes" />
</openmp>

<pthread enabled="no">
  <locks enabled="no" />
  <counters enabled="yes" />
</pthread>

<callers enabled="yes">
  <mpi enabled="yes">1-3</mpi>
  <sampling enabled="no">1-5</sampling>
</callers>
```

Instrument the MPI calls
(What is the program doing?)

Instrument the call-stack
(Where in my code?)

Step 2: Extrae XML configuration (II)

extrae.xml

```
  <counters enabled="yes">
    <cpu enabled="yes" starting-set-distribution="1">
      <set enabled="yes" domain="all" changeat-time="0">
        PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_DCM
PAPI_L3_TCM,PAPI_BR_INS,PAPI_BR_MSP,RESOURCE_STALLS:SB
      </set>
    </cpu>
    <network enabled="no" />
    <resource-usage enabled="no" />
    <memory-usage enabled="no" />
  </counters>
```

Select which HWC are measured (How is the machine doing?)

```
alogin1$ papi_best_set \
omnipresent <list-of-counters> \
<list-of-counters>
```

extrae.xml

```
  <buffer enabled="yes">
    <size enabled="yes">5000000</size>
    <circular enabled="no" />
  </buffer>

  <sampling enabled="no" type="default" period="50m"
variability="10m" />

  <merge enabled="yes"
         synchronization="default"
         tree-fan-out="16"
         max-memory="512"
         joint-states="yes"
         keep-mpits="yes"
         sort-addresses="yes"
         overwrite="yes">
    $TRACE_NAME$
  </merge>
```

Extrae buffer size (Flush/memory trade-off)

Additional sampling (Want more details?)

Automatic post-processing to generate the tracefile

Step 3: Run it!

- Submit your job as usual

```
alogin1$ sbatch -A nct_362 --reservation=POP3Tools job64p.slurm
```

- Once finished (check with "`squeue`") you will have the trace (3 files):

```
alogin1$ ls -l
  ...
  lulesh2.0_64p.pcf
  lulesh2.0_64p.prv
  lulesh2.0_64p.row
```

- Any trouble? There's a trace already generated under the "`traces`" folder

- Now let's look into it!

Paraver Installation

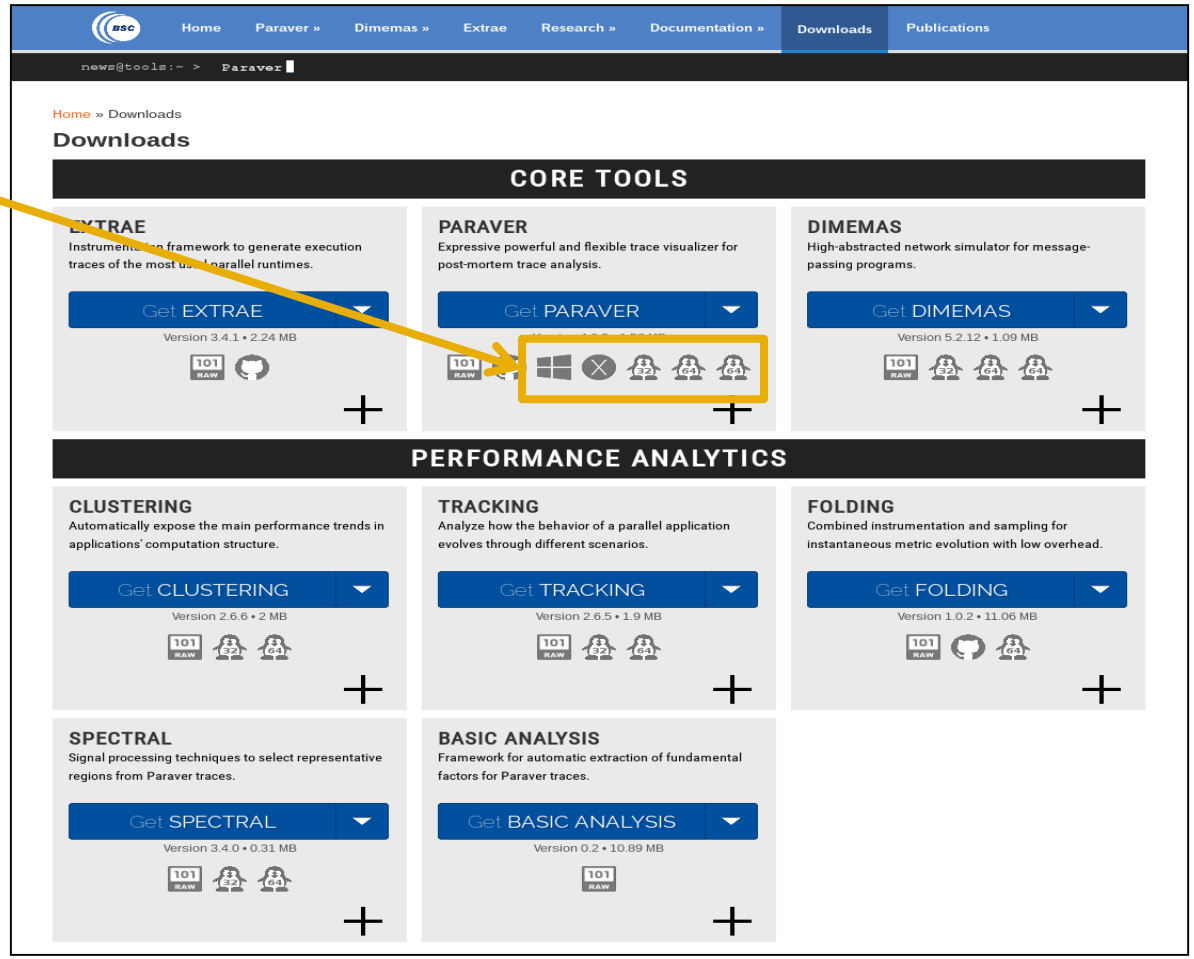
Install Paraver in your laptop

- Download a binary for your OS
  - https://tools.bsc.es/downloads

wxparaver-4.12.0-win.zip

wxparaver-4.12.0-mac.zip

wxparaver-4.12.0-Linux_x86_64.tar.gz

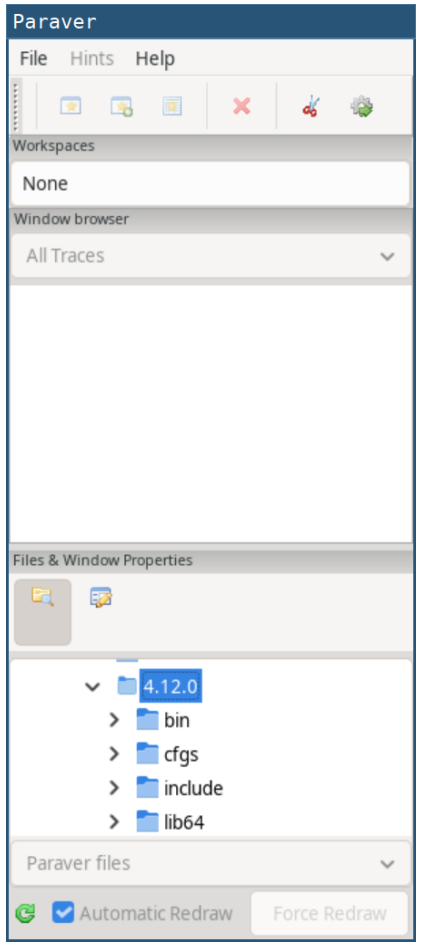


Install Paraver

- Uncompress downloaded package
- Rename the folder:
  - wxparaver-4.12.0-* → paraver
- Start Paraver:
  - Linux: Run the command:

```
laptop$ paraver/bin/wxparaver
```

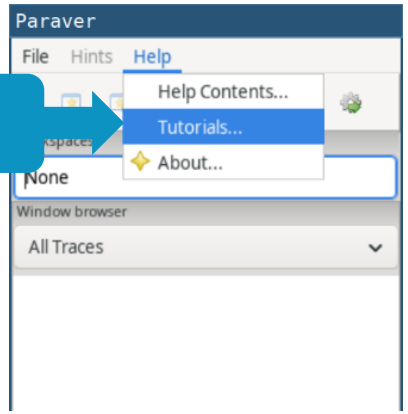
  - Windows: Double-click on paraver/wxparaver.exe
  - MAC: Double click on paraver/wxparaver.app



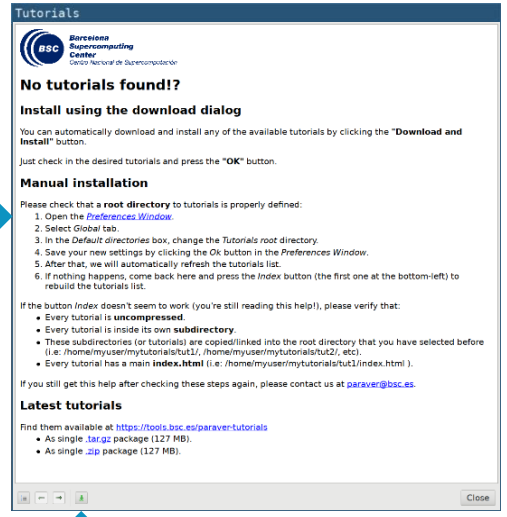
Install Paraver

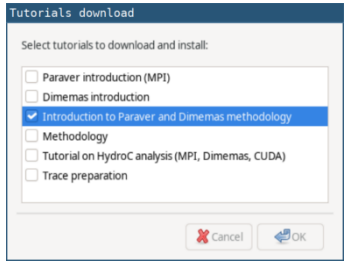
- Download tutorials

Help → Tutorials



Tutorials window pops-up





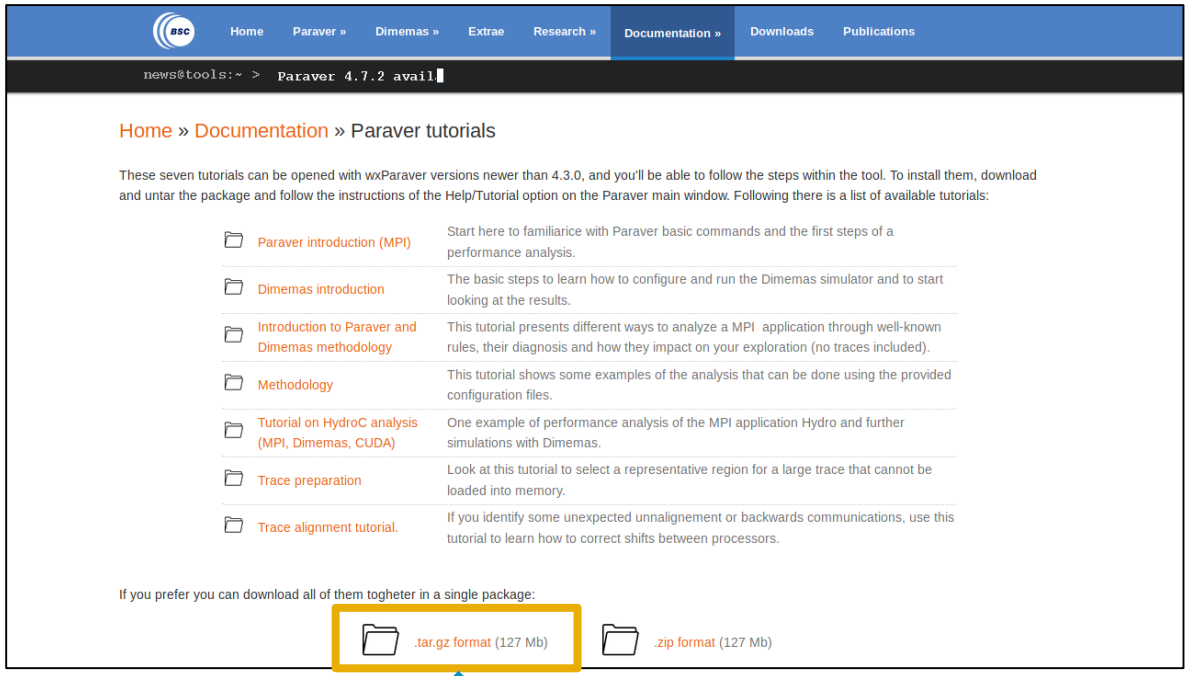
Download and install tutorials

- Follow these tutorials by clicking on the hyperlinks and reading the explanations. When you click on a link, multiple views will open.

Install Paraver tutorials (alternative method)

- Download tutorials archive
  - https://tools.bsc.es/paraver-tutorials

