Building a Library of Simulated Atom Probe Data for Different Crystal Structures and Tip Orientations Using TAPSim Supplementary Material

Markus Kühbach, Andrew Breen, Michael Herbig, Baptiste Gault

Max-Planck-Institut für Eisenforschung GmbH, Max-Planck-Str. 1, D-40237 Düsseldorf

Abstract

The process of building an open source library of simulated field desorption maps for differently oriented synthetic tips of the face-centered cubic (FCC), body-centered cubic (BCC), and hexagonal-close-packed (HCP) crystal structures using the open source software TAPSim is reported on. Specifically, the field evaporation of a total set of 4×101 single-crystalline tips was simulated. Their lattices were oriented randomly to sample economically the fundamental zone (FZ) of crystal orientations. Such data is intended to facilitate the interpretation of low density zone lines and poles that are observed on detector hit maps during Atom Probe Tomography (APT) experiments.

The datasets and corresponding tools have been made publicly available to the APT community in an effort to provide better access to simulated atom probe datasets. In addition, a computational performance analysis was conducted from which recommendations are made as to which key tasks should be optimized in the future to improve the parallel efficiency of TAPSim .

Keywords: Atom probe crystallography, Atom Probe Tomography, TAPSim, machine learning, data mining

Email address: m.kuehbach@mpie.de (Markus Kühbach, Andrew Breen, Michael Herbig, Baptiste Gault)

0.1. Performance analyses

Benchmarking library construction. To best serve the interests of different readerships, the following section is split into two parts. The first is primarily addressed to APT experimentalists. Therein, the elapsed time results are delivered to answer how long the TAPSim simulations took. The second part is addressed primarily to developers of APT simulation software. It focuses on how the individual computing tasks contribute to the elapsed time and details a profiling of parallel performance. Based on these findings, a set of practical recommendations is concluded which are of interest for practitioners and developers.

Table 1 summarizes measured elapsed time results from the 404 tips library study. Individual simulations took between 34 h and 55 h. Differences in the workstation floating point performance, in combination with the non-exclusive execution protocol used within the daily research environment, caused these fluctuations. The net simulated evaporation rates in atoms per minute are in reasonable agreement with those reported by Oberdorfer et al. (Oberdorfer, 2014; Oberdorfer et al., 2013) ($\lessapprox 300\,\mathrm{min}^{-1}$) and Jägle et al. (Jägle et al., 2014). However, care should be exercised when comparing the performance reported in this study to that found in the literature as no specific details about the number of threads or workstations used in these other studies were reported.

Given that the clarity of the simulated detector hit maps improves with increasing tip radii and that the simulated tips are still at least one, if not two, orders of magnitude smaller in volume than those typically measured in APT experiments, the elapsed timings for larger tips is of practical interest. Such information was accessible in the second study, where tips with successively larger half-sphere radii were simulated. These simulations were executed with the same level of parallelism ("–threads=4") on workstation 16 (which was exclusively working on this job). Despite such equal execution conditions, Fig. 1 shows a reduction in evaporation rate with increasingly larger defined tip volume. One reason for this is the higher cost for performing the trajectory

Table 1: Performance analysis of the simulations showing the key mesh properties: the number of atoms (N_{at}), support grid points (N_{sp}), the shortest elapsed time achieved for the relaxation (t_{rx}) and evaporation simulation (t_{evap}), respectively for each atom type. $j_{evap} = \frac{N_{at}}{t_{evap}}$ reports the range of individual mean evaporation rates for each atom type collective. Deviations in the total number of integration points due to the fact that meshes in different orientations were sampled were with less than 1 % deemed insignificant. The workstation IDs refer to Tab. 2.

	N_{at}	N_{sp}	t_{rx}	t_{evap}	j_{evap}	**Workstation**
			min	min	min^{-1}	
Al	2.986×10^5	1.167×10^6	437	2847	73.1 to 104.5	16
Mg	2.111×10^5	1.052×10^6	253	1801	78.5 to 117.1	07
Zr	2.107×10^5	1.051×10^6	215	1943	46.0 to 73.6	06
W	3.095×10^5	1.180×10^6	252	1801	83.4 to 159.2	15

computations. This is one of several factors which currently hinder the scalability of TAPSim simulations, whether parallel processing is used or not. This will be detailed in the next section.

