Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200034

Learning-Based Load Balancing for Massively Parallel Simulations of Hot Fusion Plasmas

Theresa POLLINGER [a,1] and Dirk PFLÜGER [a]
[a] *Institut für Parallele und Verteilte Systeme, Universität Stuttgart*

Abstract The sparse grid combination technique can be used to mitigate the curse of dimensionality and to gain insight into the physics of hot fusion plasmas with the gyrokinetic code GENE. With the sparse grid combination technique, massively parallel simulations can be performed on target resolutions that would be prohibitively large for standard full grid simulations. This can be achieved by numerically decoupling the target simulation into several smaller ones. Their time dependent evolution requires load balancing to obtain near optimal scaling beyond the scaling capabilities of GENE itself. This approach requires that good estimates for the runtimes exist.

This paper revisits this topic for large-scale nonlinear global simulations and investigates common machine learning techniques, such as support vector regression and neural networks. It is shown that, provided enough data can be collected, load modeling by data-driven techniques can outperform expert knowledge-based fits – the current state-of-the-art approach.

Keywords. load balancing, gyrokinetics, exascale, machine learning, sparse grid combination technique, machine learning

1. Introduction

The research on hot fusion plasmas remains a pressing topic, and understanding the relevant processes through simulation is necessary to optimize large experimental reactors such as ITER. While such devices in fusion research are being built, simulation results should always be one step ahead to assist in understanding and planning experiments [1]. This does not simply happen due to Moore's Law (and successors), because the gyrokinetic formulation of the Vlasov-Maxwell equations is at least five-dimensional, and numerical discretizations in higher dimensions suffer the so-called "curse of dimensionality". We have proposed the sparse grid combination technique to mitigate the curse of dimensionality and to gain insight into the physics of hot fusion plasmas [2] with the gyrokinetic code GENE [1]. It replaces the computationally infeasible target solution by a combination of many smaller solutions. Using our parallel combination technique framework, one can compute the partial solutions in parallel process groups, where each

[1] Corresponding Author: Theresa Pollinger, Institut für Parallele und Verteilte Systeme, Universität Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany; Email: theresa.pollinger@ipvs.uni-stuttgart.de

process group is assigned multiple tasks, i.e., partial solutions, to solve [3]. At this point, load balancing becomes crucial, since imbalanced task distributions will lead to unnecessary idle times for thousands of processors. Previous work investigated load balancing for linear local initial-value computations with the combination technique [4]. For simulations that fully capture the global non-stationary behavior of the plasma, this is not sufficient any more. We therefore revisit this topic, in order to facilitate computations at scales that have not been attempted with the combination technique before.

Accordingly, this paper first gives a comprehensive introduction to the scientific background, i.e., global nonlinear gyrokinetic simulations with GENE and decoupling via our sparse grid combination technique framework. We discuss how load imbalances can arise by mis-estimating simulation runtimes. As the main contribution of this paper, we improve on the state of the art by introducing machine learning methods for load balancing. We follow Heene et al. [4] and try to find a good a-priori estimate of the runtime of each task. Runtime data on varying tasks is collected depending on simulation parameters and the degree of parallelization. We have compared different methods of predicting the runtimes on previously unseen grids: nearest neighbor interpolation, support vector regression and neural networks, as well as the state-of-the-art model based on expert knowledge. The latter is still a reasonable approach for our purposes, but can be outperformed by a neural-network based prediction. It follows that by collecting data early on, we can save on efficiency losses and re-initialization overhead for large simulations, while being able to extend our models as we collect more runtime data.

2. Scientific Background

2.1. Gyrokinetic Simulations with the GENE Code

The GENE code is a state-of-the-art solver for the Maxwell-Vlasov system of equations, consisting of the Maxwell equations and the Vlasov equation

$$\frac{\partial F_\sigma}{\partial t} + \frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} \cdot \nabla F_\sigma + \frac{\mathrm{d}v_\parallel}{\mathrm{d}t} \frac{\partial F_\sigma}{\partial v_\parallel} + \frac{\mathrm{d}\mu}{\mathrm{d}t} \frac{\partial F_\sigma}{\partial \mu} = 0, \tag{1}$$

which connects the plasma particle distribution function f to the electromagnetic field [5]. While f is actually located in six-dimensional phase space, the gyrokinetic transform reduces these to five, by integrating out the gyration direction of the plasma particles. The remaining cartesian dimensions for GENE's Eulerian approach are denoted by $x, y, z, v_\parallel, \mu$. The time step integration is performed through an explicit fourth-order Runge-Kutta scheme.