```
paraver-tutorials-20150526.tar.gz
```



All tutorials

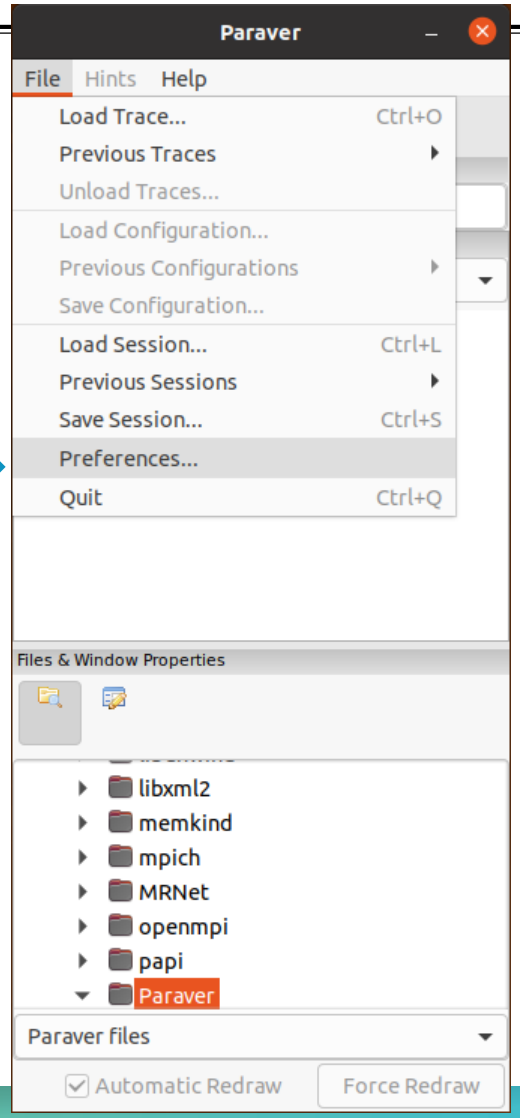
Install Paraver tutorials (alternative method)

- Uncompress downloaded package
- Rename the folders:
  - paraver-tutorials-20150526 → tutorials
- Start Paraver:
  - Linux: Run the command:

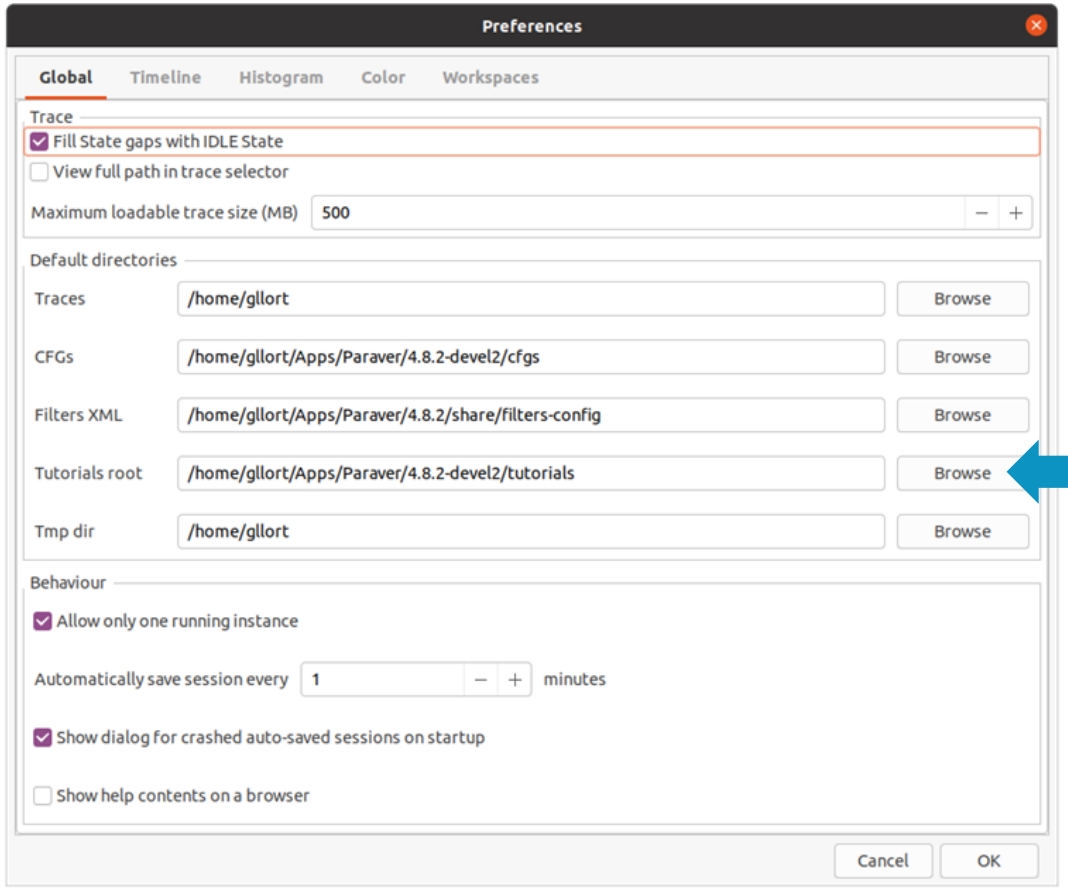
```
laptop$ paraver/bin/wxparaver
```

  - Windows: Double-click on paraver/wxparaver.exe
  - MAC: Double click on paraver/wxparaver.app
- Open File → Preferences

File → Preferences



Install Paraver tutorials (alternative method)



- Setup the “Tutorials root” pointing to your folder “tutorials”

Click and navigate to the uncompressed tutorials folder

Analysing an MPI trace with Paraver

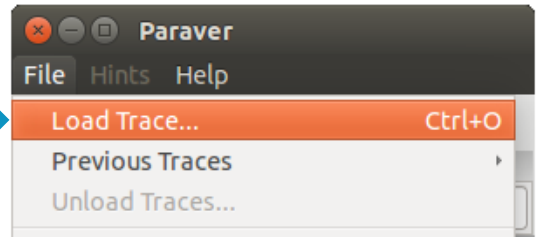
First steps of analysis

- Copy the trace to your computer

```
laptop$ scp <USER>@alogin1.bsc.es:~/bsctools/extrae/MPI/lulesh2.0_64p.* $PWD
```

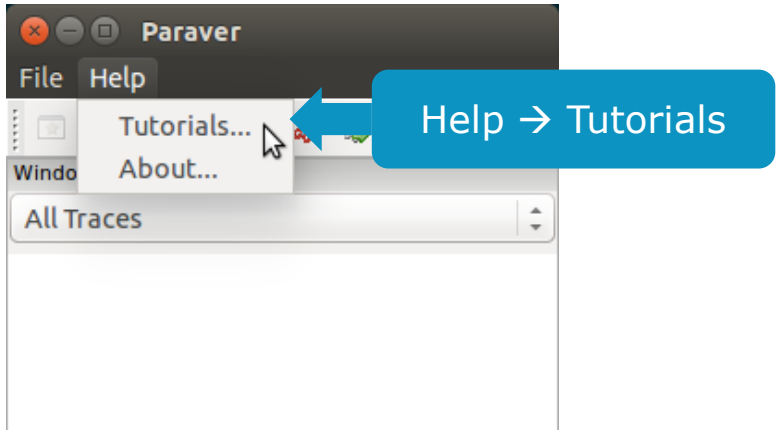
- Load the trace with Paraver

File → Load Trace → navigate to "lulesh2.0_64p.prv"



First steps of analysis

- Follow Tutorial #3
  - Introduction to Paraver and Dimemas methodology



Measure the parallel efficiency

- Click on "mpi_stats.cfg"
  - Check the **Average** for the column labelled "**Outside MPI**"

Tutorials

The first question to answer when analyzing a parallel code is "how efficient does it run?". The efficiency of a parallel program can be defined based on two aspects: the parallelization efficiency and the efficiency obtained in the execution of the serial regions. These two metrics would be the first checks on the proposed methodology.

- To **measure the parallel efficiency** load the configuration file cfgs/mpi/mpi_stats.cfg This configuration pops up a table with %time that every thread spends in every MPI call. Look at the global statistics at the bottom of the outside mpi column. Entry *Average* represents the application parallel efficiency, entry *Avg/Max* represents the global load balance and entry *Maximum* represents the communication efficiency. If any of those values are lower than 85% is recommended to look at the corresponding metric in detail. Open the control window to identify the phases and iterations of the code.
- To **measure the computation time distribution** load the configuration file cfgs/general/2dh_usefulduration.cfg This configuration pops up a histogram of the duration for the computation regions. The computation regions are delimited by the exit from an MPI call and the entry to the next call. If the histogram does not show vertical lines, it indicates the computation time may be not balanced. Open the control window to look at the time distribution and visually correlate both views.
- To **measure the computational load (instructions) distribution** load the configuration file cfgs/papi/2dh_useful_instructions.cfg This configuration pops up a histogram of the instructions for the computation regions. The computation regions are delimited by the exit from an MPI call and the entry to the next call. If the histogram doesn't show vertical lines, it indicates the distribution of the instructions may be not balanced. Open the control window to look at the time distribution and

Close

Parallel efficiency (Avg)

Comm efficiency (Max)

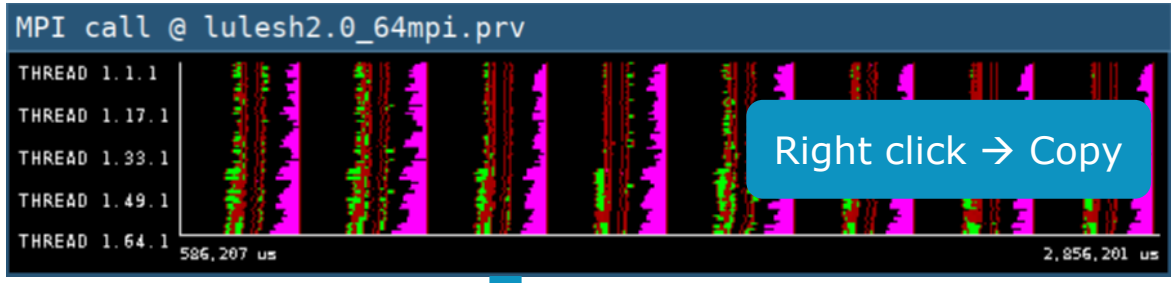
Load balance (Avg/Max)

MPI call profile @ lulesh2.0_64mpi.prv

Default

	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Barrier	MPI_Reduce	MPI_Allredu
THREAD 1.57.1	78.41 %	0.07 %	0.02 %	6.42 %	4.27 %	0.05 %	1.13 %	9.5
THREAD 1.58.1	78.17 %	0.10 %	0.03 %	7.06 %	3.69 %	0.02 %	1.22 %	9.6
THREAD 1.59.1	75.34 %	0.10 %	0.04 %	4.00 %	7.40 %	0.02 %	1.32 %	11.7
THREAD 1.60.1	81.40 %	0.07 %	0.02 %	4.34 %	4.18 %	0.02 %	0.95 %	8.9
THREAD 1.61.1	72.71 %	0.05 %	0.02 %	0.50 %	11.01 %	0.28 %	1.56 %	13.8
THREAD 1.62.1	69.38 %	0.07 %	0.02 %	0.65 %	11.92 %	0.26 %	1.75 %	15.9