Detailed benchmarking. This paragraph summarizes key performance results from the Meshgen and TAPSim profiling study — the Mg tips in the reference orientation. Instead of quantifying performance using elapsed time data, focus is put on a dimensionless quantity which characterizes the efficiency of a program - the CPI index. The CPI relates how many useful CPU program instructions were successfully completed to how many CPU cycles this took. Different computational task have different CPU cycle count demands[1]. For this reason average CPI values (scores) per function call are reported. A net

[1]Examples are the reading of data from main memory (millions of cycles), performing arithmetic operations on data in the fastest CPU cache (1 cycle), evaluating trigonometric functions or re-organizing the instruction workflow upon branch misprediction (approximately a dozen cycles)

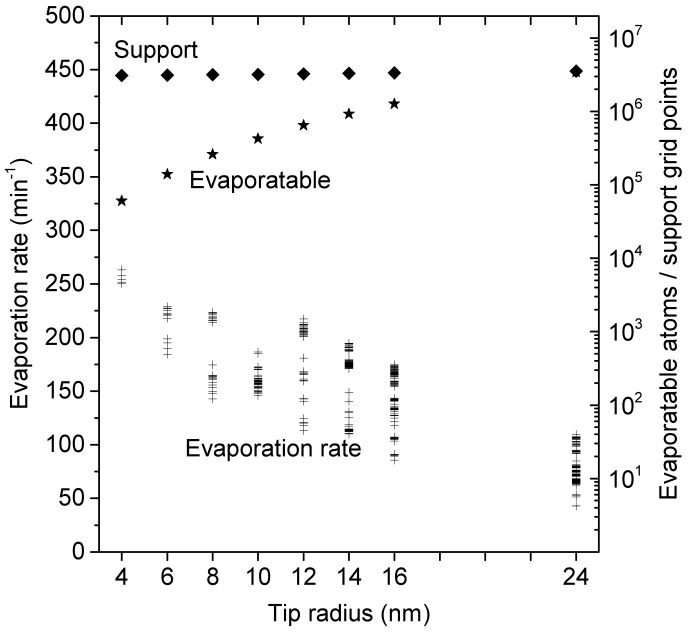
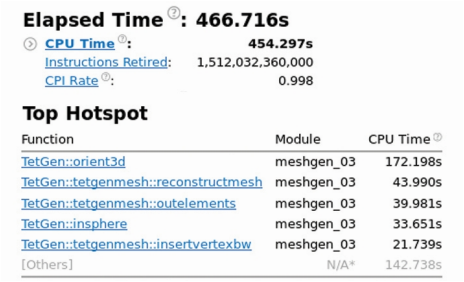


Figure 1: All tips in the second study used the same workstation and level of parallelism. Nevertheless, their net efficiency was found to be systematically lower the larger the tip volume.

CPI score of 1 indicates a reasonably efficient program. However, given that modern CPU cores can execute multiple arithmetic operations in one cycle, lower CPI scores are in principle achievable, and should be aimed for during optimizing a program [2].

Observing a $CPI = 1.0$ substantiated that Meshgen executed reasonably efficiently. It spent most of its time in the sequential meshing library TetGen (Si, 2015). The meshing itself took 8 min. This was comparable to the tip synthesis time with VESTA and MATLAB . Conversely, TAPSim (Fig. 2b) reached a less efficient score of $CPI = 1.36$ when executing sequentially. In the benchmark

[2] Facing super-scalar multi-level cache hierarchy architectures the discussion is more complicated than can be summarized here. The interested reader is referred to (Hennessy & Patterson, 2012) for details about such streaming instruction level parallelism and vectorization.



Elapsed Time: 466.716s

CPU Time: 454.297s
Instructions Retired: 1,512,032,360,000
CPI Rate: 0.998

Top Hotspot

Function	Module	CPU Time
TetGen::orient3d	meshgen_03	172.198s
TetGen::tetgenmesh::reconstructmesh	meshgen_03	43.990s
TetGen::tetgenmesh::outelements	meshgen_03	39.981s
TetGen::insphere	meshgen_03	33.651s
TetGen::tetgenmesh::insertvertexbw	meshgen_03	21.739s
[Others]	N/A*	142.738s

