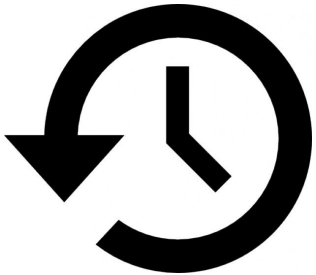
Predicting the performance of QuantumESPRESSO

Pietro Bonfà, Fabio Affinito, Carlo Cavazzoni
CINECA

MaX International Conference 2018, Trieste 29-31 January 2018

Hardware software co-design

Intel: [...] the new architecture we are designing has 1.4 GHz cores, but new vector instructions and more than 64 cores in a single socket and many GBs of High Bandwidth Memory (HBM).



2010

1. Can QE exploit this kind of architecture?
2. How many GBs of HBM are appropriate for QE?

Performance modeling

Analytical Performance Modeling is a method of Software performance testing generally used to evaluate design options and system sizing based on actual or anticipated system behaviour.

In the context of co-design it may be used for:

- Making predictions on the efficacy of hardware.
- Monitor hotspots and bottleneck as the hardware is designed.
- Avoid longer and more expensive performance testing.

Performance modeling

Analytical Performance Modeling is a method of Software performance testing generally used to evaluate design options and system sizing based on actual or anticipated system behaviour.

In the context of code development it may be used for:

- Understand where there is room for improvement.
- Monitor hotspots and bottleneck as the hardware evolves.
- Avoid longer and more expensive performance testing.

Performance modeling

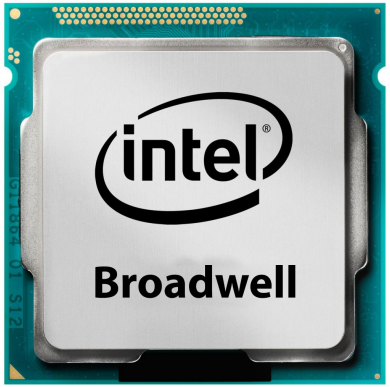
Analytical Performance Modeling is a method of Software performance testing generally used to evaluate design options and system sizing based on actual or anticipated system behaviour.

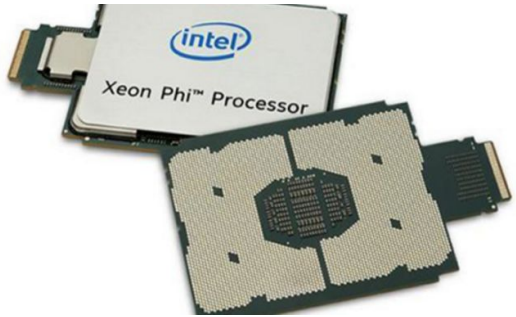
In the context of code usability it may be used for:

- Provide indications in the job timings in advances.
- Auto tuning of parallel parameters.
- Avoid performance testing for projects' submission.

Task details

Create a performance model to obtain the relevant information about **pw.x** to be used in hardware participatory design, targeting the standard total energy task and **a modern HPC node**, i.e. tens of cores, tens of GBs of RAM.





Contributions to the total execution time

The total execution time for an application can be approximated as the sum of a few contributions:

$$T(f,BW,NB) = MPI(BW, NB) + IO(BW,NB,IOB) + SERIAL(f,BW,NB)$$

Where *NB* is the network bandwidth, *BW* is the memory bandwidth per core, *f* is the CPU frequency and the SERIAL part is the code executed by each of the MPI processes. All these term have an **implicit dependence** on the **input** parameters.

Performance projection

Approach 1:

$$T(f,BW,NB)/T_{ref} = \alpha_{MPI}(NB_{ref}/NB) + \alpha_{CPU}(f_{ref}/f) + \alpha_{BW}(BW_{ref}/BW(f)) + \ldots$$

PRO: change few parameters to extract values for α_x.
CONS: limited predictive power (practically probably few generations). Need to repeat the analysis after every (major) code change.

Approach 2:

$$T(f,BW,NB) = \sum T_{kernel}(f,BW,NB,IOB) + T_{other} \quad \text{with} \quad T_{kernel}(f,BW,NB) \simeq T_c(f) + T_{mem}(BW) + T_{MPI}(NB) + T_{I/O}$$

PRO: detailed absolute time predictions.
CONS: requires extensive analysis of the code execution flows.