Even though the complexity of solving is drastically reduced by the gyrokinetic transform, the computational work required for solving the integro-differential equations still suffers the curse of dimensionality – simulations are unfeasible for high resolutions.

A simplified way of looking at the equations that GENE solves is splitting into a linear and nonlinear part

$$\frac{\partial f}{\partial t} = \mathscr{L}(f) + \mathscr{N}(f). \tag{2}$$

While previous work looked at times measured for executing only the linear part of the simulation – and for relatively low resolutions – this topic needs revisiting, now that we are dealing with nonlinear, global large-scale simulations. The nonlinear part of the model becomes dominant, and to capture the chaotic behavior it produces, higher minimal resolutions are required. As the maximum resolution is further increased, more features can be resolved [5].

At this point, we need to clearly distinguish two different effects that affect the runtime of a GENE simulation: the *time per time step* is the time needed to process one single explicit time step, which we assume to stay constant during the course of a given simulation, and *adaptive time-stepping* needed to ensure stability, which will change nonlinearly with the simulated fields [6].

This paper will focus on the *time per time step* needed for a given simulation grid. This is a reasonable restriction for scenarios where we assume that a combination takes place after all grids have progressed by one time step – more on this in the next section, which focuses on the combination technique for sparse grids.

2.2. Massively Parallel Computation with the Sparse Grid Combination Technique

Our approach to break the curse of dimensionality is the use of the sparse grid combination technique [7]. The basic idea is that we can run the simulation on many relatively coarse anisotropic grids in the index set I; the d-dimensional *level vector* $\vec{\ell} = [\ell_1, \ldots, \ell_d]$ defines each grid's resolution as $2^{\ell_i} + 1$ in dimension i. I contains all $\vec{\ell}$ in the convex hull of the simplex spanned by the minimum level $\vec{\ell}_{min}$ and the corners of the cartesian hypercube between $\vec{\ell}_{min}$ and the maximum level $\vec{\ell}_{max}$. The combination results in a sparse grid representation $f^{(c)}$

$$f^{(c)} = \sum_{\vec{\ell} \in I} c_{\vec{\ell}} f_{\vec{\ell}} \qquad , \qquad c_{\vec{\ell}} = \sum_{\vec{z} \leq \vec{1}} (-1)^{|\vec{z}|_1} \chi_I(\vec{\ell} + \vec{z}). \tag{3}$$

of the solution, defined on a finer (target) grid of resolution $\vec{\ell}_{max}$. χ_I is the characteristic function of I. For a thorough description of sparse grids and the combination technique please refer to [8].

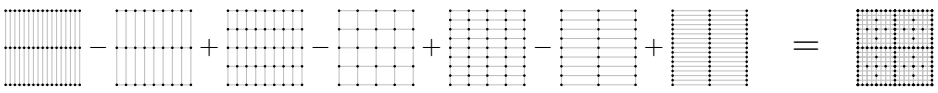

Figure 1. Schematic of the standard sparse grid combination technique with $\vec{\ell}_{min} = [2,2]$ and $\vec{\ell}_{max} = [4,4]$

Existing legacy solvers for cartesian grids can be used in a black-box fashion, and they do not need to be refactored to implement the numerical operators on the sparse grid itself. With respect to the GENE code, this means that one can start multiple instances of GENE to compute the solution to the same physical problem on one of these grids respectively [2]. We will call this combination of simulation parameters and grid resolution $\vec{\ell}$ a *task*. After the simulation has progressed by a defined time interval for each task, the solution is recombined by the use of the combination technique, cf. Fig. 1, such that the values match on all points which are shared between grids. Note that the simulation step for one task is independent of other tasks, meaning that the tasks can be processed in an embarrassingly parallel manner [3].