(a) Meshgen

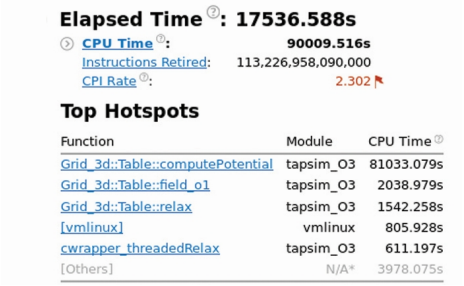
Elapsed Time: 31521.192s

CPU Time: 31194.046s
Instructions Retired: 67,822,169,140,000
CPI Rate: 1.363

Top Hotspots

Function	Module	CPU Time
Grid_3d::Table::computePotential	tapsim_O3	25950.206s
Grid_3d::Table::field_o1	tapsim_O3	1387.255s
Grid_3d::Table::relax	tapsim_O3	757.939s
_int_free	libc-2.23.so	433.984s
_int_malloc	libc-2.23.so	345.766s
[Others]	N/A*	2318.896s

(b) TAPSim , 1 thread



Elapsed Time: 17536.588s

CPU Time: 90009.516s
Instructions Retired: 113,226,958,090,000
CPI Rate: 2.302

Top Hotspots

Function	Module	CPU Time
Grid_3d::Table::computePotential	tapsim_O3	81033.079s
Grid_3d::Table::field_o1	tapsim_O3	2038.979s
Grid_3d::Table::relax	tapsim_O3	1542.258s
[vmlinux]	vmlinux	805.928s
cwrapper_threadedRelax	tapsim_O3	611.197s
[Others]	N/A*	3978.075s

(c) TAPSim , 12 threads

Figure 2: Using advanced hot spot analyses it was identified how long it took to execute a) Meshgen sequentially and b) TAPSim sequentially versus c) in parallel.

studies, TAPSim maintained an average processing rate of $\approx 270\,\mathrm{min}^{-1}$ atoms per minute. The higher evaporation rate is a consequence of the exclusive utilization of workstation 06 and its higher CPU clocking.

Intuitively, one seeks to speed up the simulation (Fig. 2c) by using more cores. When 12 cores were used on a single simulation, the elapsed time was

approximately halved in comparison to sequential execution. However, the 11 additional cores were not fully loaded. As such, in terms of High Performance Computing (HPC) resource utilization economy, moderately scaled parallelized TAPSim simulations generated unnecessary additional core-hour consumption compared to scheduling batches of individual, sequentially processed tip simulations, trivially in parallel.

There are two key observations which quantify scalability limitations of TAPSim when it runs in parallel. Firstly, a higher $CPI \approx 2.3$ than expected. Further, an approximate (1.7x) as many instructions to execute in total compared to when running sequentially. To argue that this overhead is merely due to work sharing constructs, which every parallel program has to execute, in addition to its pure workload in sequential execution, is not convincing. Rather, it is a systematic overhead. This conclusion is supported by the higher CPI value, which, quantifies the mean efficiency of the entire simulation. In fact, if TAPSim were to also execute the additional work sharing constructs as efficiently as it executes its sequential workload, the CPI score should remain approximately 1. Hence, to pinpoint the reason for such low efficiency, it is useful to study how efficiently TAPSim kept the cores busy (Fig. 3).

Compared to the sequential execution, the load gets dissimilarly distributed across the cores when running in parallel. Specifically, the first core is forced to process most of the work, while the others idle frequently. In fact, five cores are kept busy where 12 would have been expected for an ideally strong scaling program. In effect, Fig. 3 quantifies that TAPSim is choked by load imbalance and sequential overhead. An even closer inspection of the elapsed time expenditures for individual functions of TAPSim is detailed in Fig. 4. A top-down analysis proves that the computePotential() part in particular, in which $\gtrapprox 83\,\%$ of the work accrues, executes less efficiently when utilizing more cores. Any attempt to optimize TAPSim in the future should focus on this part first.