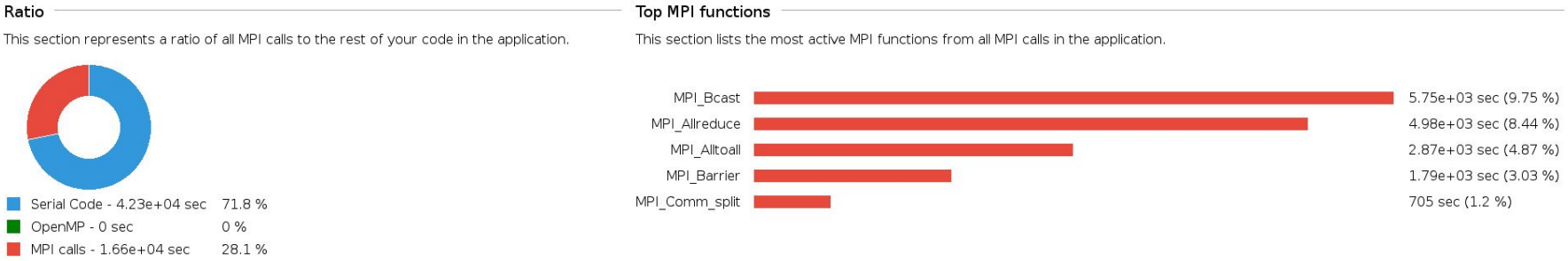
Step one: profiling

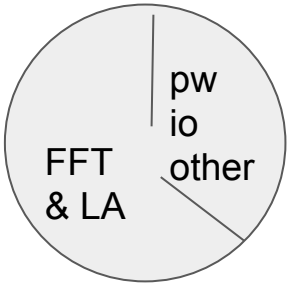
Classify code sections: compute, memory, communication, i/o bound

Identify computationally intensive parts

Profiling of pw.x

Time in medium to large sized simulation most of the time is spent in MPI and LA calls.





Time mostly on three kernels:

- GEMM
- Diagonalization
- FFT

I/O is negligible, MPI is mainly Alltoall (FFT) and Bcast/Allreduce (Diagonalization)

Profiling of yambo

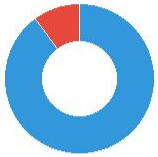
Summary: yambo.stf

Total time: **2.86e+05** sec. Resources: **64** processes, **1** node.

Continue >

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.



Serial Code - 2.57e+05 sec	89.8 %
OpenMP - 0 sec	0 %
MPI calls - 2.9e+04 sec	10.1 %

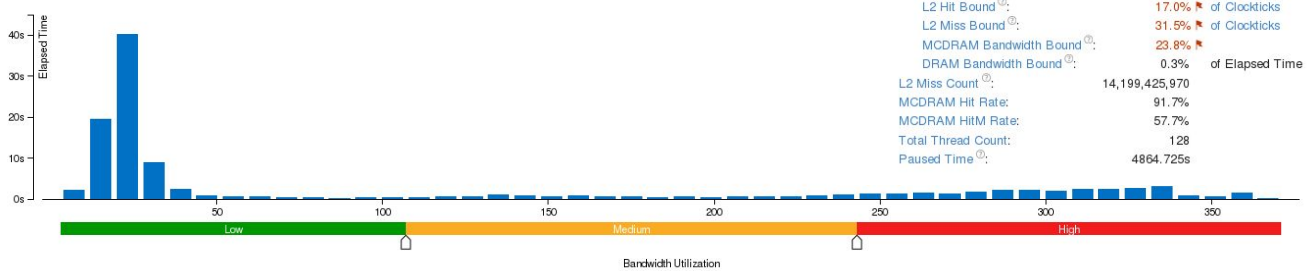
Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

Bandwidth Domain: MCDRAM, GB/sec

Bandwidth Utilization Histogram

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropr maximum achievable DRAM and QPI bandwidth.



Elapsed Time: 4985.788s

CPU Time:	7413.935s	
Memory Bound:		
L2 Hit Rate:	87.9%	
L2 Hit Bound:	17.0%	of Clockticks
L2 Miss Bound:	31.5%	of Clockticks
MCDRAM Bandwidth Bound:	23.8%	
DRAM Bandwidth Bound:	0.3%	of Elapsed Time
L2 Miss Count:	14,199,425,970	
MCDRAM Hit Rate:	91.7%	
MCDRAM HitM Rate:	57.7%	
Total Thread Count:	128	
Paused Time:	4864.725s	

The pw.x model components

FFTXlib kernel: FFT kernel + MPI Alltoall + memory access

MM kernel: used during iterative diagonalization

Diagonalization kernel: serial LAPACK function: zhegv, zhegvx

Unbalance: kpoints distribution

Step two: kernel's details

Count FLOP or data access as a function of input parameters.

Choose model parameters: cpu frequency, cache size, memory bandwidth per code, memory hierarchy, vectorization, software stack, openMP, ...