THREAD 1.63.1	72.95 %	0.07 %	0.02 %	1.50 %	9.89 %	0.27 %	1.64 %	13.6
THREAD 1.64.1	68.52 %	0.05 %	0.01 %	0.57 %	11.45 %	0.26 %	1.92 %	17.1
Num. Cells	64	64	64	64	64	64	64	
Total	5,340.71 %	5.21 %	2.13 %	93.09 %	346.67 %	2.94 %	60.69 %	545.1
Average	83.45 %	0.08 %	0.03 %	1.45 %	5.42 %	0.05 %	0.95 %	8.5
Maximum	95.29 %	0.14 %	0.06 %	7.06 %	12.26 %	0.28 %	1.92 %	17.1
Minimum	68.52 %	0.03 %	0.01 %	0.11 %	0.62 %	0.00 %	0.00 %	0.7
StDev	7.53 %	0.03 %	0.01 %	1.29 %	3.46 %	0.06 %	0.44 %	3.7
Avg/Max	0.88	0.59	0.58	0.21	0.44	0.16	0.49	0

Focus on the iterative part



Right click → Copy

Drag and drop to zoom on the iterative region

Open Control Window

MPI call profile @ lulesh2.0_64mpi.prv

	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank
THREAD 1.57.1	75.26 %	0.07 %	0.02 %	6.88 %	5.24 %	12.52 %	0.02 %
THREAD 1.58.1	75.88 %	0.10 %	0.03 %	7.63 %	4.01 %	12.34 %	0.02 %
THREAD 1.59.1	72.73 %	0.10 %	0.03 %	3.92 %	8.57 %	14.63 %	0.02 %
THREAD 1.60.1	79.97 %	0.07 %	0.02 %	3.94 %	4.80 %	11.18 %	0.02 %
THREAD 1.61.1	69.81 %	0.05 %	0.01 %	0.69 %	12.24 %	17.19 %	0.02 %
THREAD 1.62.1	65.61 %	0.07 %	0.02 %	0.44 %	13.82 %	20.03 %	0.02 %
THREAD 1.63.1	70.26 %	0.07 %	0.02 %	1.78 %	10.90 %	16.95 %	0.02 %
THREAD 1.64.1	64.69 %	0.05 %	0.01 %	0.41 %	13.18 %	21.65 %	0.02 %
Num. Cells	64	64	64	64	64	64	64
Total	5,216.57 %	4.94 %	2.11 %	103.59 %	384.20 %	687.26 %	1.33 %
Average	81.51 %	0.08 %	0.03 %	1.62 %	6.00 %	10.74 %	0.02 %
Maximum	94.77 %	0.13 %	0.06 %	7.63 %	13.82 %	21.65 %	0.02 %
Minimum	64.69 %	0.03 %	0.01 %	0.12 %	0.77 %	0.90 %	0.02 %
StDev	8.20 %	0.03 %	0.01 %	1.32 %	3.80 %	4.63 %	0.00 %
Avg/Max	0.86	0.58	0.58	0.21	0.43	0.50	0.86

Right click → Paste → Time

Efficiency of iterative region

- 3 numbers to quickly describe the efficiency of your code
  - Parallel efficiency → % of time my program is computing (100% is perfect)
  - Comm efficiency → At least 1 process can finish all communications in 100 - Maximum % of the program's time (100% is perfect)
  - Load balance → Ratio of slow/fast processes (1 is perfectly balanced)
  - Any value below 85% (0.85)?
    - Pay attention there

MPI call profile @ lulesh2.0_64mpi.prv

	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank
THREAD 1.57.1	75.26 %	0.07 %	0.02 %	6.88 %	5.24 %	12.52 %	0.02 %
THREAD 1.58.1	75.88 %	0.10 %	0.03 %	7.63 %	4.01 %	12.34 %	0.02 %
THREAD 1.59.1	72.73 %	0.10 %	0.03 %	3.92 %	8.57 %	14.63 %	0.02 %
THREAD 1.60.1	79.97 %	0.07 %	0.02 %	3.94 %	4.80 %	11.18 %	0.02 %
THREAD 1.61.1	69.81 %	0.05 %	0.01 %	0.69 %	12.24 %	17.19 %	0.02 %
THREAD 1.62.1	65.61 %	0.07 %	0.02 %	0.44 %	13.82 %	20.03 %	0.02 %
THREAD 1.63.1	70.26 %	0.07 %	0.02 %	1.78 %	10.90 %	16.95 %	0.02 %
THREAD 1.64.1	64.69 %	0.05 %	0.01 %	0.41 %	13.18 %	21.65 %	0.02 %
Num. Cells	64	64	64	64	64	64	64
Total	5,216.57 %	4.94 %	2.11 %	103.59 %	384.20 %	687.26 %	1.33 %
Average	81.51 %	0.08 %	0.03 %	1.62 %	6.00 %	10.74 %	0.02 %
Maximum	94.77 %	0.13 %	0.06 %	7.63 %	13.82 %	21.65 %	0.02 %
Minimum	64.69 %	0.03 %	0.01 %	0.12 %	0.77 %	0.90 %	0.02 %
StDev	8.20 %	0.03 %	0.01 %	1.32 %	3.80 %	4.63 %	0.00 %
Avg/Max	0.86	0.58	0.58	0.21	0.43	0.50	0.86

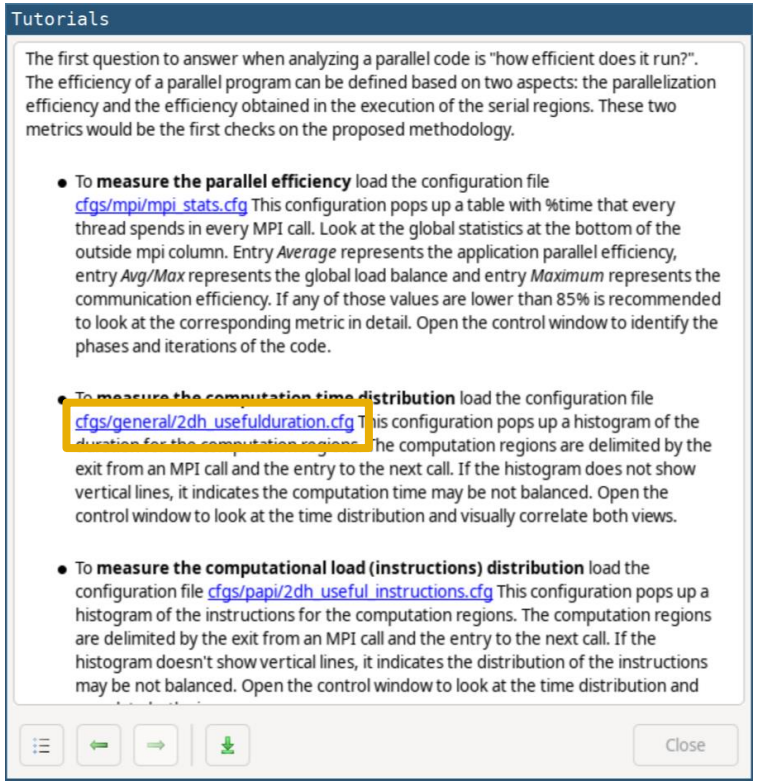
Parallel efficiency (Avg)

Comm efficiency (Max)

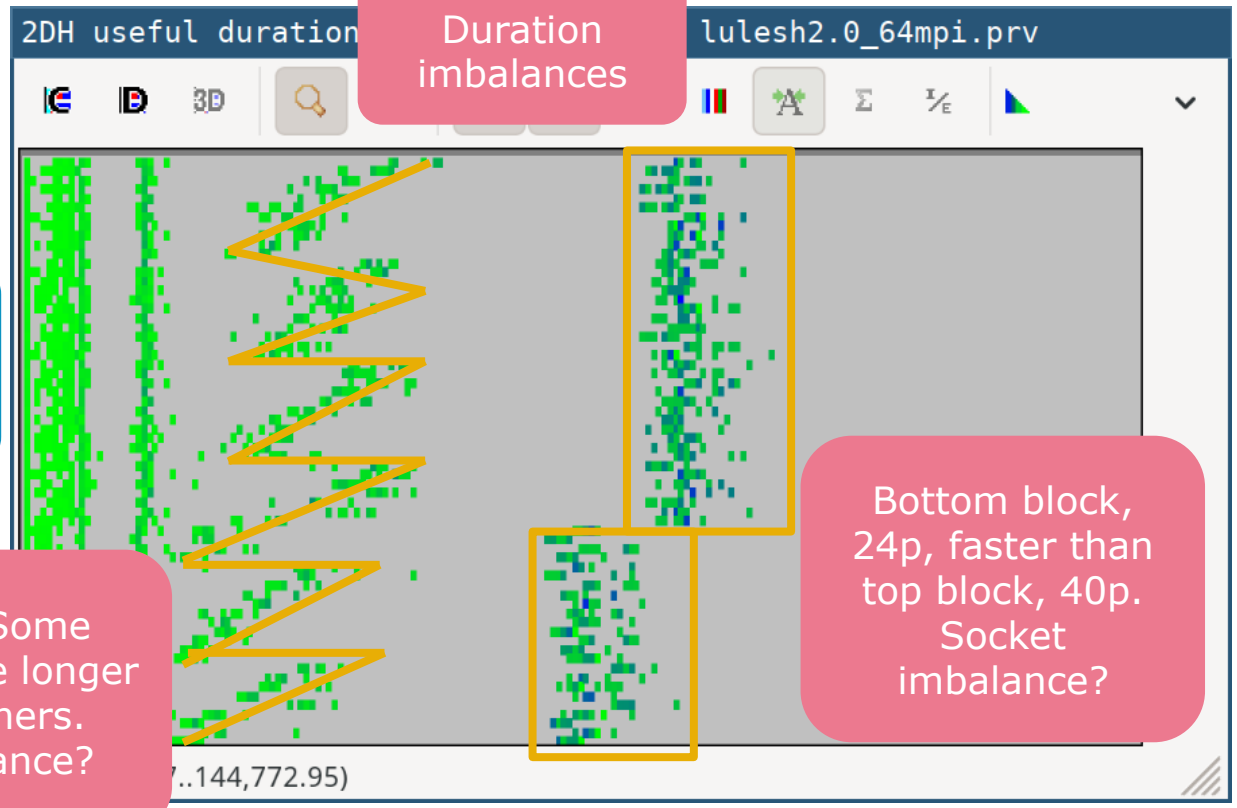
Load balance (Avg/Max)

Computation time distribution

- Click on “2dh_usefulduration.cfg” (2nd link) → Shows **time computing**



Right click → Paste → Time From previous timeline



Duration imbalances

Zig-zag → Some processes take longer than the others. Work imbalance?

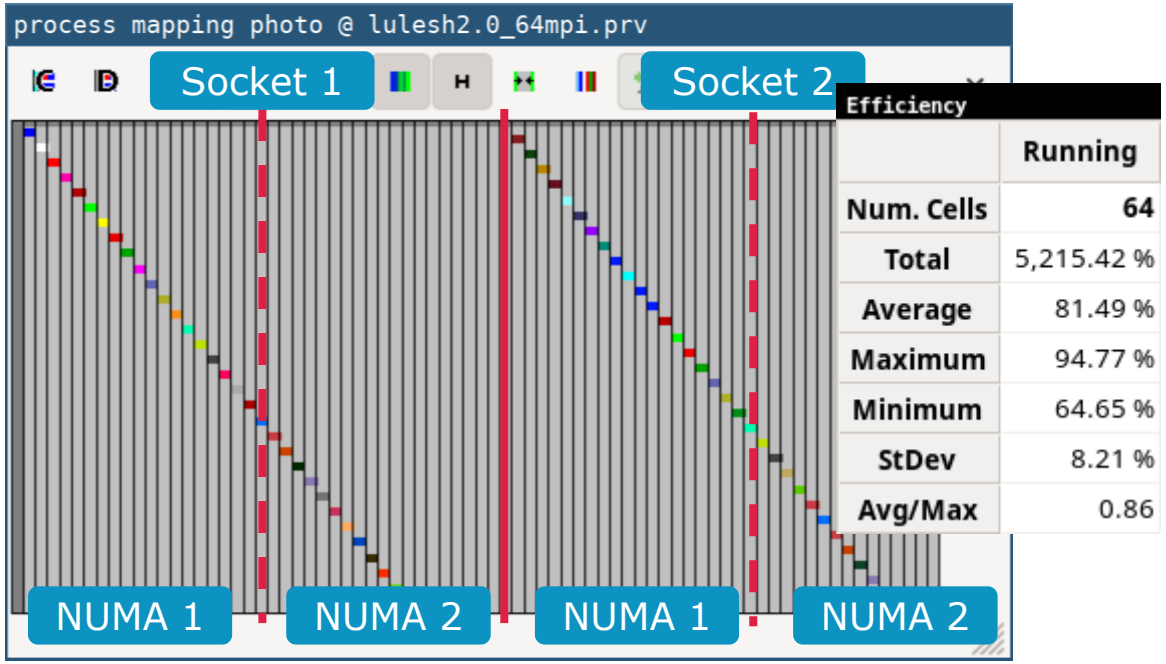
Bottom block, 24p, faster than top block, 40p. Socket imbalance?

Check process mapping

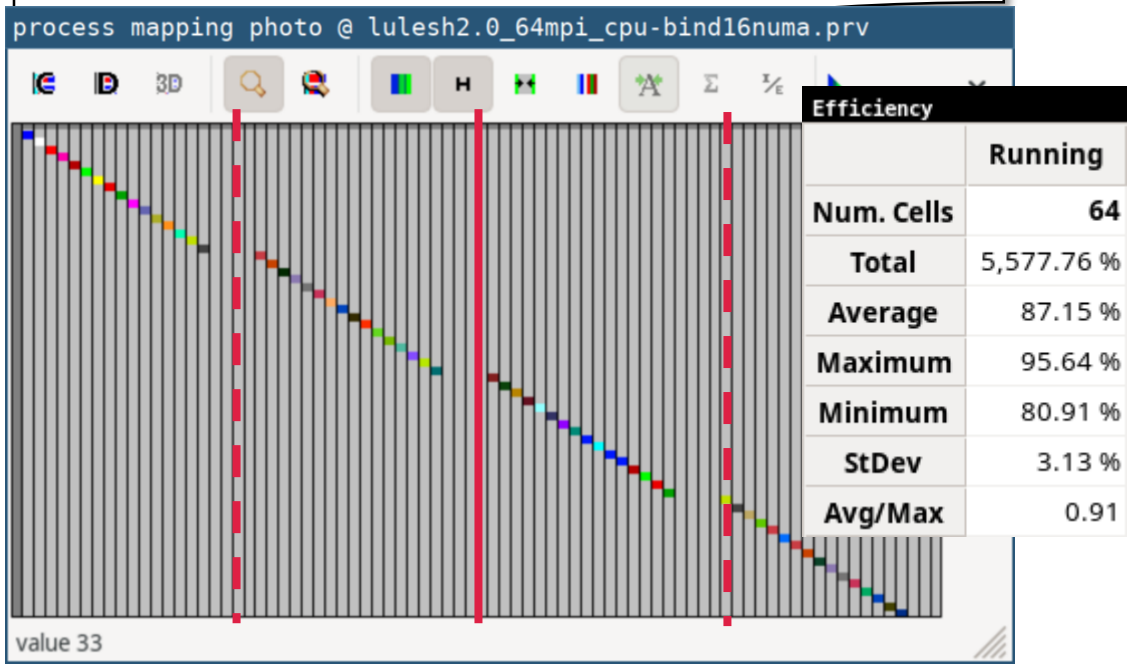
- Hints → Resources → Process Mapping
- MareNostrum5 ACC topology
  - 2 sockets with 2 NUMA nodes each
  - 2nd NUMA node on each socket → half-empty
- Runtime down by ~0.4s (~15%)

job64p-bal.slurm

