EPW performance assessment report

Document Information

Reference Number	POP_AR_28
Author	Brian Wylie (JSC)
Contributor(s)	Ilya Zhukov (JSC)
Date	September 5, 2016

Notices: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676553.*

©2015 POP Consortium Partners. All rights reserved.

Contents

1	Background	3
2	Behaviour and syntactic structure	3
3	Focus of analysis (FOA)	4
4	Parallel Efficiency Metrics	6
5	Load Balance	6
6	Serial performance	7
7	Communications	7
8	Summary of observations	7

1 Background

Applicants Name: Samuel Poncé
Institution: University of Oxford, UK
Application Name: EPW, version 4.0.0
Programming Language: Fortran90
Programming Model: MPI
Source Code Available: yes (GPL)
Input data: GaN/epw-CB-4q (polar wurtzite gallium nitride crystal with 64 k-points)
Performance study: check (audit)
User description: Currently the EPW code relies on MPI parallelization and scales correctly up to 200 cores. We would like to improve scalability to 1000 cores and also optimize the code for improved performance. We would be happy to be have an audit to identify the bottlenecks in the code and focus on those.
Application Description: EPW (www.epw.org) is an Electron-Phonon Wannier code which calculates properties related to the electron-phonon interaction using Density Functional Perturbation Theory and Maximally Localized Wannier Functions. It is distributed as part of the Quantum ESPRESSO suite.
Testcase Description: 48 MPI processes on 2 compute nodes.
Machine Description: ARCHER Cray XC30 at EPCC, comprising 4920 compute nodes, with dual 12-core Intel Xeon E5-2697v2 (Ivy Bridge) 2.7 GHz processors sharing 64GB or memory and joined by two QPI links, connected via proprietary Cray Aries interconnect (Dragonfly topology). PrgEnv-intel using Intel 15.0.2.164 compilers.
Analysis tools: Score-P/2.0.2, Scalasca/2.3.1, PAPI/5.4.1 (following hardware counters were collected: PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_LD_INS, PAPI_L1_DCM). Score-P default (compiler+MPI) instrumentation, combined with runtime measurement filter specifically for FFTXlib fftw routines.

2 Behaviour and syntactic structure

EPW is executed following several short QE/PW SCF/NSCF (plane-wave (non)self-consistent field) electronic band structure data preparation steps with the same number of MPI ranks. The EPW execution consists of two parts: interpolation from coarse Bloch grid to real-space Wanner [`elphon_shuffle`, etc], followed by interpolation from real-space Wannier to dense Bloch grid [`ephwann_shuffle`]. In the provided testcase, the initial coarse grid has 64 k-points and a random fine grid has 231 k-points (whereas more realistic configurations would have many more).

The execution timeline of EPW in Figure 1 clearly shows two phases and some of their internal structure. Whereas `ephwann` is characterised by 10,000 fine-grained iterations (each with short `selfen_elec_q` calls, apart from the final call which collates and writes output files), `elphon` starts with the serial `createkmap_pw2` followed by 12 instances of purely computational `gmap_sym` alternating with varying numbers (up to 12) of `elphon_shuffle` calls each containing FFT communication.

MPI collective communication calls are generally preceded by an explicit `MPI_Barrier` (which unfortunately prevents distinction of `MPI_Bcast` and `MPI_Allreduce` in the timeline view), however, they indicate various computational load imbalances. During `elphon_shuffle`, the first 16 MPI ranks spend much less time waiting in `MPI_Barrier` than the remaining 32 ranks (16 to 47) which wait approximately half of the time. Within `ephwann_shuffle`, there's

Figure 1: Execution timeline of EPW GaN testcase execution on two Archer Cray XC30 compute nodes each with 24 MPI ranks (48 MPI processes).
Timeline chart of 48 MPI processes at top and callstack chart of MPI rank 0 at bottom shows two phases with `ephwann_shuffle` (pale green) — including `selfen_elec_q` (yellow) and `ephwan2blochp` (blue) — preceded by `createkmap_pw2` (cyan) and `elphon_shuffle` (light green) alternating with `gmap_sym` (pale blue) and related routines. Other application routines are dark green and MPI routines are shown in red.

a rather more subtle imbalance with the last 9 MPI ranks (39 to 47) spending over one-fifth of the iteration time waiting, and four MPI ranks (6 to 9) spending the entire time waiting between `ephwann2blochp` calls when other ranks execute `rgd_blk_epw2`.

3 Focus of analysis (FOA)

A profile showing a simplified call-tree in Figure 2 (left) shows the proportion of time on key callpaths, with a region profile (right) sorted by inclusive time.