One may argue that throughout the analysis done so far it was implicitly assumed that all computational tasks were ideally thread-parallelized. This,

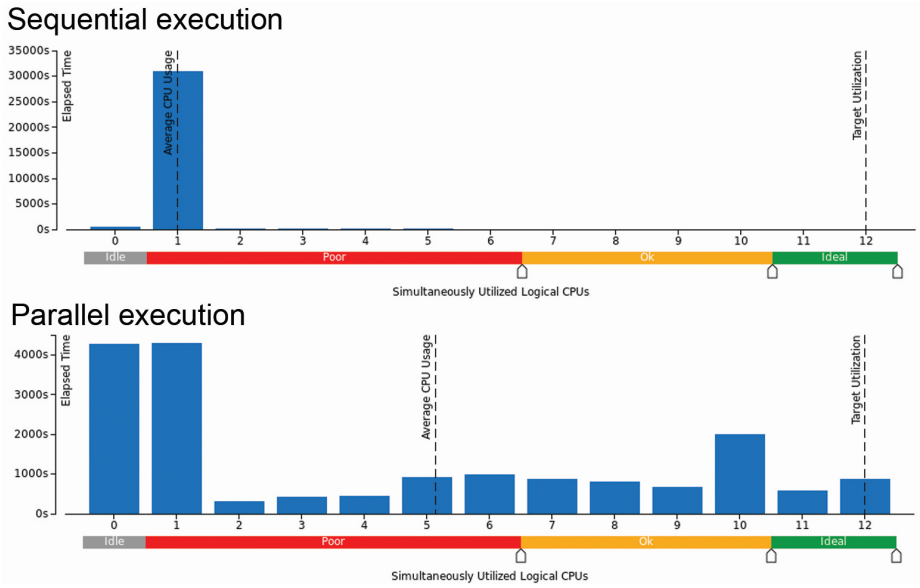


Figure 3: Core utilization for sequential and parallel execution of TAPSim. In practice, TAPSim jobs run a factor of two faster when using 12 threads instead of only one. However, this is at the cost of substantially more computational resources; most of which then work less efficiently. This is because the computational load is unevenly distributed across the cores and substantial sequential overhead exists.

however, is not the case for TAPSim — only its potential relaxation is fully parallelized. Consequently, only a fraction of the sequential runtime is reduced. The resulting maximum speedup can be estimated using Amdahl's law (Amdahl, 1967) where p and $1-p$ specify the parallel and sequential execution time fractions respectively, and n, applied to the present example, the number of threads in use. According to Fig. 4 a fraction of 80 % was spent in computePotential(). Consequently, a reduction of the sequential runtime to at most 0.26 is expected. Considering that the latter function labels a collection of multiple tasks (Fig. 5), it is evident that the actual elapsed time fraction TAPSim spent in parallel was even lower. Using $p = 0.8 \cdot 0.881 \approx 0.7$ evaluates, to a maximum practically, achievable speedup of ≈ 3. To measure the quantitative data in Fig. 5, an additional simulation of the Magnesium tip in the reference configuration was executed sequentially. Therein, workstation 16 was used exclusively and

Sequential execution

Function Stack	CPU Time: Total — Effective Time by Utilization (Idle, Poor, OK, Ideal, Over)	CPU Time: Self — Effective Time by Utilization (Idle, Poor, OK, Ideal, Over)	Instructions Retired: Total	CPI Rate: Total
Total	100.0%	0s	100.0%	1.363
Grid_3d::Table::computePotential	83.2%	25950.206s	70.7%	1.621
▸ Grid_3d::Table::field_o1	4.4%	1387.255s	8.2%	0.707
▸ Grid_3d::Table::relax	2.4%	757.939s	2.3%	1.108
▸ _int_free	1.4%	433.984s	3.6%	0.520
▸ _int_malloc	1.1%	345.766s	3.1%	0.498
▸ cwrapper_threadedRelax	1.1%	333.131s	3.0%	0.475
▸ __libc_malloc	1.0%	318.715s	1.8%	0.742
▸ std::_Rb_tree_increment	0.5%	164.780s	0.4%	1.782

Parallel execution

Function Stack	Effective Time by Utilization (Idle, Poor, OK, Ideal, Over)	Effective Time by Utilization (Idle, Poor, OK, Ideal, Over)	Instructions Retired: Total	CPI Rate: Total
Total	99.9%	0s	100.0%	2.302
Grid_3d::Table::computePotential	90.0%	81033.079s	73.4%	2.829
Grid_3d::Table::field_o1	2.3%	2038.979s	6.9%	0.754
Grid_3d::Table::relax	1.7%	1542.258s	2.6%	1.462
▸ [vmlinux]	0.9%	805.928s	0.4%	4.262
▸ _int_free	0.7%	604.988s	3.0%	0.537
▸ cwrapper_threadedRelax	0.7%	611.197s	3.2%	0.468
▸ _int_malloc	0.5%	479.403s	2.5%	0.517
▸ __libc_malloc	0.5%	447.834s	1.6%	0.737
▸ std::_Rb_tree_increment	0.3%	306.431s	0.2%	3.597
▸ lu_decmp<float, (int)3>	0.2%	214.525s	0.9%	0.608