```
[…]
- srun ./trace.sh ./lulesh2.0_mpi -i 10 -s 65 -p
+ srun ./trace.sh --cpu-
bind=map_cpu:0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,20,21,22
,23,24,25,26,27,28,29,30,31,32,33,34,35,40,41,42,43,44,45,4
6,47,48,49,50,51,52,53,54,55,60,61,62,63,64,65,66,67,68,69,
70,71,72,73,74,75 ./trace.sh ./lulesh2.0-mpi -i 10 -s 65 -p
```



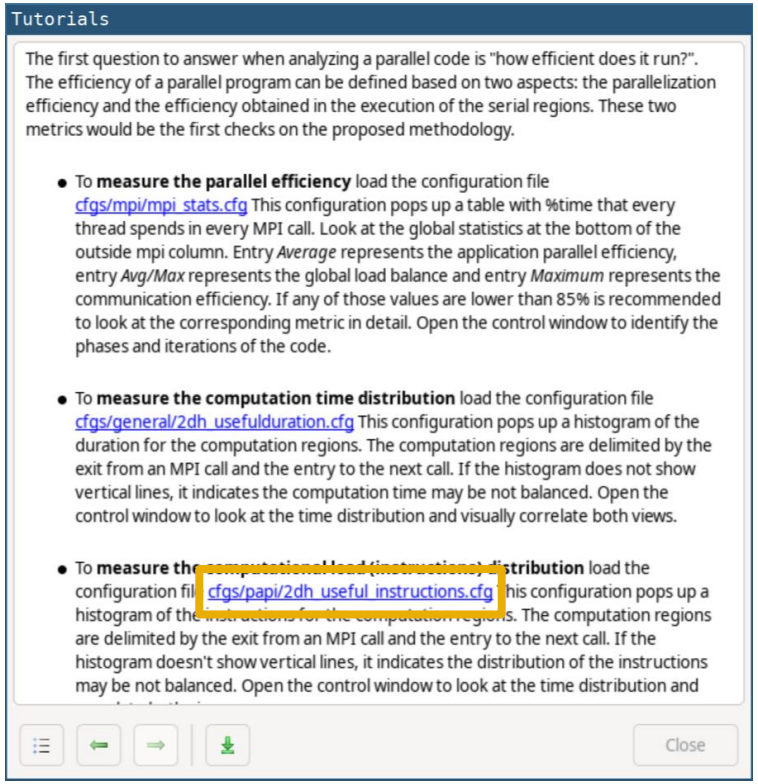
	Running
Num. Cells	64
Total	5,215.42 %
Average	81.49 %
Maximum	94.77 %
Minimum	64.65 %
StDev	8.21 %
Avg/Max	0.86



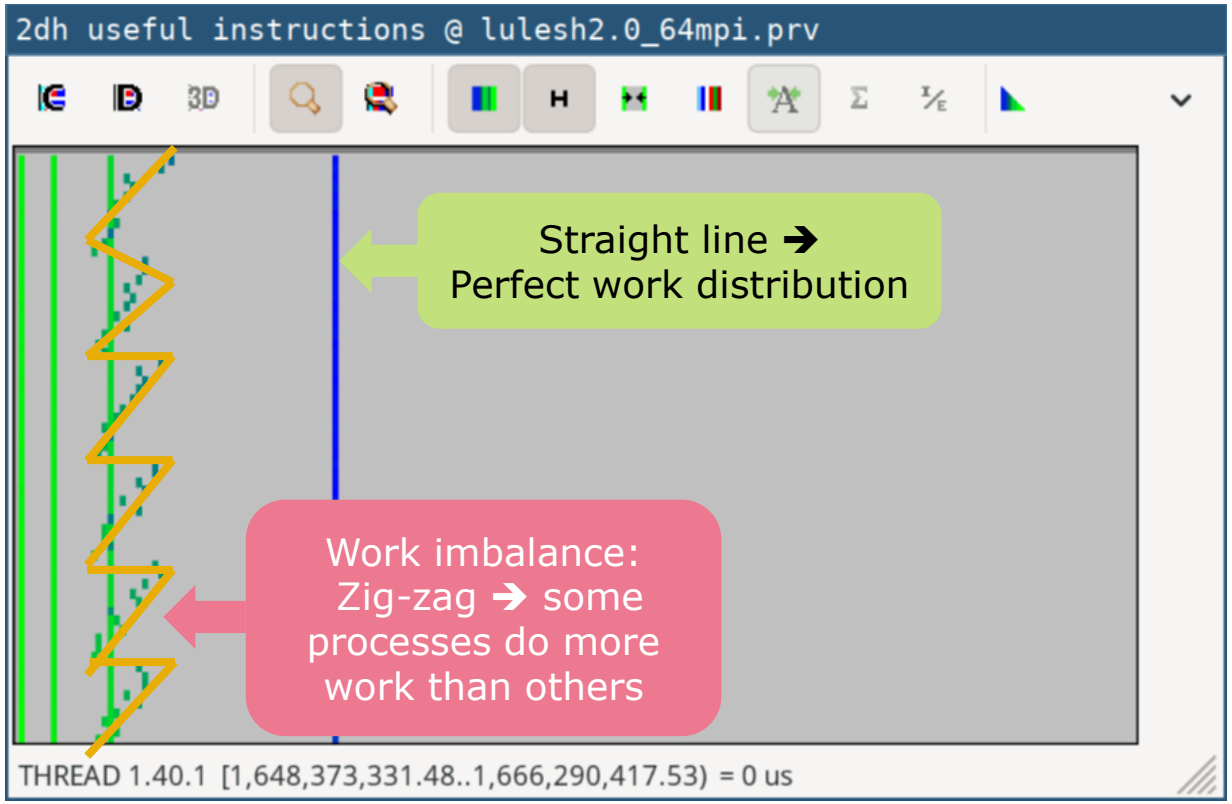
	Running
Num. Cells	64
Total	5,577.76 %
Average	87.15 %
Maximum	95.64 %
Minimum	80.91 %
StDev	3.13 %
Avg/Max	0.91

Computation load distribution

- Click on “2dh_useful_instructions.cfg” (3rd link) → Shows **amount of work**

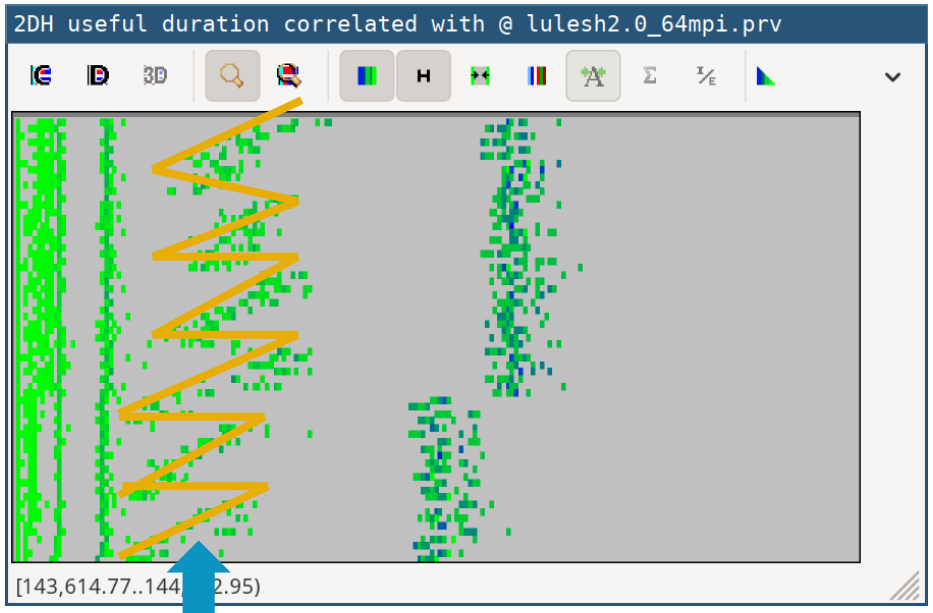


Right click → Paste → Time From previous timeline



Computation load distribution

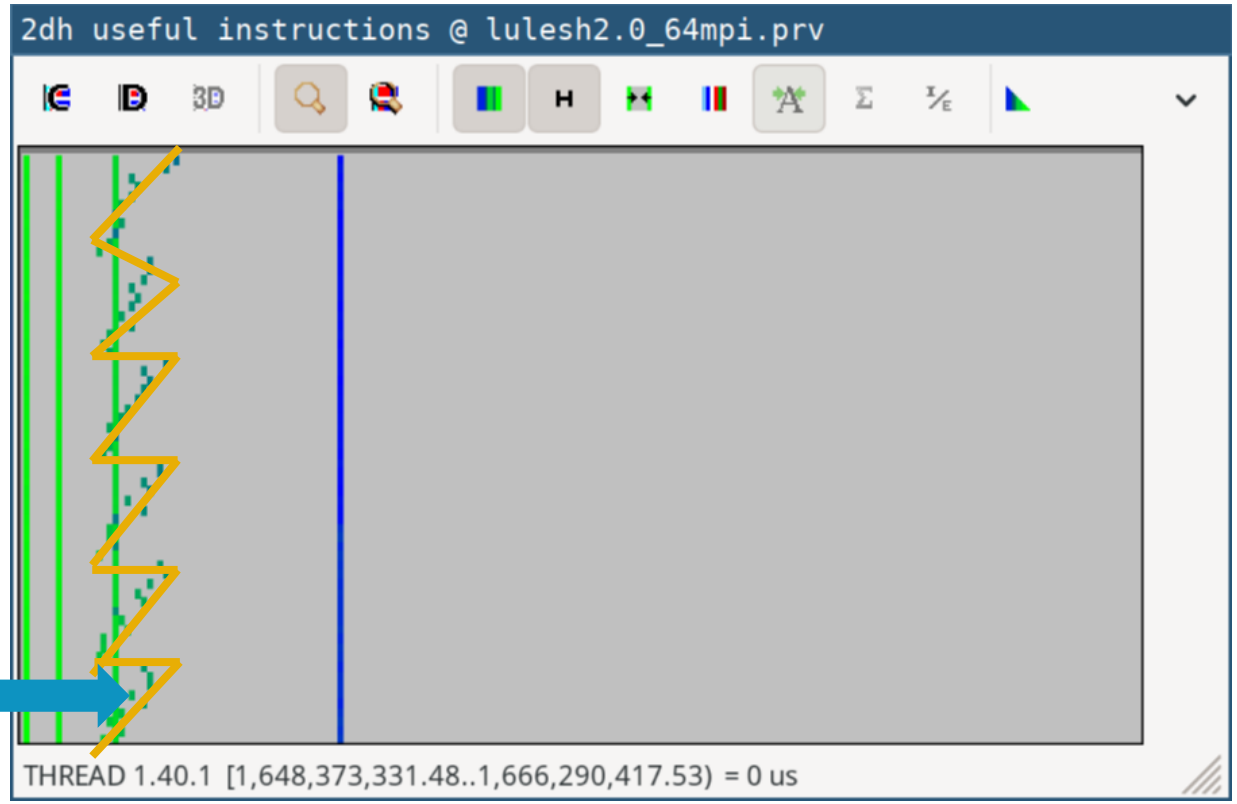
- Comparing the two histograms → **Similar shapes** → Work distribution determines time computing



Effect

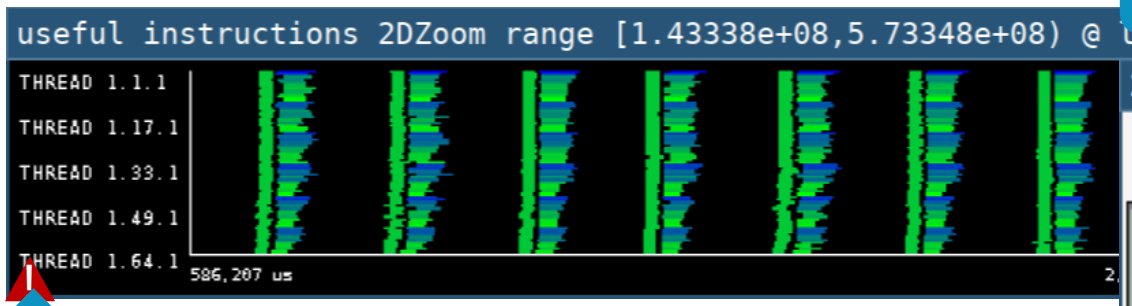
Work imbalance caused by different amount of work

Cause

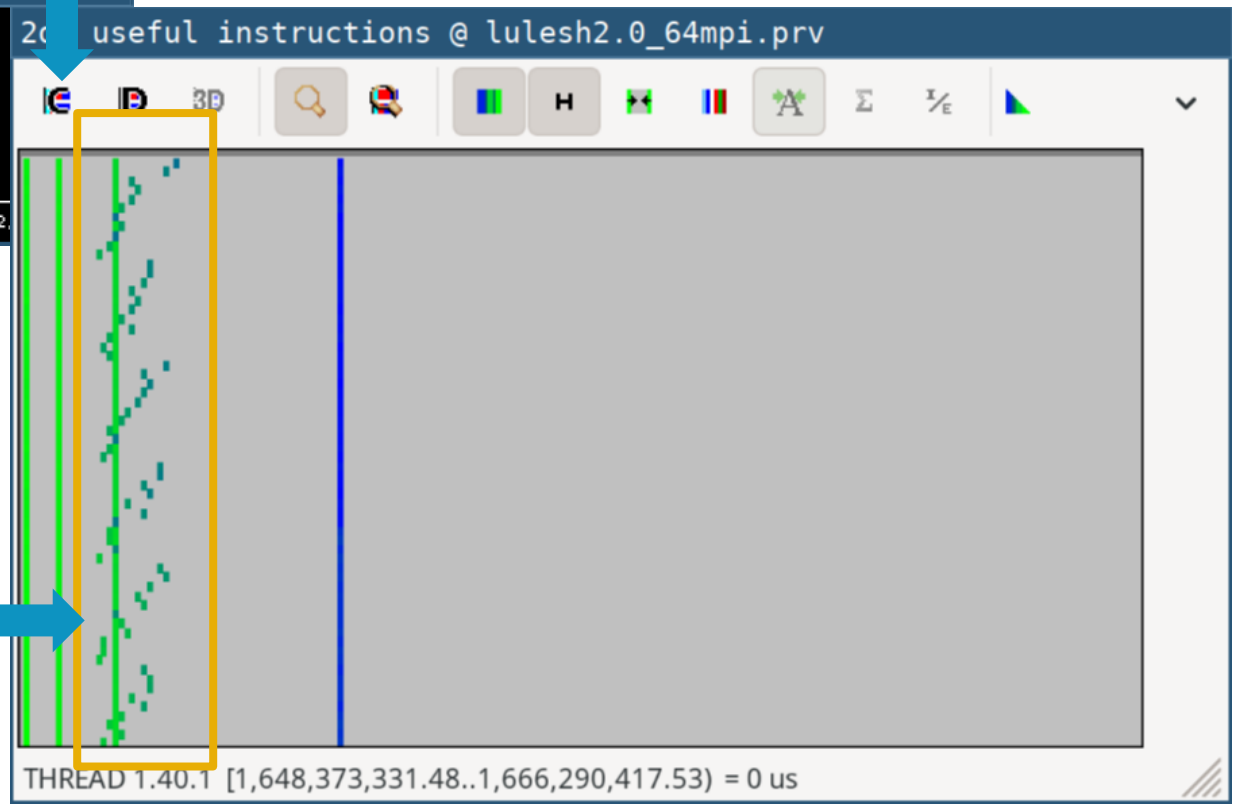


Where does this happen?

- Go from the table to the timeline



Open Filtered Control Window

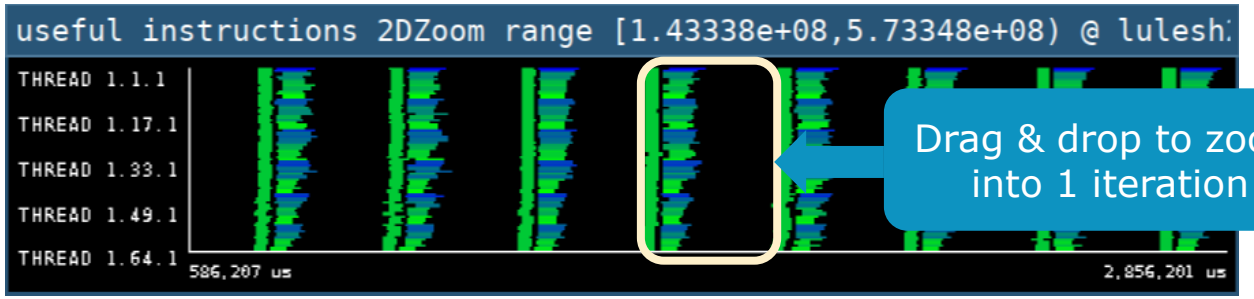


Clicking here always recomputes the scale. Same as: Right click → Fit semantic scale → Fit both

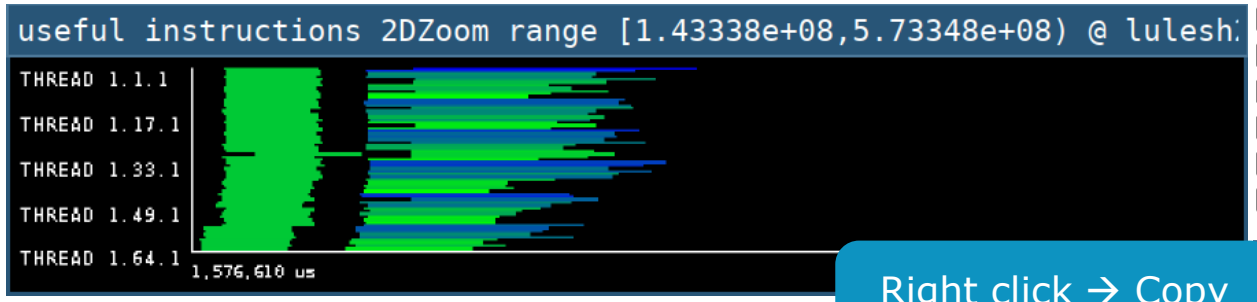
Drag & drop to select imbalanced area

Where does this happen?

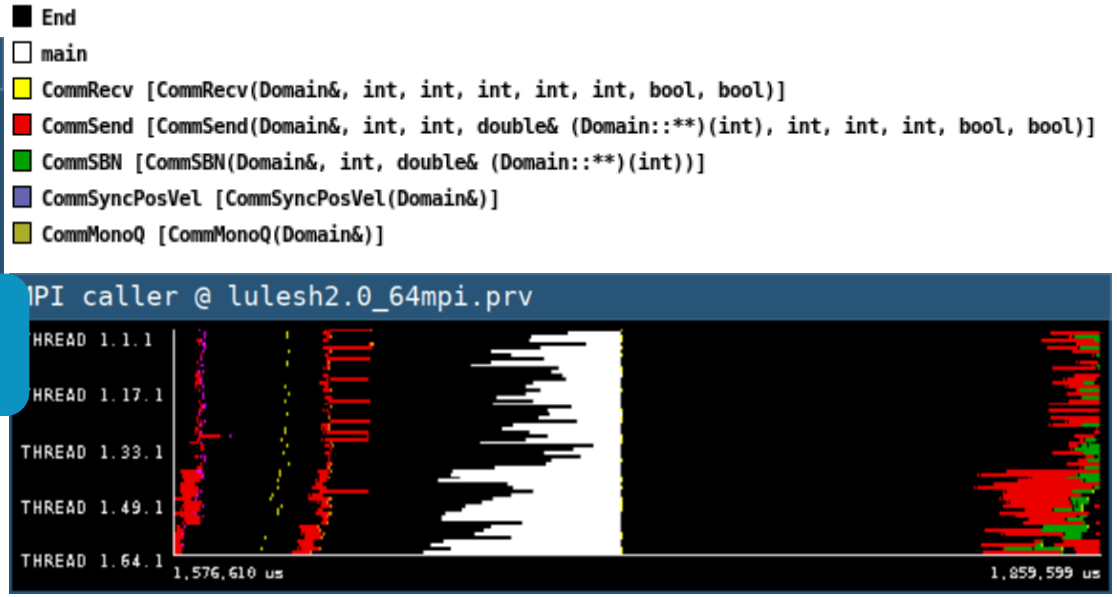