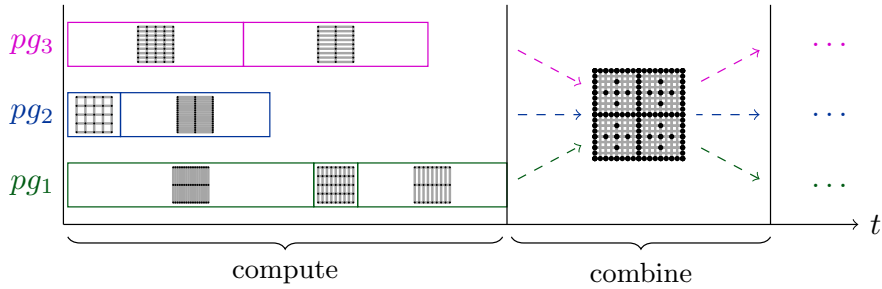

Figure 2. Possible parallel computation scheme of the grids in Fig. 1

Since the GENE simulations considered here are far too large to fit on a single node – usually requiring 20 to hundreds of GB in main memory – our C++ framework for the sparse grid combination technique employs a manager-worker scheme, where a worker consists of a whole process group using up to thousands of cores. The manager distributes *tasks* to the process groups. Currently, the framework is restricted to process groups of the same size only. Each process group will execute the assigned tasks one after the other, each until the simulation time of the next combination step is reached. Then, the grids of the tasks are updated with the results of the other tasks by way of the combination technique. This usually means that some of the process groups will have to wait for the longest-running one to finish, as illustrated by the gap in Fig. 2. If we assume the simplest set-up where every task uses the same time step, the optimization reduces to finding the best possible assignment of tasks to the process groups.

But *what is a well-balanced assignment of tasks with respect to the process groups?* This is the core question of this paper, and the following sections will present an approach to answer it.

3. Data-driven Load Modeling

Load balancing in HPC is often implemented by sophisticated domain decomposition schemes [9], or by reserving resources to different parts of the algorithm, such as different solvers [10]. We however are concerned with a particular set-up in which balance of load is asserted at a larger scale by an approach that is offered by the sparse grid combination technique. We estimate the runtime of single grids beforehand, and assign them to process groups from "longest" to "shortest", filling up idle times, cf. Fig. 2. At the same time, the black box solver may use additional load balancing internally. To the best of our knowledge, modeling the runtime of simulation time steps via machine learning techniques has not been implemented for the combination technique before.

The problem considered here is closely connected to basic scheduling algorithms. Let us assume, however, that re-assignment of a task to another process group is a costly operation that should be avoided. In our set-up, this is based on the fact that, in addition to the data on the grid, large amounts of internal simulation data are required per task – e.g., the GENE gyro matrix. If a task were to be moved, the internal data would have to be explicitly transferred or recomputed on the target process group, and the initialization of GENE can take over one hundred regular time step lengths. This is in addition to the

time required to transfer the current grid (which for high resolutions may contain a large amount of data, up to several GB).

Note that the runtime of GENE is determined by many factors which impede the use of a simple linear or performance model normal form ansatz: adding to the usual caching and communication effects, the sparsity pattern of the gyro matrix, an update algorithm with access complexity in $\mathcal{O}(n_x^2)$, and the fast fourier transform applied in y direction influence the run time heavily. But since GENE is run on many different machines, runtime data may be collected across many different machines and physical problems, in order to obtain a transferable load model at no additional computational cost.