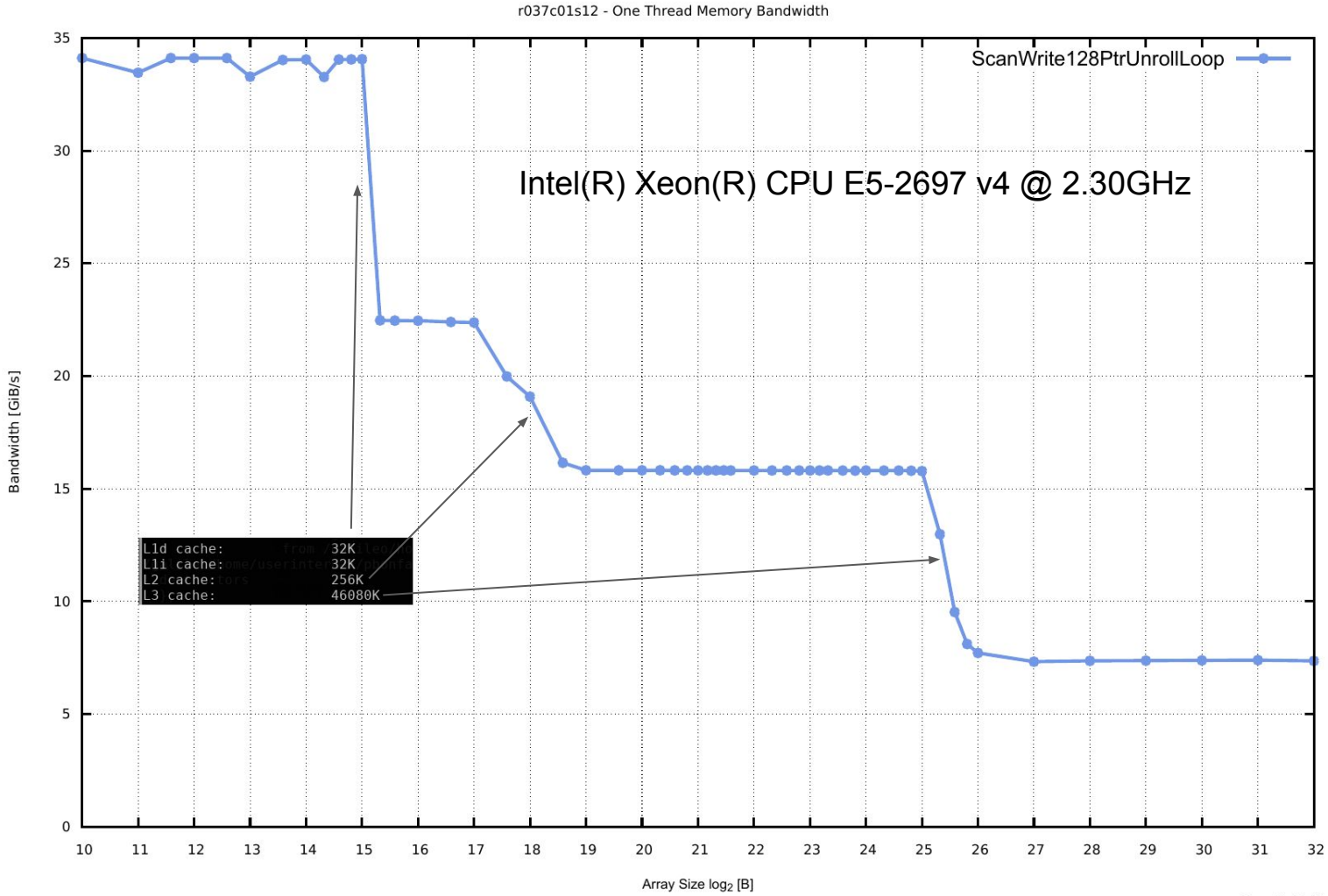
```
#**********************************************************************
#*             Generic formula coming from LAWN 41
#**********************************************************************

#
# Level 2 BLAS
#
FMULS_GEMV = lambda __m, __n : ((__m) * (__n) + 2. * (__m))
FADDS_GEMV = lambda __m, __n : ((__m) * (__n))

FMULS_SYMV = lambda __n : FMULS_GEMV( (__n), (__n) )
FADDS_SYMV = lambda __n : FADDS_GEMV( (__n), (__n) )
FMULS_HEMV = FMULS_SYMV
FADDS_HEMV = FADDS_SYMV

#
# Level 3 BLAS
#
FMULS_GEMM = lambda __m, __n, __k: ((__m) * (__n) * (__k))
FADDS_GEMM = lambda __m, __n, __k: ((__m) * (__n) * (__k))

FLOPS_ZGEMM = lambda __m, __n, __k: (6. * FMULS_GEMM((__m), (__n), ...
FLOPS_CGEMM = lambda __m, __n, __k: (6. * FMULS_GEMM((__m), (__n), ...
FLOPS_DGEMM = lambda __m, __n, __k: (        FMULS_GEMM((__m), (__n), .
FLOPS_SGEMM = lambda __m, __n, __k: (        FMULS_GEMM((__m), (__n), .
```