- **Slow** & **Fast** at the same time? → Imbalance



Drag & drop to zoom into 1 iteration

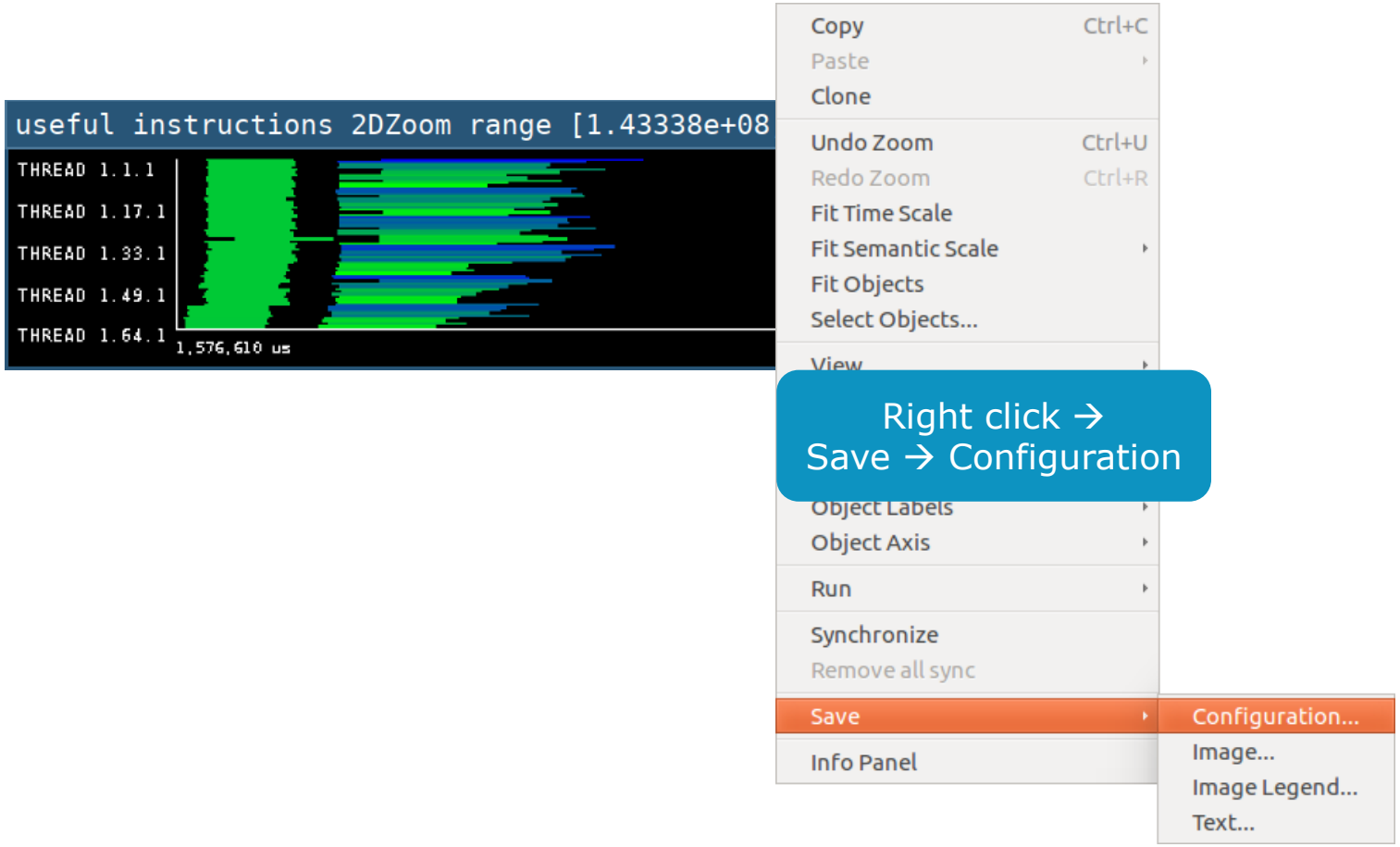


Right click → Copy & Paste Time



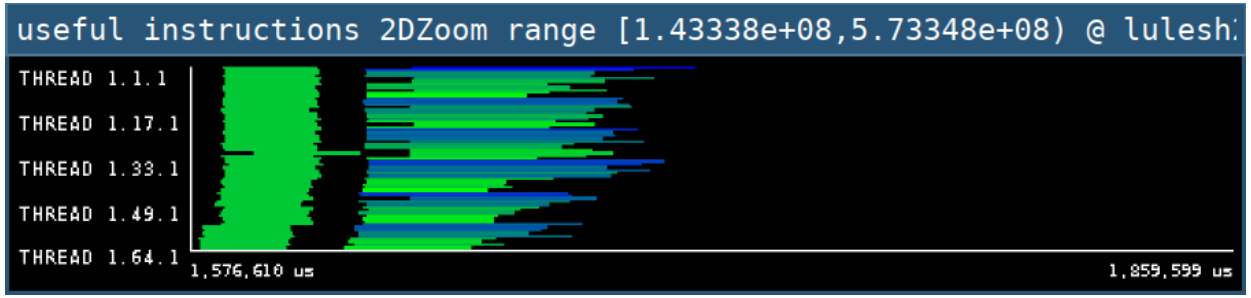
- **Hints → Call stack references → Caller function**

Save CFG's (method 1)

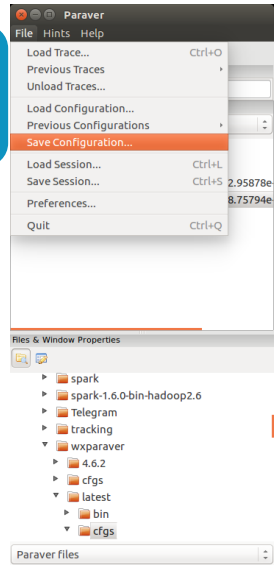


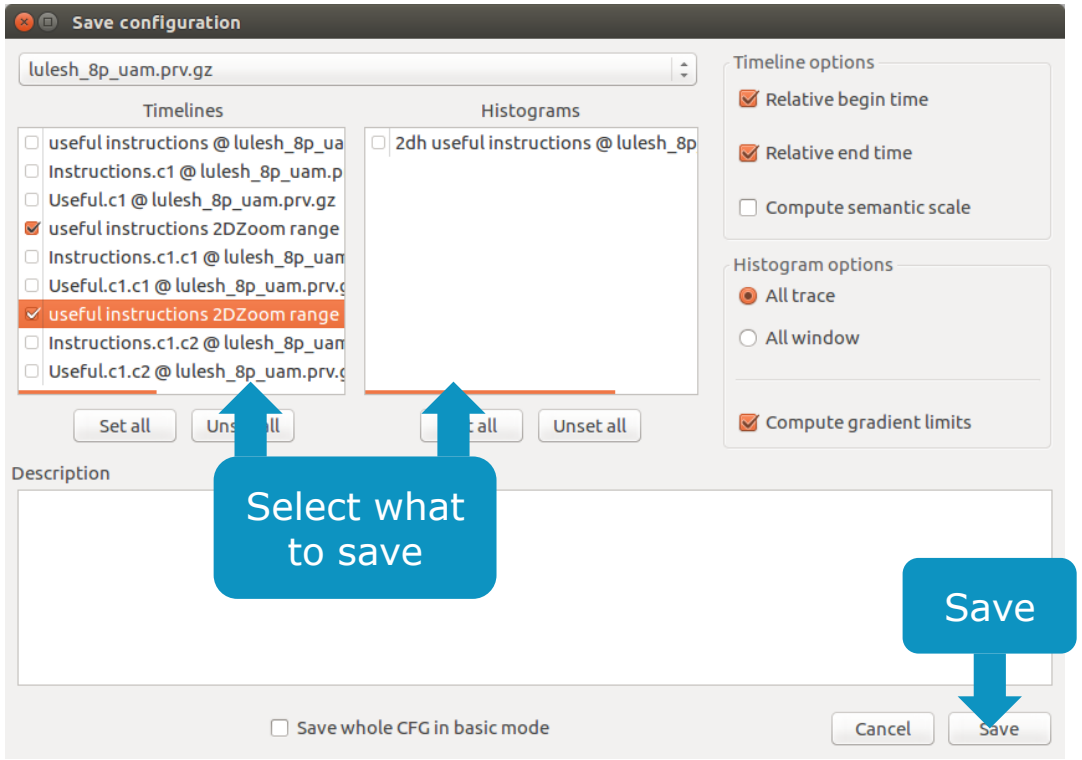
Right click → Save → Configuration

Save CFG's (method 2)



File → Save Configuration



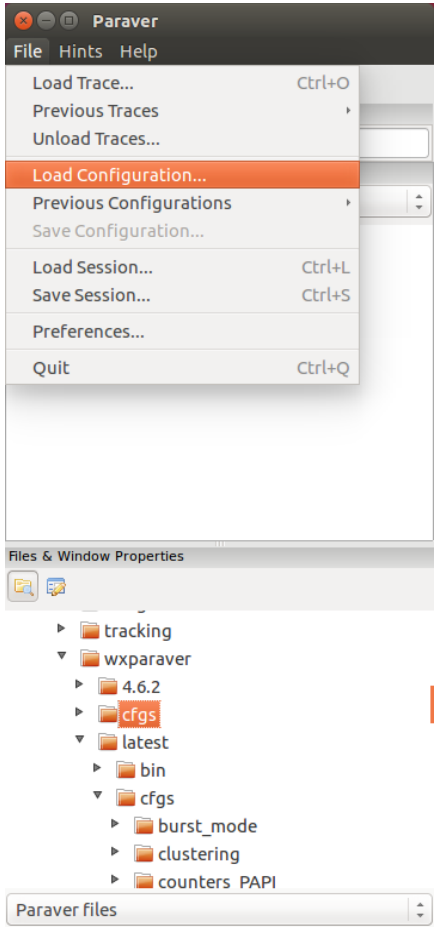


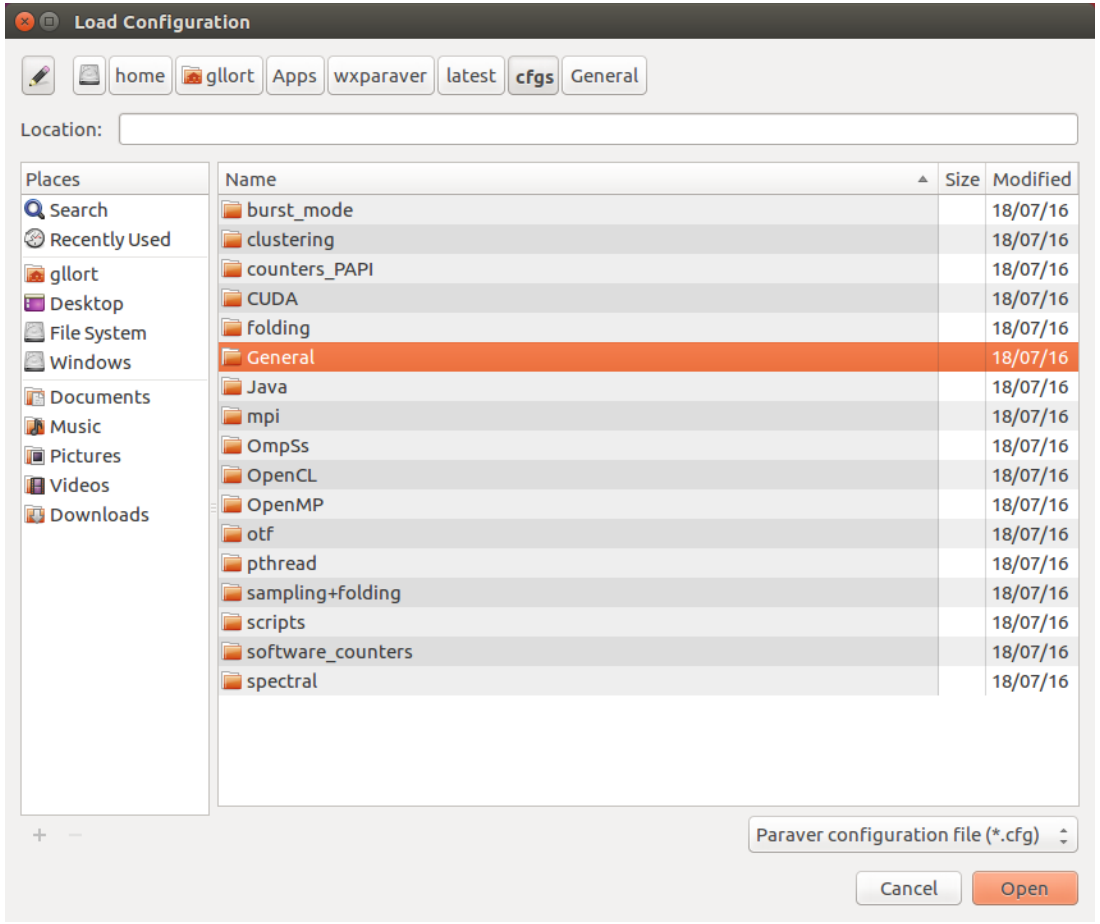
Select what to save

Save

CFG's distribution

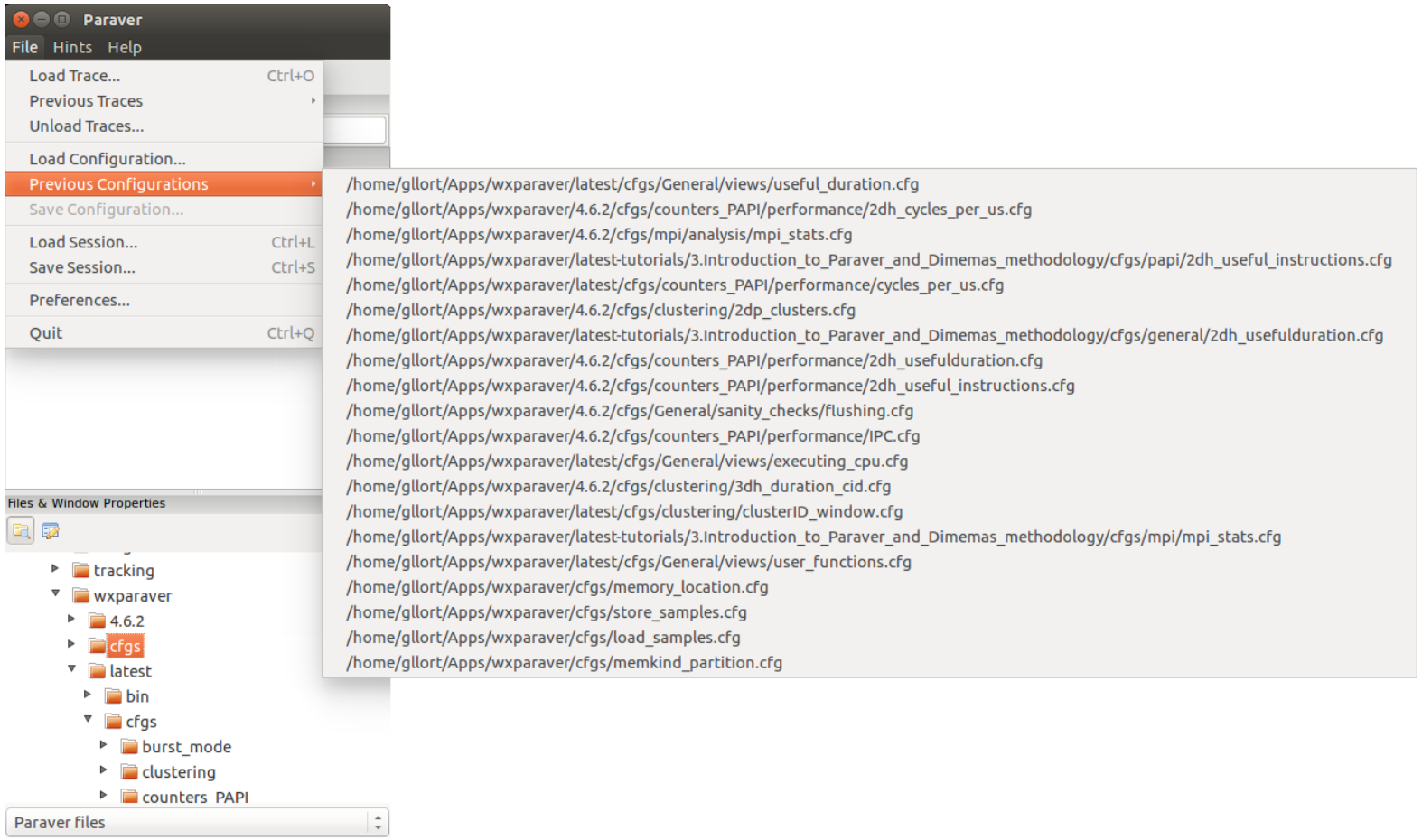
- Paraver comes with many included CFG's ➔ Apply any CFG to any trace!





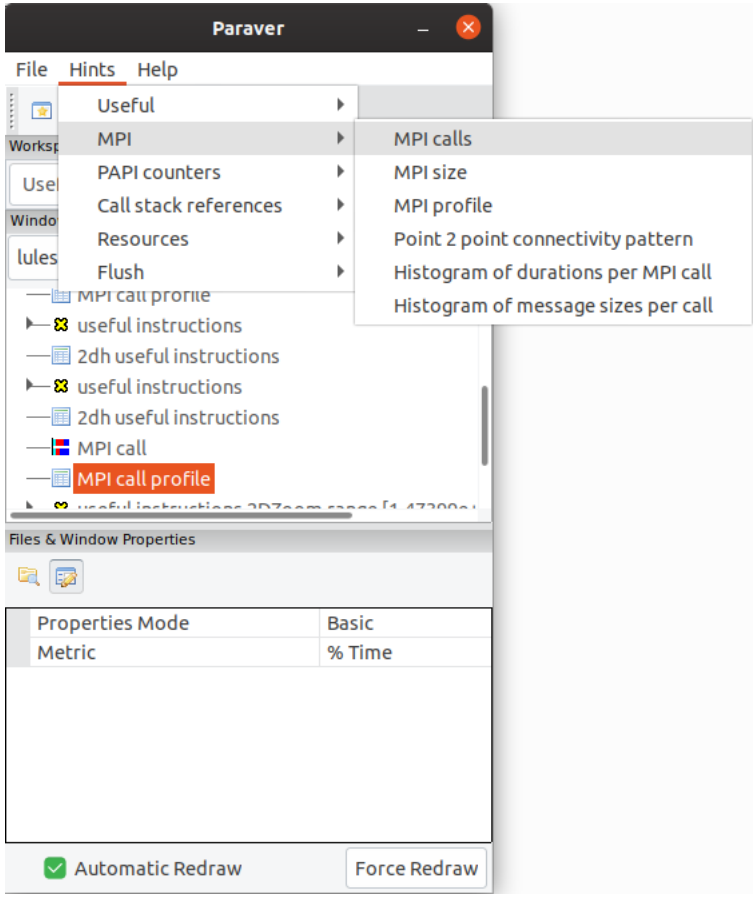
CFG's distribution

- Paraver comes with many included CFG's ➔ Apply any CFG to any trace!



Hints: a good place to start!

- Paraver suggests CFG's based on the contents of the trace



Do it on your code!

- Follow guidelines from slides 7-16 to your own code to get a trace
  - There are more examples of tracing scripts for different programming models under `$EXTRAE_HOME/share/examples`
- Follow guidelines from slides 17-34 to conduct an initial analysis
  - The usual suspects:
    - Parallel Efficiency is low? Load balance issues?
    - Imbalances in the durations of computations?
    - Are these caused by work imbalance?

Getting a hybrid MPI+OMP trace with Extrae

Step 1: Adapt the job script to load Extrae

- Jobscript modified to use the MPI+OPENMP binary (`extrae/MPI+OMP/job8p+8t.slurm`)

job8p+8t.slurm