The simulation setup (mostly `wann_run`) is relatively short and negligible compared to the

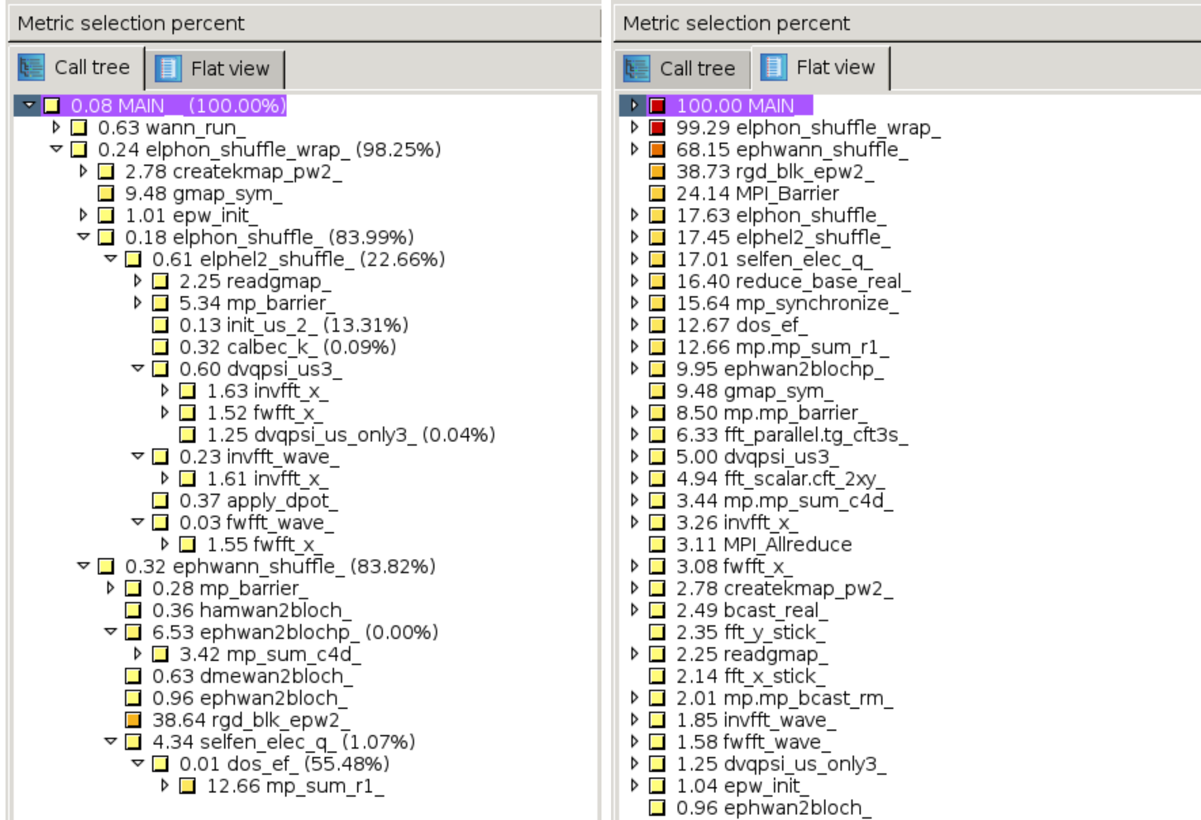

Figure 2: EPW GaN execution (simplified) syntactic structure showing percentage of total time for key callpaths (48 MPI process execution on Archer). On right, region profile sorted by inclusive time.

two significant `elphon` and `ephwann` phases, each of which manifests distinct execution characteristics and is worth considering separately. Although `ephwann_shuffle` is executed within `elphon_shuffle_wrap`, it is preferred to extract it and consider the remainder of `elphon_shuffle_wrap` (including `elphon_shuffle`) as `elphon+`.

Table 1 shows although most of the total execution time was computation, MPI collective operations accounted for one quarter overall. Only a few percent was actual collective communication, with the majority being preceded by (explicit) barrier synchronization. While `ephwann_shuffle` did more collective communication, and `elphon` correspondingly less, with respect to barrier synchronization time the situation is reversed.

Table 1: Percentage of total execution time of EPW's specific regions (48 MPI processes)

Part of application	Percentage of total execution time, %			
	Computation	MPI collective operations		Rest of MPI
		synchronization	communication	
Combined	74	23	3	0
- `elphon+`	66	33	1	0
- - `createkmap_pw2`	2	98	0	0
- - `elphon_shuffle`	56	43	1	0
- `ephwann_shuffle`	78	17	4	0
- - `ephwann2blochp`	71	1	28	0

4 Parallel Efficiency Metrics

Basic parallel efficiency metrics are shown in Table 2. The higher the value (closer to 1.00) then the better is the efficiency. *Load balance* is the ratio of average computation to maximal computation time. *Communication* efficiency is the ratio of maximal computation to maximal executing time, and also the product of *Serialization* reflecting loss caused by dependencies between processes that result in blocked/waiting time and *Transfer* efficiency which quantifies loss due to actual data transfer. *Parallel efficiency* is the ratio of the average computation time to the maximal executing time which is also the product of *Load balance* and *Communication*.