3.1. Data Acquisition

Data was collected on the Hazel Hen supercomputer, a Cray XC40 system with Intel Haswell processors, two sockets per node, each at 12 cores. GENE was started with parameters for a scenario with adiabatic electrons at different discretizations, i.e., the level vector $\vec{\ell}$ that lives in the gyrokinetic dimensions $x, y, z, v_\parallel, \mu$. Samples were randomly selected between $\vec{\ell}_{min} = [6,4,3,4,3]$ and $\vec{\ell}_{max} = [14,8,6,8,6]$, leaving out grids that are too small to represent the underlying physics, and also those that would be larger than 2^{29} degrees of freedom in total. This was done for different levels of parallelization, for power-of-two node counts from 2^5 to 2^{15}. We will denote the processor count for each sample by 2^p. For 2^{15}, only about half as many samples were taken as for the other 2^p, since this data was quite costly to obtain. Note that previous work [4] considered grid sizes small enough to fit on $2^5 = 32$ cores on Hazel Hen, such that the higher degrees of parallelization were not a matter of discussion then.

As a simplification, the parallelization was constrained to a specific strategy depending on the level vector $\vec{\ell}$. Domain knowledge by the GENE developers is leveraged to always set a close-to-optimal parallelization: parallelize from the outer to the inner loops, i.e., first in μ direction, then in $v_\parallel$, and so on. This approach was validated on a small subset of the space (31 samples) where exhaustive parallelization tests were run: comparing the optimal runtime to the runtime obtained by way of this heuristic gave an average runtime penalty of $\approx 15\%$. The prediction of optimal parallelizations was beyond the scope of this work but would be interesting for future work into data-driven methods for GENE.

To summarize, our inputs x are the level vector $\vec{\ell}$ and the degree of parallelization p

$$x_i = (\ell_{x,i}, \ell_{y,i}, \ell_{z,i}, \ell_{v_\parallel,i}, \ell_{\mu,i}, p_i), \tag{4}$$

which means that sample i had a resolution of $2^{\ell_{x,i}}$ grid points in x direction, and was run on 2^{p_i} processes. On these samples x_i, GENE was run to find the resulting runtime t^* (averaged over 20 time steps). Out of the 2048 randomly sampled tasks, 1837 fit into the main memory of the assigned processes. They constitute our training/validation data set (80% or 1470 samples) and the test set, on which the comparisons in Section 4 will be based. The corresponding outputs – the actual wall clock times – are distributed unevenly: the mean is at 0.957, while the median runtime is only 0.268. The long-tail shape of the distribution is similar for the whole data set and the separate parallelizations, as well as the test schemes discussed in Section 4.1.

In the following, we investigate how the runtime can be estimated based on this data.

3.2. Data-Driven Methods for Load Balancing

For our purposes, *data driven* means that, apart from the input data, no further domain knowledge about the physical or computational properties of the tasks need to be known – they should be represented by the learned model that we generate from data.

We first predict the runtimes of the tasks. Based on these estimates, a descendingly ordered list is created, and the corresponding tasks are distributed to the process groups accordingly. Note that a good ordering is more important than an accurate prediction of the actual runtimes. Heene et al. [4] discussed the differences between static and (initial) dynamic load balancing. Whereas in static load balancing, the full task assignment is given at the beginning of the simulation, the dynamic variant will wait for the currently running task to finish before assigning the next in a work-stealing fashion. This dynamic assignment is done for the first time step only. Here, we will focus on the dynamic variant. We start by discussing the anisotropy-based model currently in use for the application. It is then compared to three different machine learning approaches.

3.2.1. Anisotropy-based Fits for Runtime Estimation based on Expert Knowledge

Following up on previous work by Heene et al. [4], we tested model-based fits to predict the runtime of tasks based on the resolutions in the different directions. The dependence on the overall number of points $r(N)$ is modeled based on expert knowledge by an exponential fit for each degree of parallelization p individually. These estimations are then enriched with another least-squares fit h on the anisotropy s of the level vector $\vec{\ell}$

$$r(N) := mN^k + c, \qquad h(\vec{s}_{\vec{\ell}}) = c + \sum_{i=1}^{d-1} c_i s_{\vec{\ell},i}, \qquad \vec{s}_{\vec{\ell},i} = \frac{\vec{\ell}_i}{|\vec{\ell}|_1} \tag{5}$$

to give the overall runtime estimate

$$t(N, \vec{s}_{\vec{\ell}}) = r(N) \cdot h(\vec{s}_{\vec{\ell}}). \tag{6}$$

The model is based on the observation that higher resolutions in some directions lead to substantially higher runtime and memory footprints. This least-squares fit on the runtime thus constitutes an expert knowledge-based baseline against which we can compare.