```
!/usr/bin/env bash

#SBATCH --job-name=lulesh2.0_8p+8t
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --ntasks=64
#SBATCH --time=00:10:00
#SBATCH --qos=acc_debug
#SBATCH --exclusive
#SBATCH --constraint-perfparanoid

export OMP_NUM_THREADS=8

export TRACE_NAME=lulesh2.0_8p+8t.prv

srun ./trace.sh ./lulesh2.0_mpi+openmp -i 10 -s 65 -p
```

Access to Hardware Counters

8 OpenMP threads

Run with Extrae

Step 1: Adapt the job script to load Extrae

- Changes in launcher and configuration file (`extrae/MPI+OMP/{trace.sh,extrae.xml}`)

job8p+8t.slurm

```
!/usr/bin/env bash

#SBATCH --job-name=lulesh2.0_8p+8t
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --ntasks=64
#SBATCH --time=00:10:00
#SBATCH --qos=acc_debug
#SBATCH --exclusive
#SBATCH --constraint-perfparanoid

export OMP_NUM_THREADS=8

export TRACE_NAME=lulesh2.0_8p+8t.prv

srun ./trace.sh ./lulesh2.0_mpi+openmp -i 10 -s 65 -p
```

trace.sh

```
#!/usr/bin/env bash

module load extrae

export EXTRAE_CONFIG_FILE=./extrae.xml
# For MPI+OPENMP apps
- export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so
+ export LD_PRELOAD=${EXTRAE_HOME}/lib/libompitrace.so

## Run the desired program
$*
```

Tracing library for MPI + OpenMP

extrae.xml

```
 - <openmp enabled="no">
 + <openmp enabled="yes">
     <locks enabled="no" />
     <counters enabled="yes" />
   </openmp>
```

Enables OpenMP instrumentation

Step 3: Run it!

- Submit your job as usual

```
alogin1$ sbatch -A nct_362 --reservation=POP3Tools job8p+8t.slurm
```

- Once finished (check with "`squeue`") you will have the trace (3 files):

```
alogin1$ ls -l
  ...
  lulesh2.0_8p+8t.pcf
  lulesh2.0_8p+8t.prv
  lulesh2.0_8p+8t.row
```

- Any trouble? There's a trace already generated under the "`traces`" folder

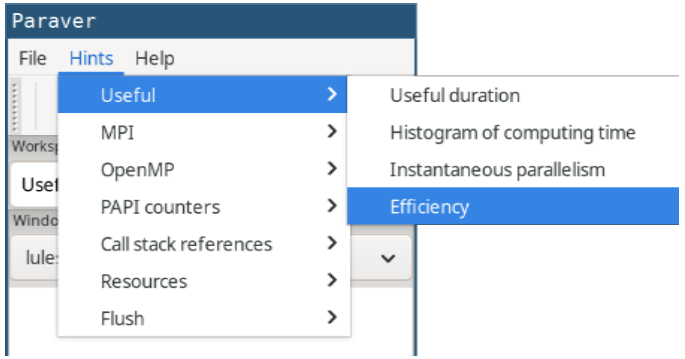
- Now let's look into it!

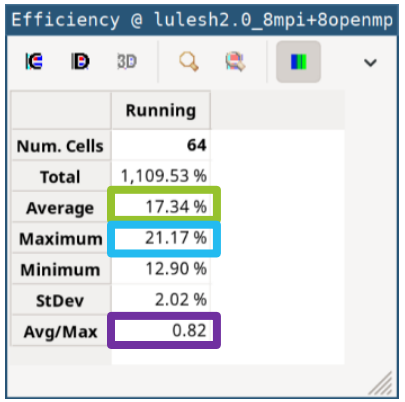
Analysing an MPI+OpenMP trace with Paraver

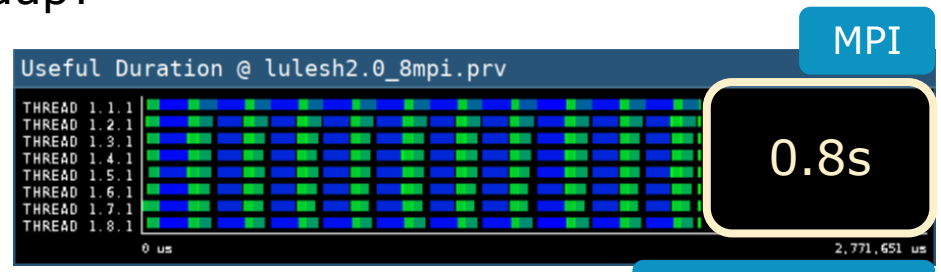
How did it go?

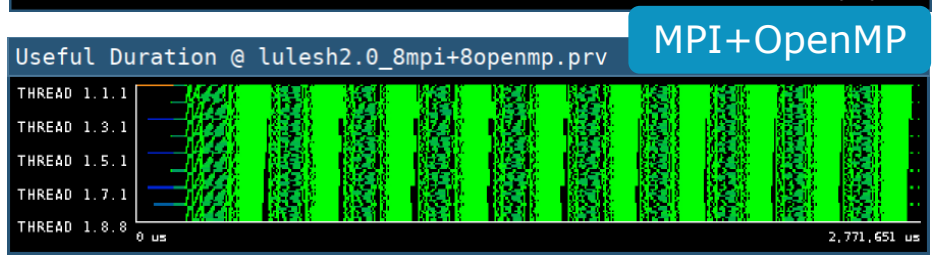
- We added 8 threads per process, did we achieve an 8x speedup?
  - NO!
  - Runtime of MPI version: 1.9s
  - Runtime of hybrid version: 2.7s

- What happened?



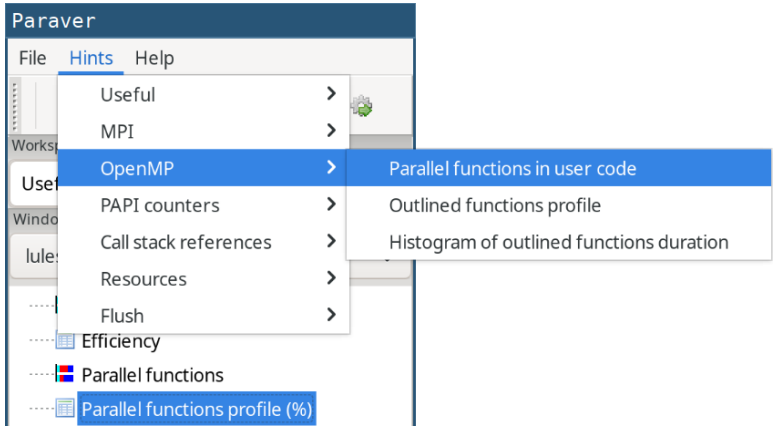


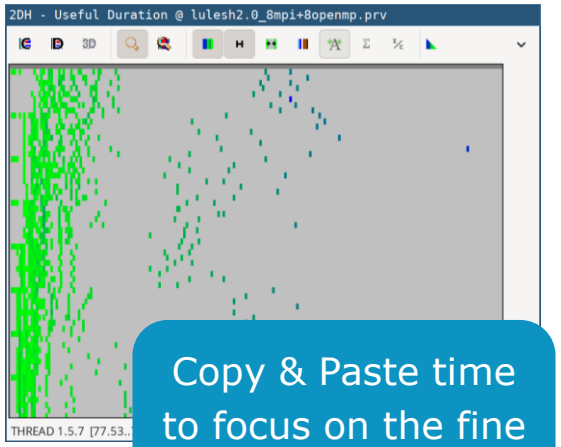




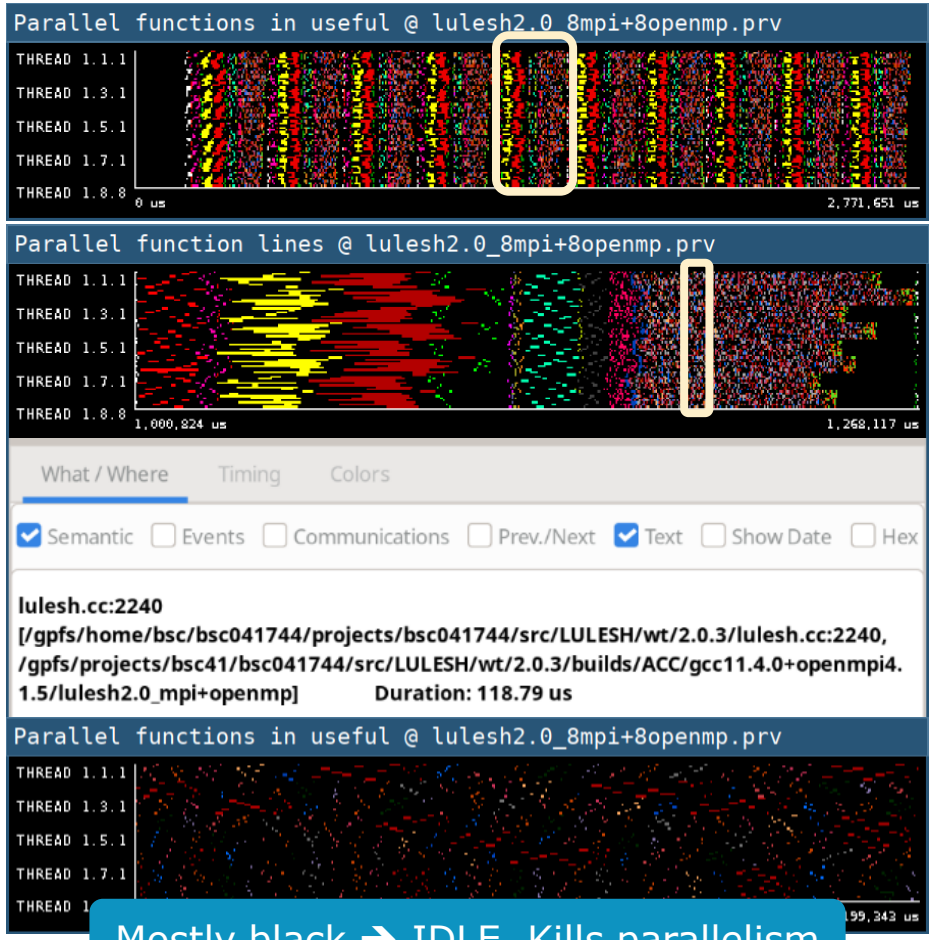
- Very low ParEff, only 17%
- Thread computing the most, only 21% of total time

Let's take a look at the parallelization





Copy & Paste time to focus on the fine grained region

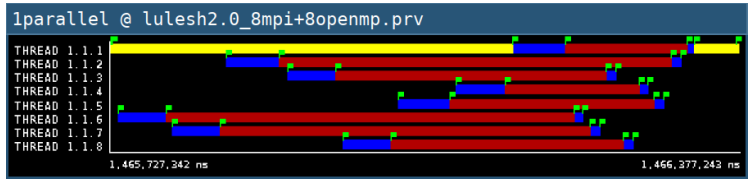


Mostly black ➔ IDLE. Kills parallelism

- Two "larger" parallels, yellow and red ~20ms
- Pink and green ~2ms
- Very fine-grained region (<80µs) taking more than 40% of the total iteration time
- *Parallel function lines* timeline shows us where to look in the code

Duration of computations in parallel regions

1 parallel



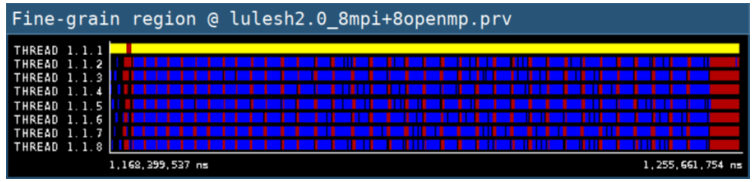
2D 1parallel @ lulesh2.0_8mpi+8openmp.prv

	Running	Synchronization	Scheduling and Fork/Join
Num. Cells	8	8	1
Total	490,100 ns	2,201,738 ns	460,157.00 ns
Average	61,262.50 ns	275,217.25 ns	460,157.00 ns
Maximum	65,349 ns	421,409 ns	460,157.00 ns
Minimum	58,675 ns	125,793 ns	460,157.00 ns
StDev	2,302.82 ns	109,560.04 ns	0 ns
Avg/Max	0.94	0.65	1

Immediate Receive

- ~60µs computing
- ~700µs overhead

Fine-grain region

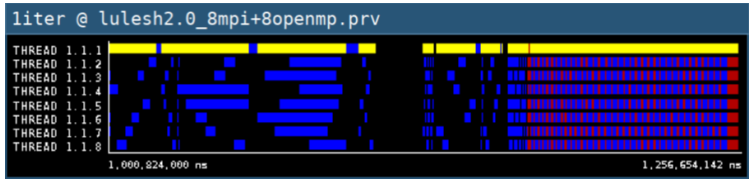


2D Fine-grain region @ lulesh2.0_8mpi+8openmp.prv

	Running	Synchronization	Scheduling and Fork/Join
Num. Cells	8	8	1
Total	40,941,610.52 ns	74,577,120 ns	75,883,051.52 ns
Average	5,117,701.31 ns	9,322,140 ns	75,883,051.52 ns
Maximum	5,209,849 ns	13,131,857 ns	75,883,051.52 ns
Minimum	4,987,866 ns	5,377,437 ns	75,883,051.52 ns
StDev	68,523.19 ns	1,996,359.56 ns	0 ns
Avg/Max	0.98	0.71	1

- ~5ms p.p. computing
- ~85ms p.p. overhead

1 full iteration



2D 1iter @ lulesh2.0_8mpi+8openmp.prv

	Running	Synchronization	Scheduling and Fork/Join
Num. Cells	8	8	1
Total	344,213,739 ns	74,577,120 ns	202,378,455 ns
Average	43,026,717.38 ns	9,322,140 ns	202,378,455 ns
Maximum	61,498,315 ns	13,131,857 ns	202,378,455 ns
Minimum	22,110,688 ns	5,377,437 ns	202,378,455 ns
StDev	12,222,318.30 ns	1,996,359.56 ns	0 ns
Avg/Max	0.70	0.71	1

Synchronization

- ~43ms p.p. computing
- ~210ms p.p. overhead

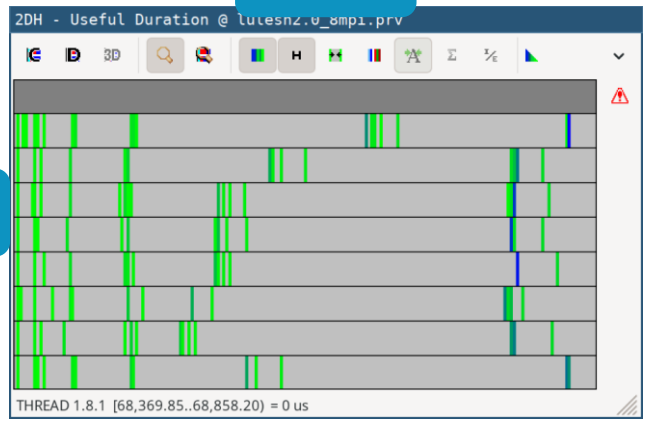
Most of the iteration time is spent in OpenMP overhead

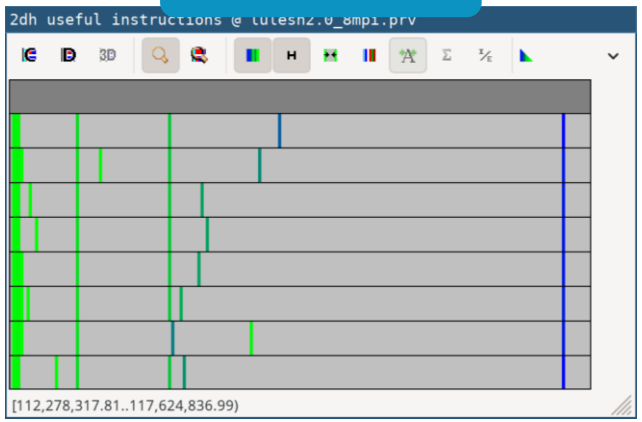
Computation comparison vs pure MPI with 8 ranks

Useful

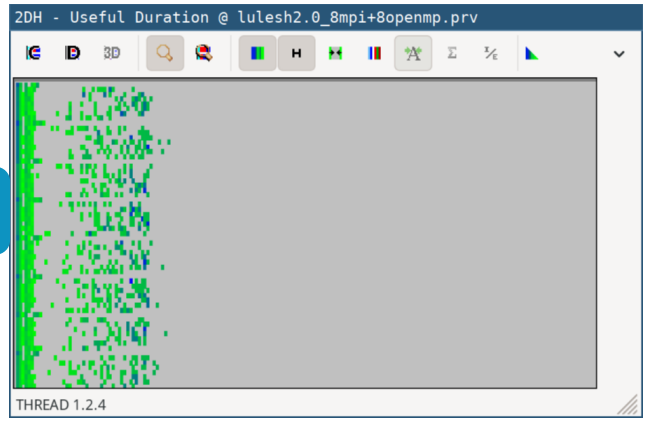
#instructions

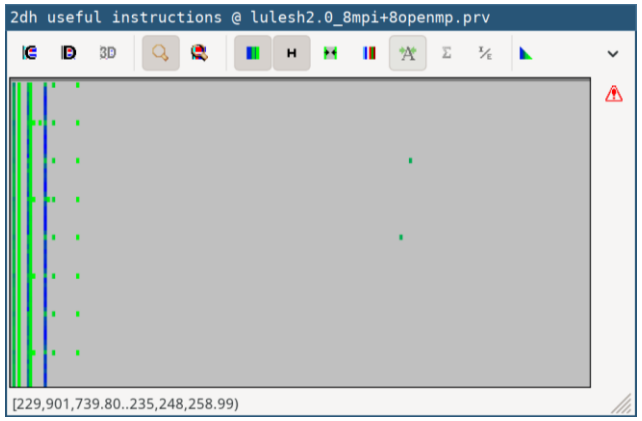
MPI





Hybrid





- LULESH does weak scaling with MPI but strong with OpenMP.
- We had 80ms computations
- We tried to split-up those into smaller chunks ➔ orchestration overhead dominates runtime

Getting a MPI+CUDA trace with Extrae

Step 1: Adapt the scripts to instrument MPI + CUDA binaries

- Changes in jobscript, launcher and configuration file (`extrae/MPI+CUDA/{trace.sh,extrae.xml}`)

job4p+4d.slurm