Figure 4: A profiling of the individual function calls reveals that the most significant work package is the potential relaxation. Execution of exactly this part, though, becomes more inefficient when running in parallel as documented through a higher $CPI \approx 2.8$ compared to the sequentially achieved $CPI \approx 1.6$. This pinpoints that primarily load imbalance and work organization overhead choke performance.

100 TAPSim monitored via its VERBOSE option.

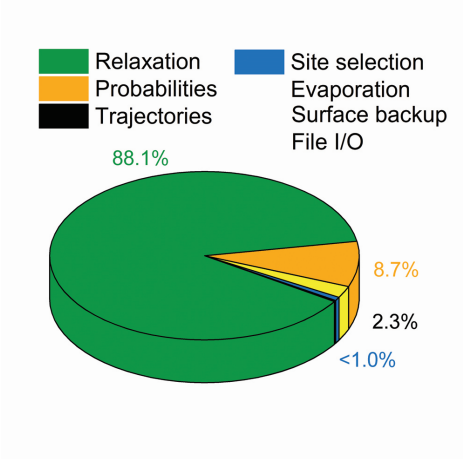


Figure 5: Analyzing the individual computational tasks identifies an approximate runtime fraction of 88 % for the relaxation.

The above quantitative corrections put the scalability of TAPSim into per-

spective. They pinpoint that technical modifications to the code base are necessary to enable more productive TAPSim simulations, in terms of efficiently simulating tips similar to that observed experimentally as well as improving the parallel efficiency of the application. The latter is useful in particular to remain compliant and competitive with the ever stricter strong scalability requirements of HPC architectures such as supercomputers. While implementing these changes is beyond the scope of this work, it provides insight into the execution of the current source code sections and areas for improvement potential.

Source code analysis to identify improvement potential. Specifically, the pthread library cwrapper core functions threadedRelax(), threadedLocalRelax(), and aforementioned computePotential() are of interest. With these, TAPSim cycles through a global data structure, the cell grid (referred to as "data" in the respective source code sections) and a collection of associated cell nodes (Oberdorfer, 2014). The latter are implemented as C++ class objects and stored in a global array of nodes. Individually, these objects allocate additional shared memory snippets to store nodal pieces of information for book-keeping localized references to neighboring nodes and weighting factors to accelerate the relaxation computations. The cell nodes work as follows: for each cell, its neighbors are identified. Next, their potential values are collected to compute a new value. Thereafter, the new value is written back. The processing of cells for each relaxation cycle is distributed on the threads by assigning them a number of predefined nodes to process. The procedure, as in the current code, causes a blockage because all threads have to complete their work package before a new cycle is initiated.

For such an algorithm and data structure to run efficiently, several implementation measures must be optimized. First of all, the partitioning of the cells to their respective threads should result in a workload which is as balanced as possible. Otherwise, individual threads idle before a new relaxation cycle is initiated, causing an increase in CPU cycles, i.e. higher CPI values.

Given that collecting potential values of neighboring cells is one of the key

tasks in the algorithm, it is also essential to do so most efficiently. This requires the most efficient use and re-utilization of data inside the fastest CPU caches. Currently, though, TAPSim employs a data structure which, by design, provokes the spreading of nodal pieces of information in main memory because they are stored in individual, per cell, allocated memory snippets. However, upon allocation with standard allocator implementations, there is no guarantee that such individually allocated snippets are either properly aligned at cache boundaries or refer to nearby main memory addresses at all (Hennessy & Patterson, 2012). As this fragmentation affects all cells, its induced effects on performance are expected to be more severe with larger numbers of cells, i.e. the tip volume studied. In effect, TAPSim currently dereferences effectively slower responsive memory locations which for memory bound algorithms will increase the CPI value (Hennessy & Patterson, 2012).

This source code analysis-based insight is substantiated by the measured profiling data (Fig.2). They explain how TAPSim displays a high $CPI = 1.6$, even during sequential execution. This identifies idling and fetching of local data from remote memory as one effective performance brake.