3.2.2. Nearest Neighbor

The nearest neighbor estimator stores all the data in the training data set. To predict the runtime, it takes the features of the test data and returns the value of the closest training point (in Euclidean norm). If it is queried for a data point that has the same distance to multiple known points, it will randomly return any of the neighbors' values.

We can think of the nearest neighbor estimation as "zero-th order extrapolation", the best guess we can make without actually doing any computation on the data.

3.2.3. Support Vector Regression

Support vector regression (SVR) is an application of support vector machines to regression problems [11]. The regressor is defined by learned weights w, which are determined by minimizing $\|w\| = \langle w, w \rangle$ subject to linear constraints. The constraints are designed to make sure that predictions which fall within an error of ε of the true value will not contribute to the loss; the inner product $\langle \cdot, \cdot \rangle$ is approximated by kernel functions k using a regularization parameter C [11]. For our tests, the (Gaussian) radial basis function kernel $\exp(-\gamma \|\vec{x}_i - \vec{x}_j\|^2)$ was employed. The SVR parameters were optimized by a grid search algorithm employing five-fold cross validation. Results are shown in Table 1.

For both SVR and Nearest Neighbor the `scikit-learn` Python library [12] was used. It was also used to perform standard feature scaling on the input data x for the SVR and the neural network regression, which the next section is going to discuss.

Regularization / error weight C	Soft margin width ε	RBF kernel "pointiness" γ
450	0.01	0.05

Table 1. Optimal SVR parameters on our training / validation data set

3.2.4. Neural Network / Multi-Layer Perceptron Regression

In a feed-forward artificial neural network (ANN), a function is modelled by matrix and bias vector weights, which transform the input linearly, followed by the application of a (usually) nonlinear activation function ϕ. This is done successively, layer by layer, to return the output, which is the modelled value. The transition of data y from layer l to layer $l+1$ may be described by the matrix and vector weights w and b as

$$y_{l+1,j} = \phi\left(b_{l+1,j} + \sum_{k}^{n_k} w_{l+1,k,j} \cdot y_{l+1,k}\right), \qquad \vec{y_0} = \vec{x} \tag{7}$$

where n_k is the width of layer l. The training of the network – i.e., fitting the weights – is done by the backpropagation algorithm, using stochastic optimization heuristics. For a thorough description of neural networks, please refer to [13].

Hyperparameter Fitting by Genetic Algorithms We used the `Tensorflow` package [14] to learn the training data set with five-fold cross-validation, using the robust Huber regression loss function, cf. [13]. The training was run for 100 epochs, with no batch processing. Since choosing the optimal hyperparameters for ANNs by hand is notoriously difficult, genetic techniques to select the network architecture have been shown to work in many settings [15,16]. Here, genetic selection was applied on layer depth $(1-9)$, layer width $(1-11)$, activation function, and optimizer. The fitness was the negative validation root-mean-square error (RMSE). To prevent overfitting, only those configurations that had a maximum of 700 weights to be adjusted were considered, which amounts to about half the size of the training / validation data set. The results are shown in Table 2.

# hidden layers	# nodes / layer	activation function	optimizer
6	7	`tf.nn.elu`	`tf.keras.optimizers.Adam`

Table 2. Optimal neural network parameters on our training / validation data set

4. Results

The standard machine learning metric – error on previously-unseen test data ("zero-shot test") – returned the results shown in Table 3. The neural network was randomly initialized before training. As additional set-up, one out of five trained networks was selected by lowest validation error; only this one was used to capture the error on the test sets in Section 4.1. We can see by comparing the two rightmost bars that this is not cherry-picking good random results but that the test errors are low on every trained network.