```
!/usr/bin/env bash

#SBATCH --job-name=cuda-cloverleaf_4p+4d
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=20
#SBATCH --gres=gpu:4
#SBATCH --time=00:10:00
#SBATCH --qos=acc_debug
#SBATCH --exclusive
#SBATCH --constraint-perfparanoid

ml gcc/11.4.0
ml openmpi/4.1.5-gcc
ml cuda/12/2

export TRACE_NAME=cuda-cloverleaf_4p+4d.prv

srun ./trace.sh ./cloverleaf-mpi
```

4 tasks, one for each CUDA device and 20 cores per task

trace.sh

```
#!/usr/bin/env bash

module load extrae

export EXTRAE_CONFIG_FILE=./extrae.xml
# For MPI+CUDA apps
- export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so
+ export LD_PRELOAD=${EXTRAE_HOME}/lib/libcudampitrace.so

## Run the desired program
$* --device $PMI_RANK
```

Tracing library for MPI + CUDA

Adds multi-device support

extrae.xml

```
- <cuda enabled="no" />
+ <cuda enabled="yes" />
```

Enables CUDA instrumentation

Step 3: Run it!

- Submit your job as usual

```
alogin1$ sbatch -A nct_362 --reservation=POP3Tools job4p+4d.slurm
```

- Once finished (check with "`squeue`") you will have the trace (3 files):