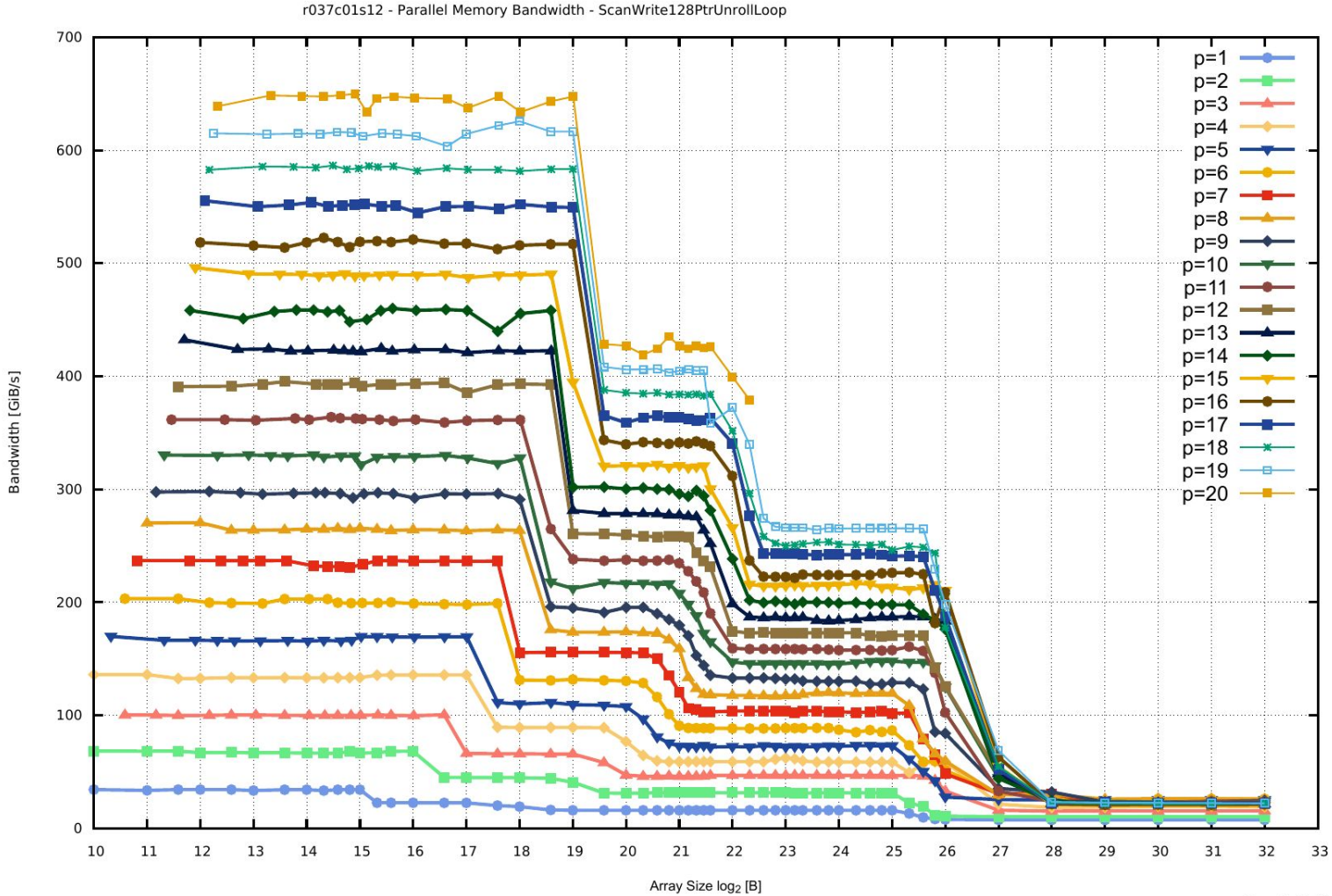
https://github.com/arporter/habakkuk

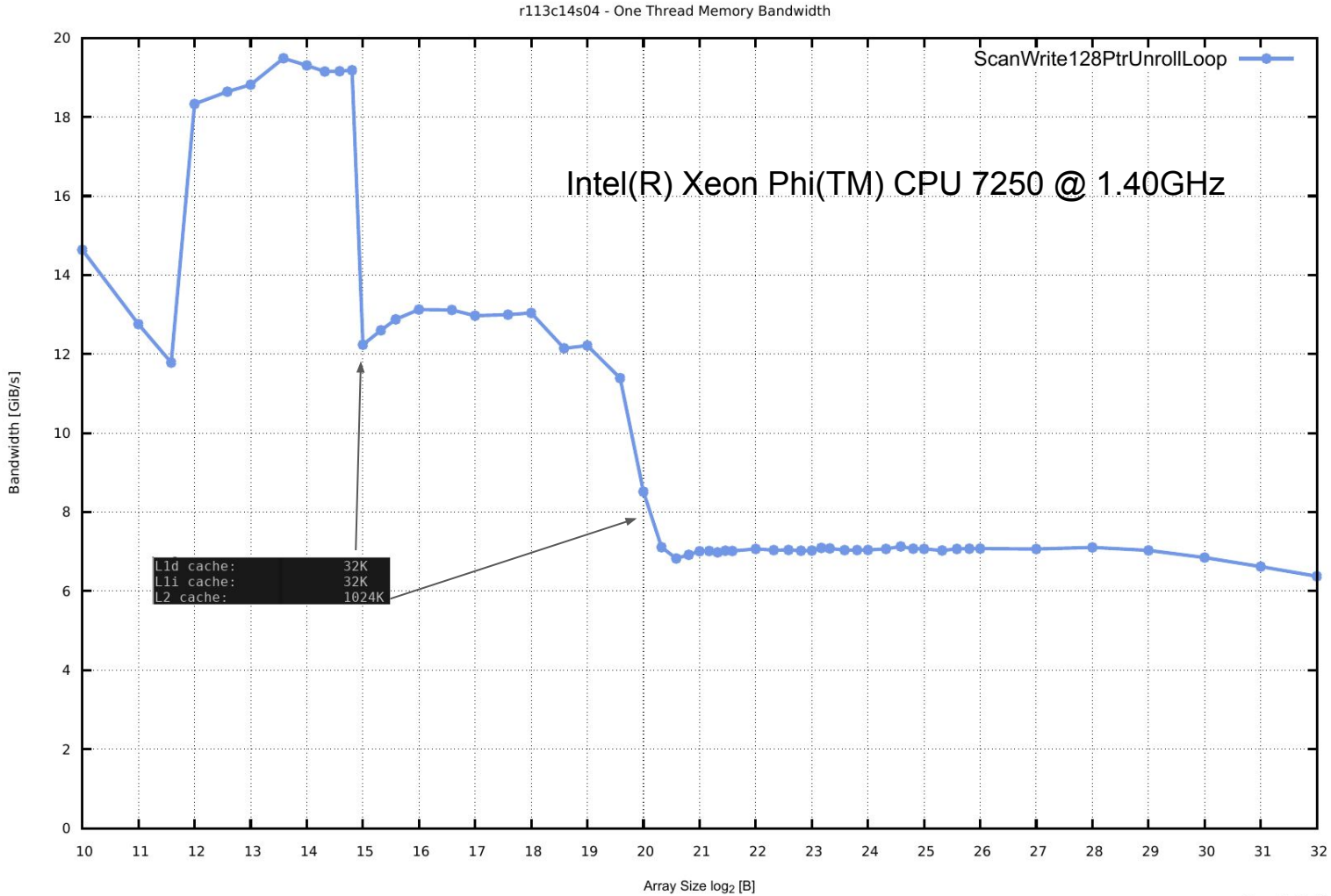
How to choose the relevant HW/SW parameters?