0.2. Recommendations for future code design

For the sake of completeness it follows a copy of the recommendation section from the main paper. The above findings prompt several recommendations to TAPSim users and developers of TAPSim kind APT simulation tools.

- The TAPSim default setting to use all available cores on the target system[3] should be replaced by running sequentially ("–threads=1") by default. Instead, the user should specify if additional threads should be used and how to make a compromise between speeding up the simulation yet avoiding unnecessary blocking of many cores. Using Amdahl's law, the present data suggest using thread counts between 1 and 4 is best to achieve this compromise.

- It is the total number of integration points rather than the number of atoms in the tip to evaporate only which defines the runtime of TAPSim . Therefore, it should become best practice in papers to include all significant pieces of information when reporting elapsed time data. These should be at least the version of TAPSim used, the number of ions in the actual tip, the number of support mesh points, the CPU type, and the execution protocol used, i.e. how many cores for threading were instructed and whether or not they were used exclusively.

- Proper reporting of hardware details in the literature when describing the run time performance of TAPSim . In particular, the affect of the inputted tip radii size on the evaporation rate. Oberdorfer reported (Oberdorfer et al., 2013; Oberdorfer, 2014) an order of magnitude lower evaporation rate $12\,\mathrm{min}^{-1}$ when executing a large simulation comprising 28×10^6 atoms, with 56×10^6 points in the emitter structure, and a total of 328×10^6 cells. The result was attributed to trajectory computations as the main

[3] through at-runtime-executed system calls using "_SC_NPROCESSORS_ONLN"

bottleneck. The argument is reasonable given that TAPSim implements a recurring sequence of parallelized local relaxation, candidate selection, and sequential trajectory integration but the explanation lacked information on the hardware details, which are required to properly understand this performance report quantitatively.

- To the best of our knowledge, this study revealed an undocumented implementation error in TAPSim . It causes some tips to fail and affects system performance critically when doing so. The issue manifests as an uncontrolled allocation of main memory within a never exiting while loop inside the get_voronoiFace() function. This is likely due to numerical side effects which occur during code execution, regardless of the optimization level chosen. This results in a continuous allocation of memory, eventually leading to system exhaustion[4].

- Consider implementing a re-meshing strategy. The current design uses the same Voronoi cell spatial resolution of the tip throughout the entire simulation. Once an atom is evaporated, its corresponding node cell type is switched from a removable atom to represent vacuum. In effect, trajectory computations become successively finer for atoms which evaporate later in the sequence as their flight path is integrated along a chain vacuum cells that once represented the tip. A re-inspection of the already existent adaptive solving strategy within TAPSim or an additional implementation of a discontinuously applied fusing of cells into larger is likely useful. Primarily if this allows to reduce the size of the node array and thereby cell structure querying costs.

- Partition the global node array, which stores the cells, into multiple arrays.

[4]This error was located within the obj−>tets.front() != obj−>tets.back() while loop (line 1301, geometry_3d.cpp, TAPSim v1.0b rev3225). Instrumented source code and a documented example was made available as supplementary material (Kühbach & Breen, 2018) to assist solving the issue.

Do so by sub-dividing the dense inner grid layer of the tip into spatially disjoint regions. Parallelized adaptive meshing tools (de Cougny et al., 1994; de Cougny & Shephard, 1999; Fryxell et al., 2000; Loseille et al., 2015; Owen et al., 2017) are an alternative to consider when improving the meshing process.

- In each such mesh region, all nodes should be stored in a contiguous array. Each thread then gets assigned one region to process. Individually, their arrays should be allocated via thread-local memory, potentially making even use of alternative allocator libraries[5], to improve memory locality. Local pieces of information for each node should always be held in aligned arrays.

- Utilize in-time parallelism and double buffering strategy. This will be useful because trajectory computations are independent from local relaxation computations, provided that a copy of potential values for the previous state in the evaporation sequence exists. Admittedly, storing multiple copies of field values may appear counter-intuitive as one usually seeks to save memory. However, allocating additional memory for carrying the previous electrostatic potential values eventually allows one to decouple the temporal sequentiality of the current evaporation algorithm into a dispatching of trajectory computations to workers. Such in-time parallelism is likely to overcompensate for the costs of allocating additional memory.

- Explore more computationally demanding trajectory integration, such as a more realistic chamber geometry with a counter electrode and handle trajectories against impenetrable tomograph chamber parts in the flight path.