```
alogin1$ ls -l
  ...
  cloverleaf-mpi_4p+4d.pcf
  cloverleaf-mpi_4p+4d.prv
  cloverleaf-mpi_4p+4d.row
```

- Any trouble? There's a trace already generated under the "`traces`" folder

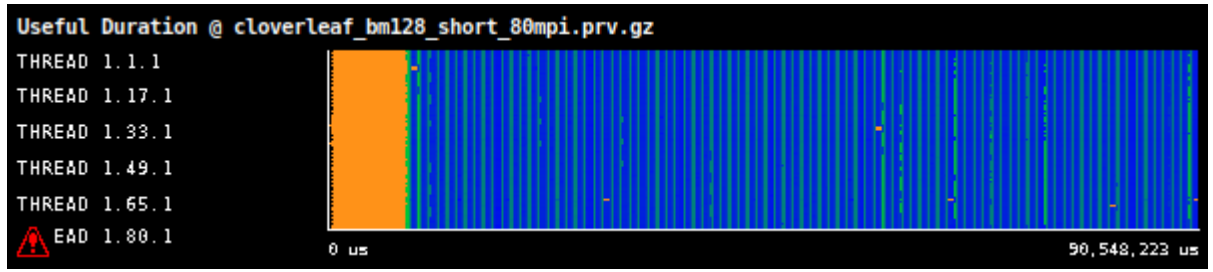
- Now let's look into it!

Getting a hybrid MPI+CUDA trace with Extrae

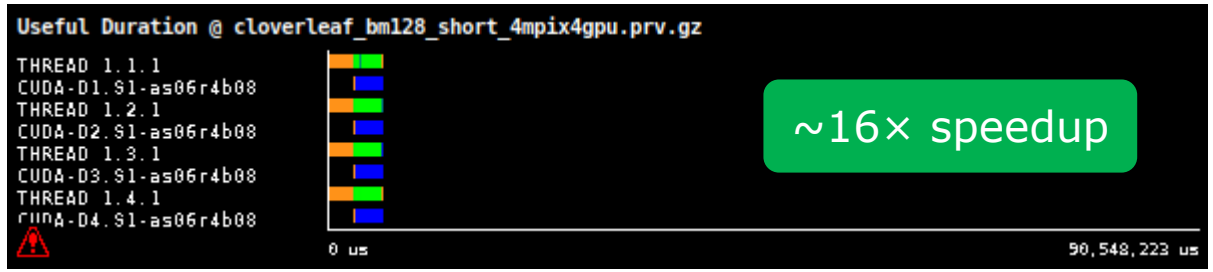
CloverLeaf: which configuration works best?

- Compressible Euler equations on a Cartesian grid
- Parallelized with MPI, CUDA (cases analyzed), HIP, OpenMP, Kokkos, SYCL, OpenACC, TBB…
- Input: clover_bm128_short.in
- Resources
  - 80 MPI ranks (fully utilize all node CPUs)
  - 4 MPI ranks x 1 GPU each (fully utilize all node GPUs)

MPI
~90 s

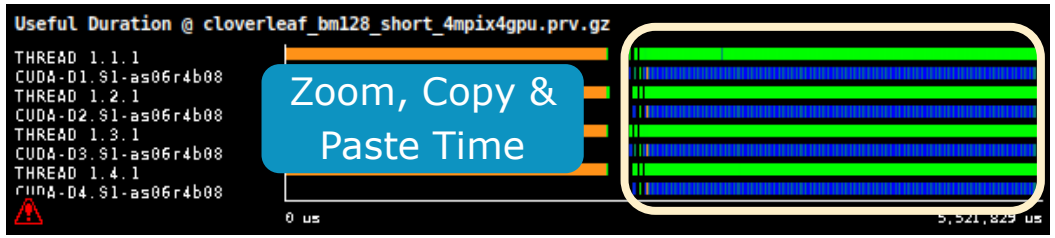


MPI+CUDA
~5.5 s



~16× speedup

Computing regions



- Hints → Efficiency
  - Untick "Show totals only"

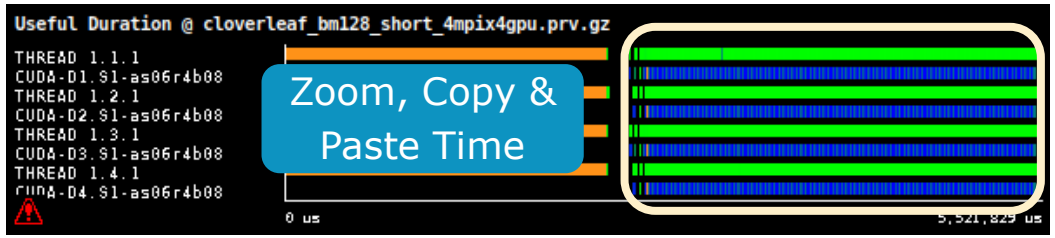
- Host processes don't do "useful work"

- GPUs are active computing kernels almost 90% of the time

Efficiency

	Running
THREAD 1.1.1	0.56 %
CUDA-D1.S1-as06r4b08	88.40 %
THREAD 1.2.1	0.36 %
CUDA-D2.S1-as06r4b08	88.43 %
THREAD 1.3.1	0.40 %
CUDA-D3.S1-as06r4b08	88.45 %
THREAD 1.4.1	0.48 %
CUDA-D4.S1-as06r4b08	88.43 %
Num. Cells	8
Total	355.52 %
Average	44.44 %
Maximum	88.45 %
Minimum	0.36 %
StDev	43.99 %
Avg/Max	0.50

GPU orchestration overhead



- Hints → CUDA → CUDA profile
  - Right click → Select objects → Filter out GPU devices
- Host processes are mainly:
  - Waiting for the GPU (~81%)
  - Data transfers (~7%)
  - Launching kernels (~18K, ~6%)
- ~95% of host activity is devoted to CUDA orchestration

% Window Time

CUDA profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	cudaDeviceSynchronize	cudaMemcpy	cudaLaunch	cudaFree	cudaMalloc
THREAD 1.1.1	81.00 %	6.58 %	5.49 %	0.63 %	0.09 %
THREAD 1.2.1	80.95 %	6.58 %	5.86 %	0.71 %	0.08 %
THREAD 1.3.1	80.94 %	6.59 %	5.85 %	0.68 %	0.09 %
THREAD 1.4.1	81.09 %	6.58 %	5.46 %	0.61 %	0.08 %
Num. Cells	4	4	4	4	4
Total	323.98 %	26.34 %	22.65 %	2.62 %	0.35 %
Average	80.99 %	6.59 %	5.66 %	0.66 %	0.09 %

% Avg. Burst Time

CUDA profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

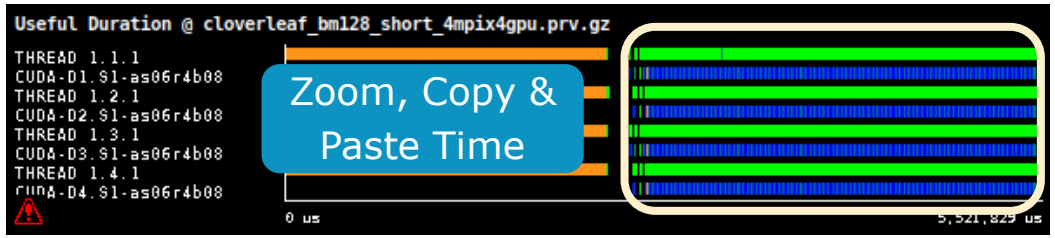
	cudaDeviceSynchronize	cudaMemcpy	cudaFree	cudaMalloc	cudaLaunch
THREAD 1.1.1	2,344,918.23 ns	1,356,401.23 ns	105,852.68 ns	17,717.56 ns	9,347.31 ns
THREAD 1.2.1	2,343,334.67 ns	1,356,034.59 ns	118,651.95 ns	16,591.32 ns	9,975.83 ns
THREAD 1.3.1	2,343,229.78 ns	1,357,053.26 ns	114,238.96 ns	17,027.66 ns	9,969.36 ns
THREAD 1.4.1	2,347,507.78 ns	1,356,406.69 ns	102,113.49 ns	16,175.75 ns	9,294.64 ns
Num. Cells	4	4	4	4	4
Total	9,378,990.46 ns	5,425,895.77 ns	440,857.08 ns	67,512.29 ns	38,587.14 ns
Average	2,344,747.62 ns	1,356,473.94 ns	110,214.27 ns	16,878.07 ns	9,646.78 ns

Bursts

CUDA profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	cudaLaunch	cudaDeviceSynchronize	cudaFree	cudaMalloc	cudaMemcpy
THREAD 1.1.1	17,777	1,046	180	155	147
THREAD 1.2.1	17,777	1,046	180	155	147
THREAD 1.3.1	17,777	1,046	180	155	147
THREAD 1.4.1	17,777	1,046	180	155	147

MPI overhead



- Hints → MPI → MPI profile
  - Right click → Select objects → Filter out GPU devices
- MPI overhead <5%, mostly in MPI_Waitall
- Small imbalance in MPI_Barrier, but related to initialization phase

% Window Time

MPI call profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	MPI_Waitall	MPI_Isend	MPI_Barrier	MPI_Allreduce	MPI_Reduce	MPI_Irecv	MPI_Finalize
THREAD 1.1.1	2.15 %	0.87 %	0.96 %	0.06 %	0.16 %	0.08 %	0.00 %
THREAD 1.2.1	2.04 %	0.89 %	0.02 %	0.10 %	0.01 %	0.08 %	0.00 %
THREAD 1.3.1	1.91 %	0.86 %	0.85 %	0.10 %	0.16 %	0.08 %	0.00 %
THREAD 1.4.1	2.10 %	1.05 %	0.04 %	0.13 %	0.01 %	0.07 %	0.00 %
Num. Cells	4	4	4	4	4	4	4
Total	8.19 %	3.67 %	1.86 %	0.39 %	0.35 %	0.30 %	0.00 %
Average	2.05 %	0.92 %	0.47 %	0.10 %	0.09 %	0.08 %	0.00 %

% Avg. Burst Time

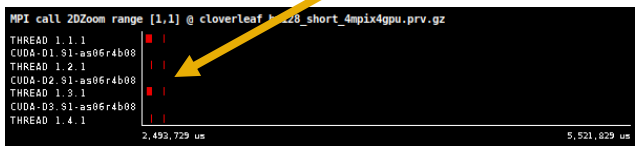
MPI call profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	MPI_Barrier	MPI_Waitall	MPI_Reduce	MPI_Isend	MPI_Finalize	MPI_Allreduce	MPI_Irecv
THREAD 1.1.1	7,239.95 us	62.10 us	98.15 us	25.26 us	14.13 us	7.54 us	2.20 us
THREAD 1.2.1	147.55 us	58.94 us	7.77 us	25.82 us	16.33 us	11.79 us	2.18 us
THREAD 1.3.1	6,420.78 us	55.42 us	96.12 us	24.97 us	15.04 us	11.70 us	2.25 us
THREAD 1.4.1	300.07 us	60.71 us	8.14 us	30.28 us	13.60 us	14.55 us	2.15 us
Num. Cells	4	4	4	4	4	4	4
Total	14,108.35 us	237.17 us	210.17 us	106.32 us	59.10 us	45.58 us	8.78 us
Average	3,527.09 us	59.29 us	52.54 us	26.58 us	14.78 us	11.39 us	2.20 us

Bursts

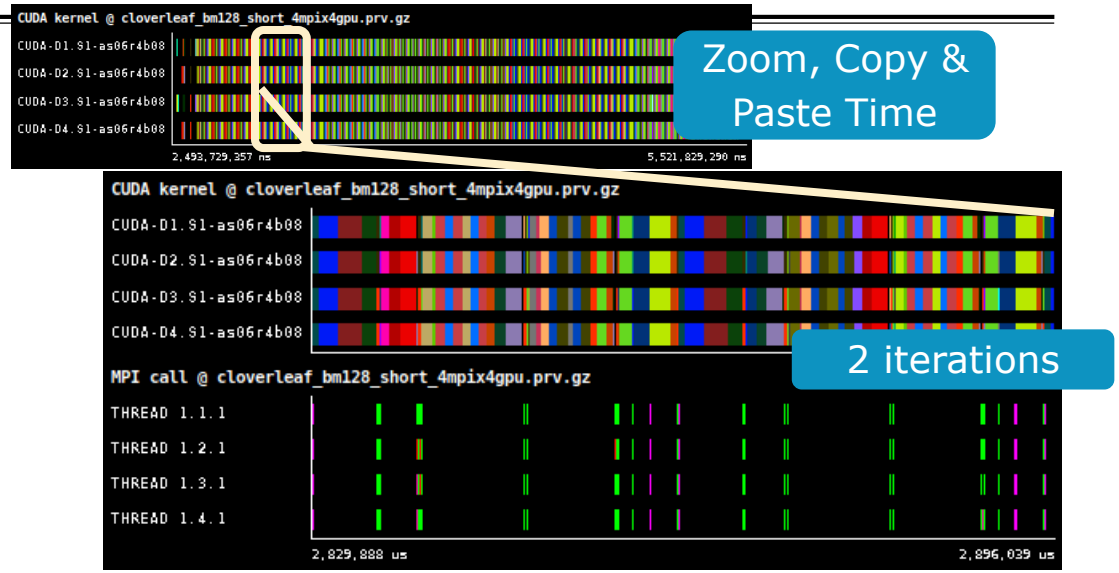
MPI call profile @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	MPI_Waitall	MPI_Irecv	MPI_Isend	MPI_Allreduce	MPI_Reduce	MPI_Barrier	MPI_Finalize
THREAD 1.1.1	1,046	1,046	1,046	261	50	4	1
THREAD 1.2.1	1,046	1,046	1,046	261	50	4	1
THREAD 1.3.1	1,046	1,046	1,046	261	50	4	1
THREAD 1.4.1	1,046	1,046	1,046	261	50	4	1



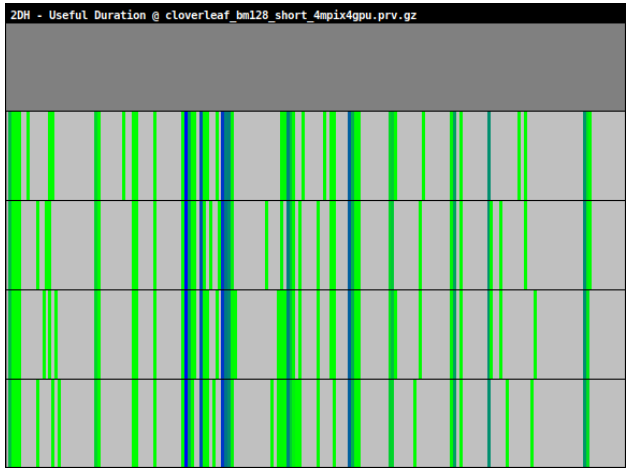
Kernel execution

- Hints → CUDA → CUDA kernel
  - Right click → Select objects → Filter out host threads
- Iterative structure delimited by `MPI_Allreduce` phase
- Main computing kernel is `par_ranged2d_kernel`
- Kernel duration is well balanced between GPUs
- Granularity is very small ranging from 10 us to 2 ms



Kernel execution @ cloverleaf_bm128_short_4mpix4gpu.prv.gz

	CUDA-D1.S1-as06r4b08	CUDA-D2.S1-as06r4b08	CUDA-D3.S1-as06r4b08	CUDA-D4.S1-as06r4b08
void clover::par_ranged2d_kernel	6.35 %	6.34 %	6.35 %	6.35 %
void clover::par_ranged2d_kernel	5.29 %	5.29 %	5.29 %	5.29 %
void clover::par_ranged2d_kernel	5.18 %	5.16 %	5.17 %	5.17 %
void clover::par_ranged2d_kernel	4.73 %	4.73 %	4.73 %	4.73 %
void clover::par_ranged2d_kernel	4.71 %	4.71 %	4.71 %	4.71 %
void clover::par_ranged2d_kernel	4.26 %	4.26 %	4.27 %	4.26 %
void clover::par_ranged2d_kernel	4.26 %	4.26 %	4.26 %	4.26 %
void clover::par_reduce_kernel	4.22 %	4.22 %	4.22 %	4.23 %
void clover::par_ranged2d_kernel	3.95 %	3.94 %	3.94 %	3.94 %
void clover::par_ranged2d_kernel	3.78 %	3.78 %	3.79 %	3.79 %
void clover::par_ranged2d_kernel	3.77 %	3.76 %	3.76 %	3.76 %
void clover::par_ranged2d_kernel	3.76 %	3.76 %	3.76 %	3.76 %
void clover::par_ranged2d_kernel	3.12 %	3.11 %	3.11 %	3.12 %
void clover::par_ranged2d_kernel	3.07 %	3.08 %	3.07 %	3.06 %
void clover::par_ranged2d_kernel	3.07 %	3.08 %	3.07 %	3.07 %
void clover::par_ranged2d_kernel	2.94 %	2.94 %	2.94 %	2.94 %
void clover::par_ranged2d_kernel	2.44 %	2.42 %	2.44 %	2.44 %
void clover::par_ranged2d_kernel	2.41 %	2.41 %	2.42 %	2.41 %

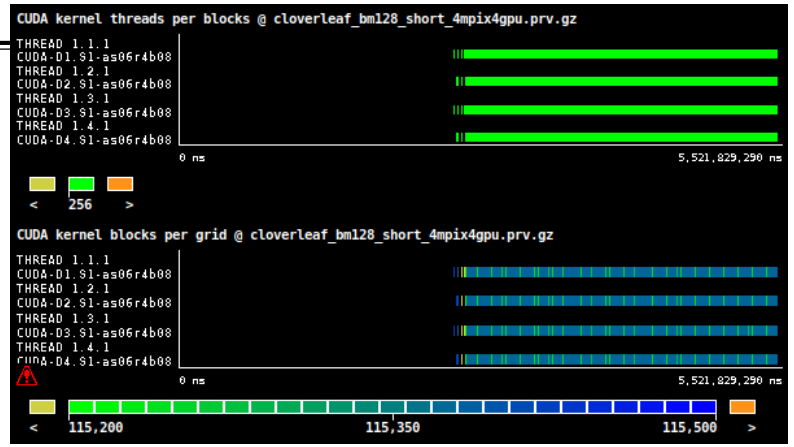


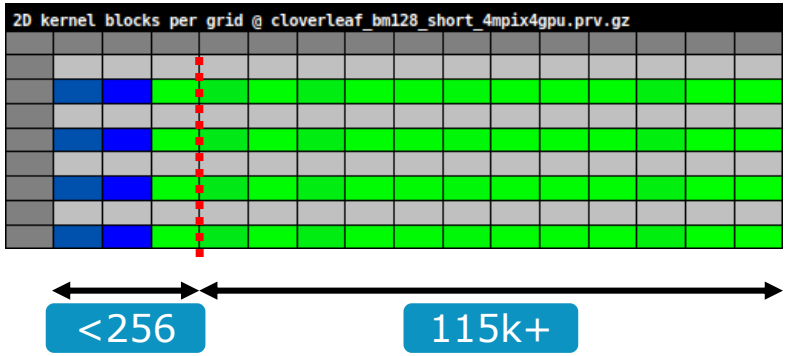
Kernel execution configuration

- Threads per block constant at 256
- ~15K kernels with small grid (16/31/256 blocks)
- ~3K kernels with large grid (~115K blocks)
  - These are much more likely to be throughput-limited (memory bandwidth, math, occupancy…)
- Execution time is dominated by the smaller count of large grid kernels (~81%) vs. large count of small grid kernels (~7%)

2D kernel threads per block

	value 256
THREAD 1.1.1	-
CUDA-D1.S1-as06r4b08	17,777
THREAD 1.2.1	-
CUDA-D2.S1-as06r4b08	17,777
THREAD 1.3.1	-
CUDA-D3.S1-as06r4b08	17,777
THREAD 1.4.1	-
CUDA-D4.S1-as06r4b08	17,777
Num. Cells	4
Total	71,108
Average	17,777
Maximum	17,777
Minimum	17,777
StDev	0
Avg/Max	1





Time in large grid kernels

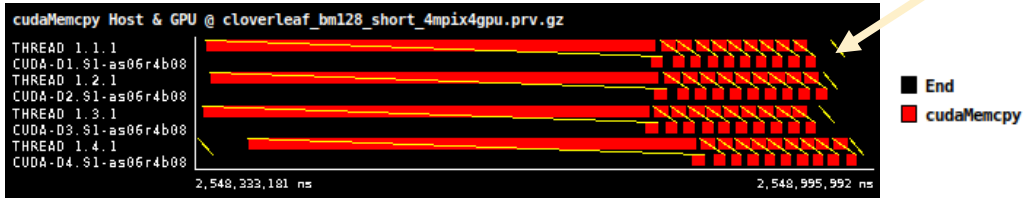
	[115,200.00..115,437.30]
CUDA-D1.S1-as06r4b08	81.32 %
CUDA-D2.S1-as06r4b08	81.32 %
CUDA-D3.S1-as06r4b08	81.31 %
CUDA-D4.S1-as06r4b08	81.31 %
Num. Cells	4
Total	325.26 %
Average	81.32 %
Maximum	81.32 %
Minimum	81.31 %
StDev	0.00 %
Avg/Max	1.00

Time in small grid kernels

	[16.00..268.00]
CUDA-D1.S1-as06r4b08	7.39 %
CUDA-D2.S1-as06r4b08	7.42 %
CUDA-D3.S1-as06r4b08	7.45 %
CUDA-D4.S1-as06r4b08	7.42 %
Num. Cells	4
Total	29.68 %
Average	7.42 %
Maximum	7.45 %
Minimum	7.39 %
StDev	0.02 %
Avg/Max	1.00

Data transfers

- Sequence of host to device data transfers before the main iterative loop



- 1 data transfer per iteration from device to host of 2 Kb
  - Right before the MPI reduction phase

BSC Tools
Hands-On

Lau Mercadal, Sandra Méndez, Germán Llort
(tools@bsc.es)
Barcelona Supercomputing Center