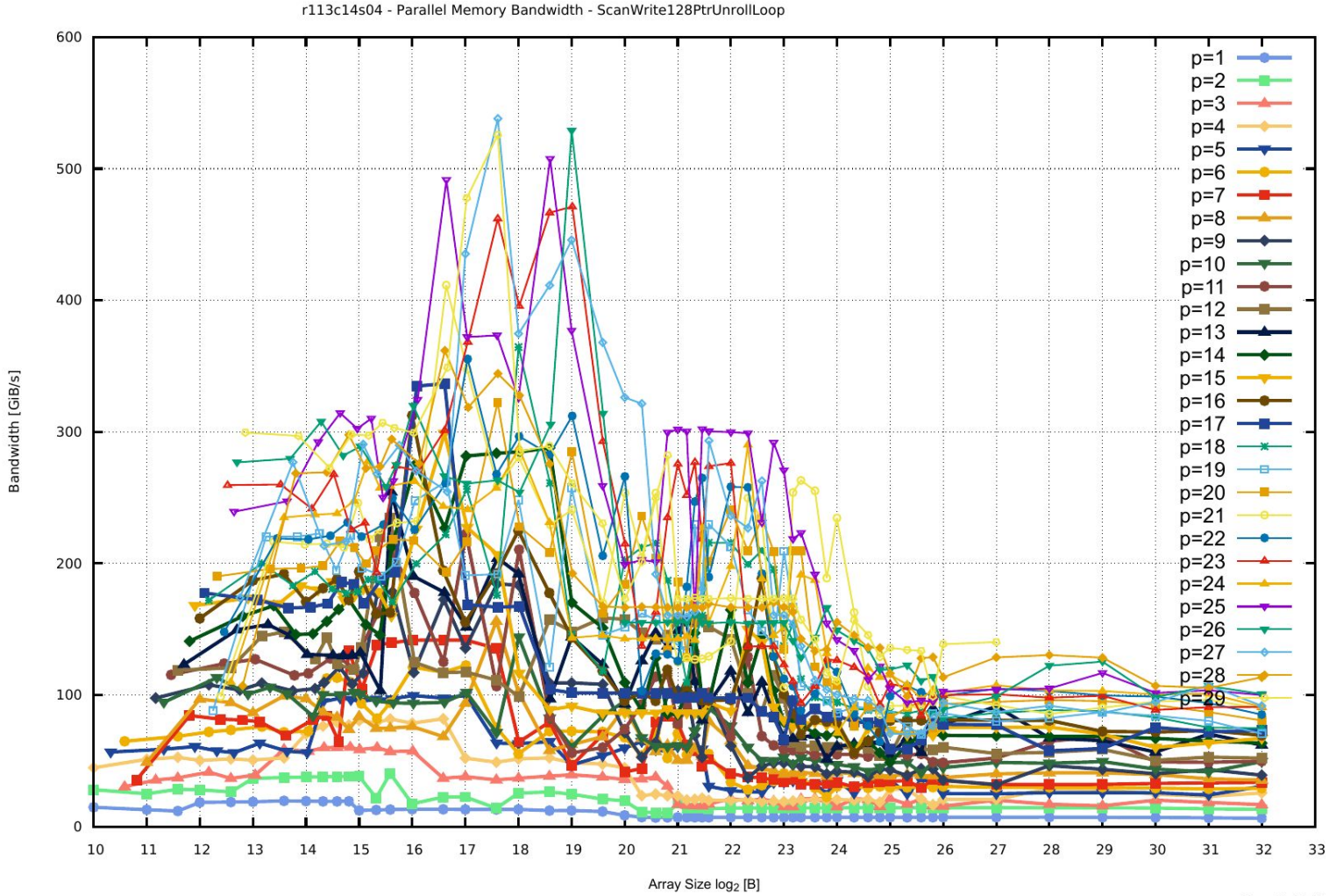
Possible parameters to consider: cpu frequency, cache size, memory bandwidth per code, memory hierarchy, vectorization, software stack, openMP, ...

What is relevant? What is correlated with what?