Table 2: Parallel efficiency metrics for EPW and selected regions

Region	Load balance	Communication	Serialization	Transfer	Parallel
MAIN	0.81	0.91	0.94	0.97	0.74
- `elphon+`	0.67	0.99			0.66
- - `createkmap_pw2`	0.02	1.00	1.00	1.00	0.02
- - `gmap_sym`	1.00	1.00	1.00	1.00	1.00
- - `elphon_shuffle`	0.57	0.98	0.99	0.99	0.56
- `ephwann_shuffle`	0.87	0.90	0.94	0.96	0.78
- - `ephwan2blochp`	0.99	0.71	0.99	0.73	0.71
- - `rgd_blk_epw2`	0.81	1.00			0.81

The Table 2 gives an overview of the parallel efficiency of the provided EPW testcase execution with 48 MPI processes. Computation load balance of 87% in `ephwann` is only fair, while in the remainder (`elphon+`) it is a rather poor 67%. Communication efficiency of 90% is fair for `ephwann`, from a combination of blocking/waiting time diminishing serialization efficiency and transfer inefficiency for copious reductions in `ephwan2blochp`, but an almost perfect 99% for the remainder. Overall parallel efficiency of 78% for `ephwann` is low, and considerably worse with 66% for the remainder.

Sub-region `gmap_sym` has almost perfectly balanced computation and `rgd_blk_epw2` has rather imbalanced computation, and neither has any MPI communication, whereas `createkmap_pw2` only has computation on rank 0 with a subsequent MPI barrier.

5 Load Balance

Excellent load balance for `gmap_sym` and `ephwan2blochp` routines combines with much poorer load balance for other parts of the code where most time is spent. `createkmap_pw2` and `readgmap` are only executed by rank 0, for the worst load balance. Within `elphon_shuffle`, 32 MPI processes only work half as much as the first 16, due to the distribution of 64 course-grid k-points.

During `ephwann_shuffle`, a similar (but less serious) load imbalance arises from the distribution of the 231 fine-grid k-points over the 48 MPI ranks resulting in the first 39 ranks having 5 bands to process compared to only 4 for the remaining 9 ranks. Furthermore, four MPI ranks (6 to 9) are entirely without work, and several others underloaded, presumably due to the nature of the particular computations in `rgd_blk_epw2` and `selfen_elec_q`. Additional load imbalance is observed that also varies from iteration to iteration.

6 Serial performance

Profile measurements including hardware counters showed an average of 1.75 instructions executed per CPU cycle (overall and within `ephwann_shuffle`), which seems like a reasonable instruction-level parallelism. The first level data cache miss rate of 0.022 overall, is considerably better at 0.009 for `ephwann_shuffle` and rather higher with 0.065 for the remainder.

7 Communications

Only MPI collective (and no point-to-point) operations are used, accounting for 26% of total time. 87% of this is `MPI_Barrier` synchronization, which often explicitly precedes collective communication with `MPI_Bcast` or `MPI_Allreduce`.

98% of the almost 200k `MPI_Allreduce` operations, transfering an average of 34MB per instance and rank, are part of `ephwann` (with the vast majority in `ephwan2blochp`) and take a total of 288 seconds on average. Of 3565 `MPI_Bcast` operations at the start of `ephwann`, taking a total of 4.3 seconds on average, 1851 have rank 0 as root and within `epw_read` 1714 use MPI_COMM_SELF on each of the other MPI ranks.

While `MPI_Allreduce` operations are much less common in the `elphon` phase, `MPI_Bcast` operations are more than five times more frequent, transfer three times more bytes and take correspondingly longer.

8 Summary of observations

From the performance analysis of the EPW GaN testcase it is possible to conclude the following:

- Execution time is defined by two distinct phases, `elphon` and `ephwann`, with the latter requiring roughly twice as long as the former.
- Load balance for the `elphon` phase seems to be determined by the 64 course-grid k-points, and for the `ephwann` phase by the 231 fine-grid k-points (and their respective characteristics).
- Serial execution of `createkmap_pw2` (and to a much lesser extent `readgmap`) are already noticable overheads which will impact scalability.
- Run-to-run execution time variation of around 5% is quite high, but likely due to significant amounts of file I/O.
- Explicit MPI barrier synchronisation is used before collective communications to workaround issues of some MPI implementations (OpenMPI). If not actually required, these may be worth eliminating (though unlikely to significantly improve performance).

Recommendations

- Investigate replacing FFTW (from the internal FFTXlib) with DFTI from Intel MKL (mkl_cdft_core) to compare performance.
- Investigate the extent to which additional cores can be used effectively. Scaling the provided GaN test case to 64 cores should be straightforward, but further cores are likely to idle during `elphon`. Clarify whether execution with more than 231 cores is meaningful.

- To the extent that inherent per-process computational imbalance is unavoidable, it may be advantageous to ensure that the excesses are distributed as equally as possible over compute node (and sockets) to most effectively exploit available memory bandwidth.
- Any unnecessary file I/O (such as diagnostics) may be worth eliminating, to potentially improve performance and reduce variability.
- Quantify file I/O performance (and variability).