- Switch from using pthreads to the Open Multi-Processing (OpenMP) Ad-

[5] such as jemalloc (https://github.com/jemalloc/jemalloc) or tcmalloc (http://goog-perftools.sourceforge.net/doc/tcmalloc.html)

vanced Programming Interface (API) (Chandra et al., 2001), for instance, for improved flexibility in the partitioning and controlling of load partitioning.

Acknowledgements

MK gratefully acknowledges the funding received from the German Research Foundation through project BA 4253/2-1. AJB acknowledges the Alexander von Humboldt Foundation (AvH) for partially funding this research through a Humboldt postdoctoral research fellowship as well as fruitful discussions about machine learning applications with Ye Wei. Achim Kuhl and Berthold Beckschäfer are acknowledged for their setting up and maintaining of the workstations.

Work distribution

MK implemented the computational tools and conducted the analyses. AB plotted and interpreted the crystallographic information on the simulated detector hit maps. MK and AB initiated and discussed the work and wrote the paper. MH and BG provided important insight and discussion throughout the duration of the study.

Appendix

References

Amdahl, G.M. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computer Capabilities, *AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, vol. 30, 483–485.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. & Menon, R. (2001). *Parallel Programming in OpenMP*, Morgan Kaufmann, San Francisco, 1 ed.

Table 2: Technical details of the Ubuntu operating system- (OS) powered in-house workstations: "OPT" is short for Opteron. "Gold" short for Xeon Gold. "C/S" reads as an abbreviation for physical cores per socket. Lower-level caches L1, L2 are per core in kB. The last level cache L3 size in MB. Main memory "RAM" in GB. Workstation 06 cores do not support hyper-threading.

MAWS	**OS**	**CPU**	**GHz**	**C/S**	**L1, L2**	**L3**	**RAM**
06	16.04.1	X5670	2.93	6/2	32, 256	12	24
07, 08	16.04.4	OPT 6174	2.20	12/4	64, 512	12	512, 256
15, 16	16.04.4	Gold 6150	2.70	18/2	32, 1024	24.75	576, 192

de Cougny, H.L., Devine, K.D., Flaherty, J.E., Loy, R.M., Ozturan, C. & Shephard, M.S. (1994). Load balancing for the parallel adaptive solution of partial-differential equations, *Applied Numerical Mathematics* **16**, 157–182.

de Cougny, H.L. & Shephard, M.S. (1999). Parallel refinement and coarsening of tetrahedral meshes, *International Journal for Numerical Methods in Engineering* **46**, 1101–1125.

Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W. & Tufo, H. (2000). Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, *The Astrophysical Journal Supplement Series* **131**, 273–334.

Hennessy, J.L. & Patterson, D.A. (2012). *Computer Architectures: A Quantitative Approach*, Morgan Kaufmann, Amsterdam, 5 ed.

Jägle, E.A., Choi, P.P. & Raabe, D. (2014). The Maximum Separation Cluster Analysis Algorithm for Atom-Probe Tomography: Parameter Determination and Accuracy, *Microscopy and Microanalysis* **20**, 1662–1671.

Kühbach, M. & Breen, A. (2018). Milling of single-crystalline tips sup-

plementary material: Zenodo open source data repository, Tech. rep., Max-Planck-Institut für Eisenforschung, GmbH, Düsseldorf, URL `https://zenodo.org/record/1466804`.

Loseille, A., Menier, V. & Alauzet, F. (2015). Parallel generation of large-size adapted meshes, *Procedia Engineering* **124**, 57–69.

Oberdorfer, C. (2014). *Numeric Simulation of Atom Probe Tomography*, Ph.D. thesis, Westfälische Wilhelms-Universität Münster, Münster, Germany.

Oberdorfer, C., Eich, S.M. & Schmitz, G. (2013). A full-scale simulation approach for atom probe tomography, *Ultramicroscopy* **128**, 55–67.

Owen, S.J., Brown, J.A., Ernst, C.D., Lim, H. & Long, K.N. (2017). Hexahedral mesh generation for Computational Materials Modeling, *26th International Meshing Roundtable, (IMR26 2017)*, vol. 203, 167–179.

Si, H. (2015). TetGen, a Delaunay-based quality tetrahedral mesh generator, *ACM Transactions on Mathematical Software* **41**, 1–36.