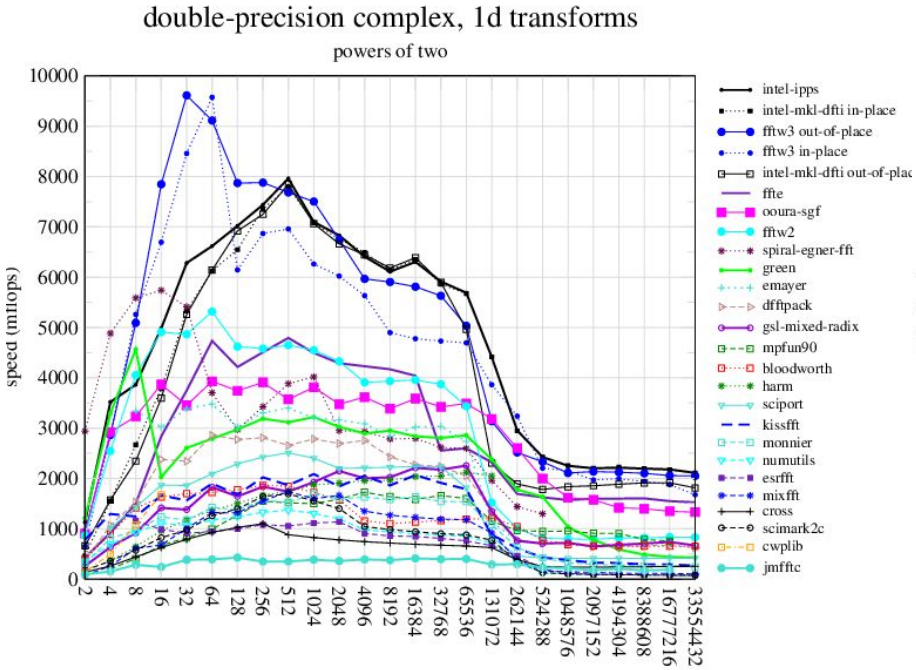

r037c01s12 - One Thread Memory Bandwidth
ScanWrite128PtrUnrollLoop
Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 46080K
Bandwidth [GiB/s]
Array Size log2 [B]
pmbw 0.6.2

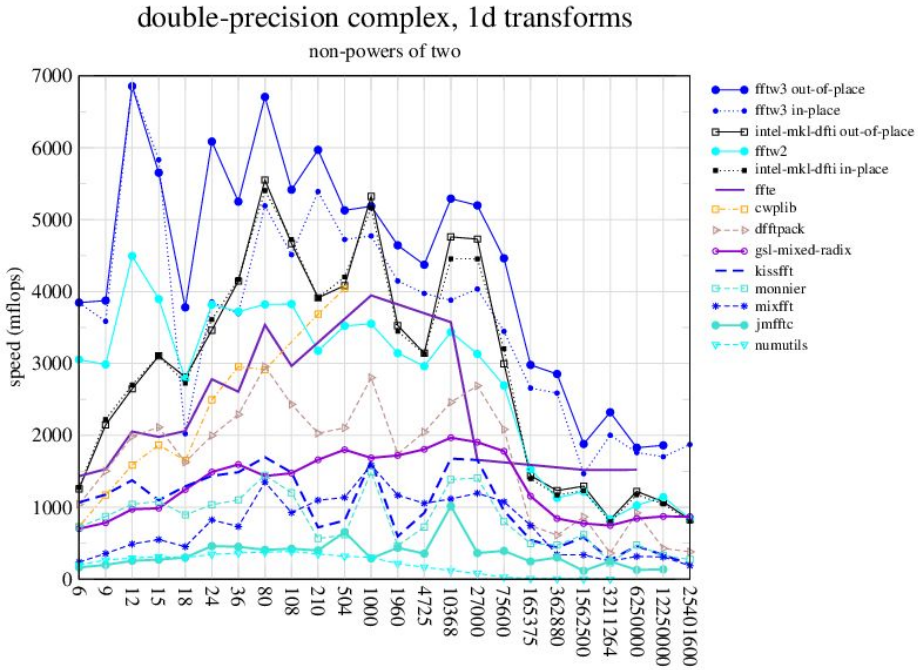

r037c01s12 - Parallel Memory Bandwidth - ScanWrite128PtrUnrollLoop
Bandwidth [GiB/s]
Array Size log2 [B]
p=1
p=2
p=3
p=4
p=5
p=6
p=7
p=8
p=9
p=10
p=11
p=12
p=13
p=14
p=15
p=16
p=17
p=18
p=19
p=20
pmbw 0.6.2


r113c14s04 - One Thread Memory Bandwidth
ScanWrite128PtrUnrollLoop
Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
Bandwidth [GiB/s]
Array Size log2 [B]
pmbw 0.6.2


r113c14s04 - Parallel Memory Bandwidth - ScanWrite128PtrUnrollLoop
Bandwidth [GiB/s]
Array Size log2 [B]
p=1
p=2
p=3
p=4
p=5
p=6
p=7
p=8
p=9
p=10
p=11
p=12
p=13
p=14
p=15
p=16
p=17
p=18
p=19
p=20
p=21
p=22
p=23
p=24
p=25
p=26
p=27
p=28
p=29
pmbw 0.6.2

Software side: FFT





Model input

1. **pw.x input** files and parallel execution details:

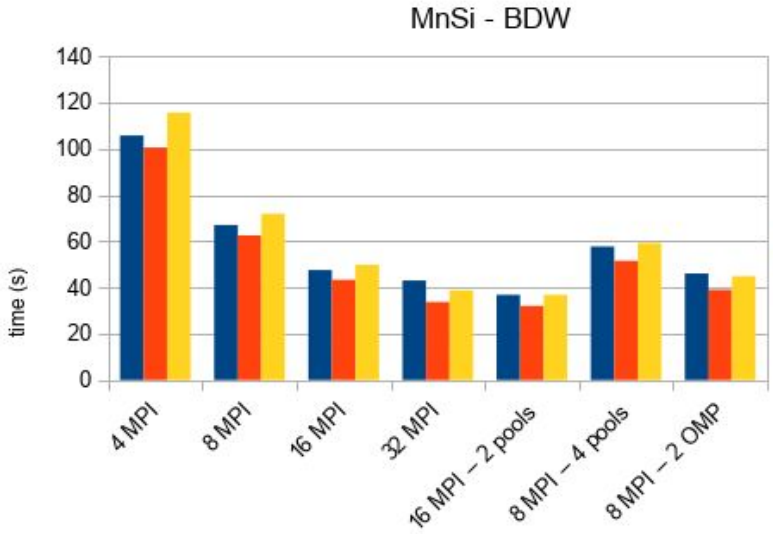
   Used to calculate the number of FLOPs of MM, FFT and diagonalization and memory accesses.

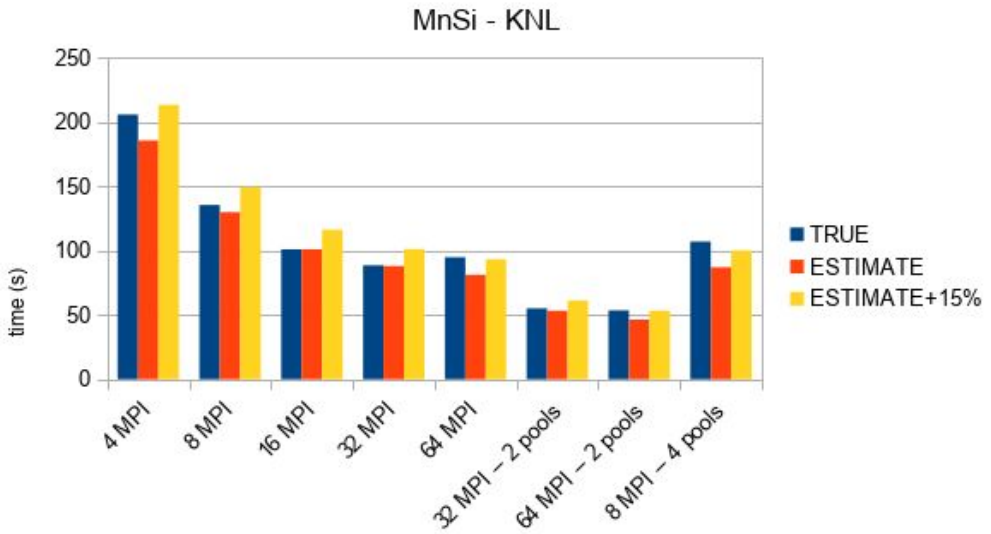
2. System parameters through **microbenchmarks**:

   FLOP/s: obtained with synthetic DGEMM and Diagonalization calls.
   FFT performance: obtained with mini FFT benchmark tool.
   Memory bandwidth: obtained with synthetic memory access.
   Network bandwidth: obtained with synthetic MPI alltoall communications.

Results

Absolute time estimate results.

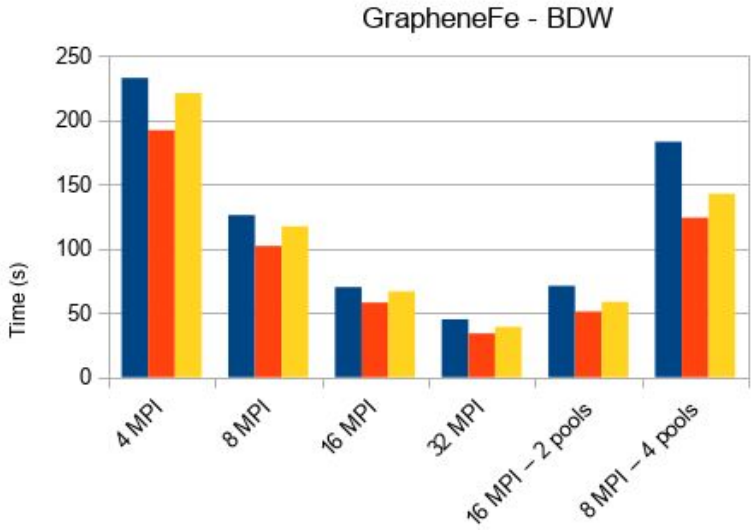


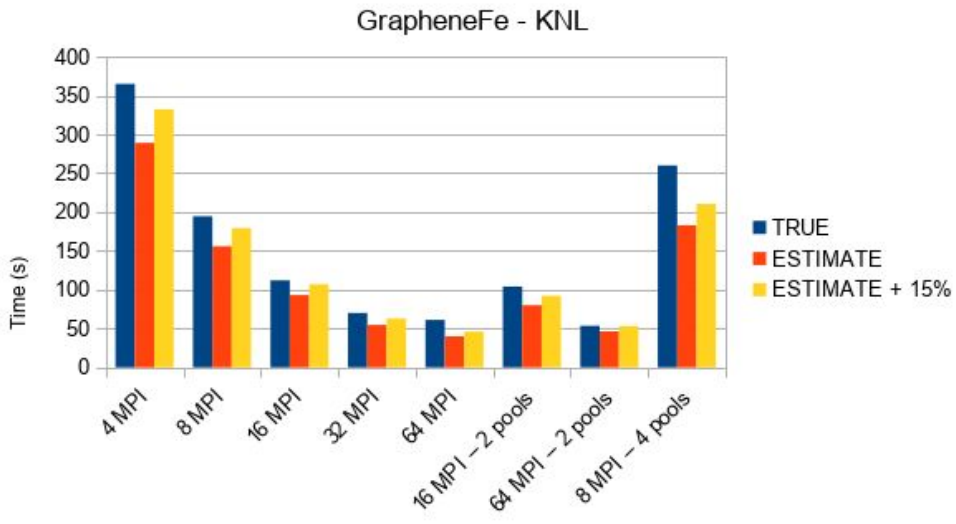


MnSi, bulk, 64 atoms, 14 k-points

Results

Absolute time estimate results.

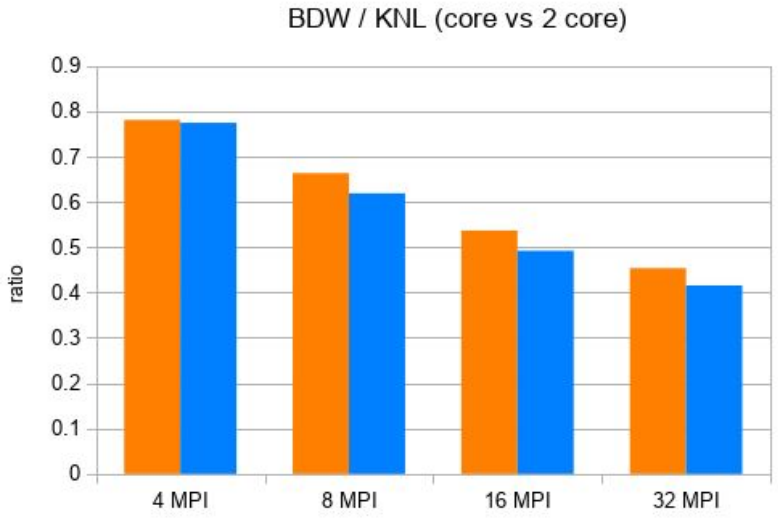




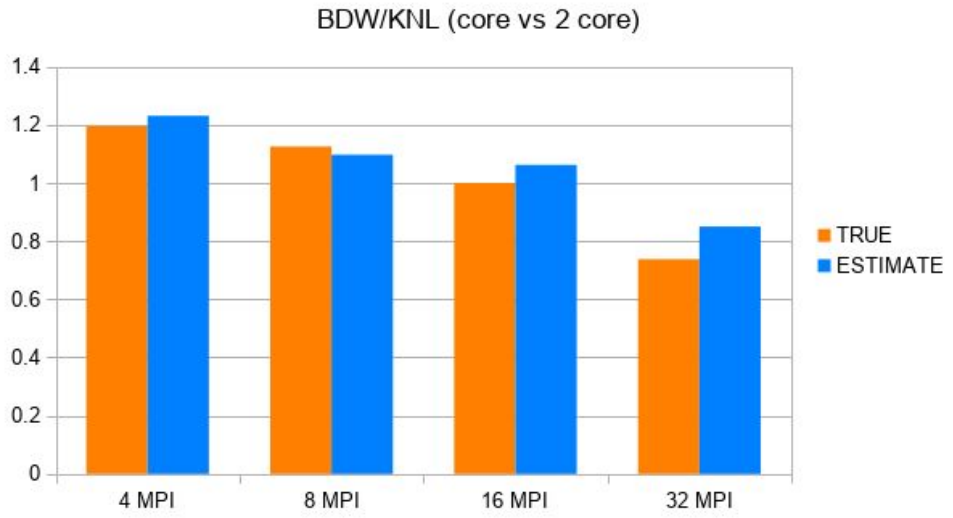
Grafene + Fe, 2D, 127 atoms, 6 k-points

Results

Relative time between different generations of HW.



MnSi - bulk, 64 atoms, 14 k-points



Grafene + Fe, 2D, 127 atoms, 6 k-points

Conclusions

- No rocket science! Select relevant kernels and find meaningful variables to evaluate the performances.
- The **tricky task** is reconstructing the subroutine **call tree**.
- Takes little time! For pw.x, the preliminary work presented here was done in **1 week of profiling** and **two weeks of development/test**.
- Results already presented and used in co-design meetings.

Future and perspectives

- Expand the model to
  - Parallel diagonalization
  - Task groups
  - Better unbalance description
  - Mixed intra-node and internode communications

```
$ mpirun -np 64 pw.x -ndiag 16 -ntg 2 ...
```

- Create and distribute automatic mini-benchmark tools
- Link hardware details to mini-benchmark results
- Training with (and adoption in) AiiDA