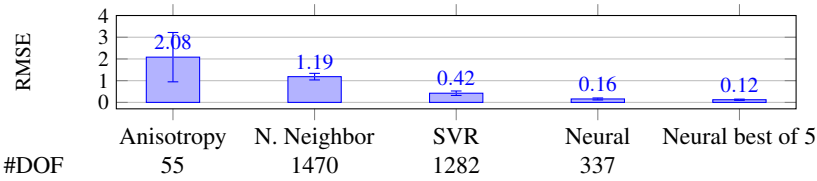

Table 3. Test errors (RMSE), based on 64 different train / test splits on the data

We observe that the estimation accuracy on the runtimes is seemingly quite good for the learning algorithms. But note that in comparison to the output distribution discussed in Section 3.1, which has a median runtime of 0.268, the estimation errors are still relatively high. Also, the estimation accuracy of the expected runtime is only suitable for evaluation to some extent. After all, a runtime estimate may be arbitrarily bad, but still help us achieve optimal scheduling if the relative ordering between the tasks is correctly represented. We will see in the next section that this leads to different outcomes if the models are applied to an actual combination scheme data set.

4.1. Results on Full Scenarios

Let us consider a test scenario, which is an actual standard combination scheme at physically relevant scales: The scheme consists of 124 grids, with level vectors between $[7,4,3,4,3]$ and $[11,8,6,8,6]$; we are approximating a full grid with 2^{39} unknowns with grids between 2^{21} and 2^{25} unknowns. All of the grids could be processed on 256 processors ($p = 8$) respectively. We also conducted a larger test case which could only be executed on bigger process groups, yielding similar results.

The resulting graph, Fig. 3, shows the parallel efficiency of the task assignment obtained depending on which trained model is used, how large the process groups are chosen (p), and how many of them there are. To make this more visually graspable, they are also averaged by method, displayed on the bottom.

We see that there are overlaps, but also a clear tendency: the neural network predicts the ordering often nearly-optimal, followed by the anisotropy-based model. The nearest-neighbor heuristic and the SVR still return reasonable results, considering we mostly get parallel efficiencies above 80%. The fact that they are lower for SVR than for the other approaches may be due to the already moderate number of input dimensions.

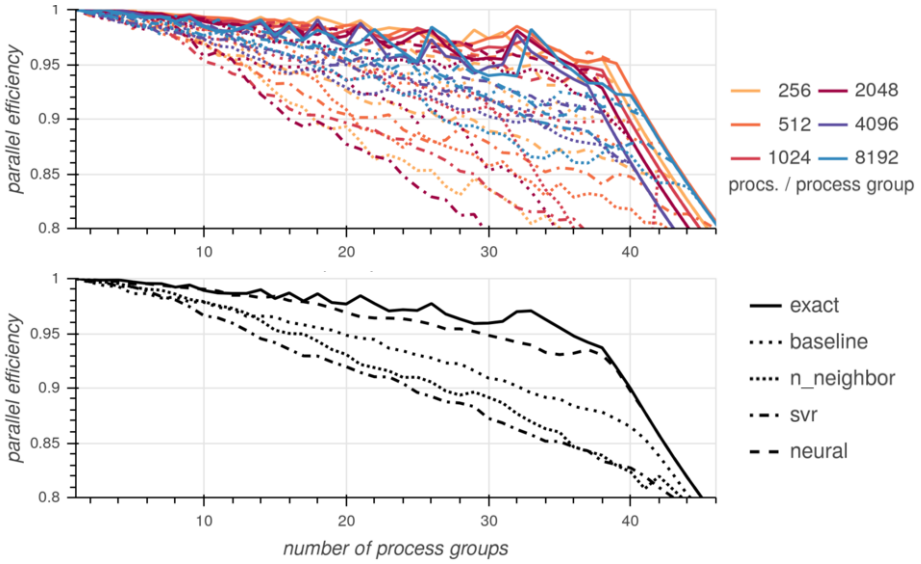

Figure 3. Parallel efficiencies for the test scenario. Note that data is included for $p = 8$ to 13.

Note that this experiment was done analytically, by adding up the exact runtimes off-line; actually running all these simulations would have been too costly. Accordingly, the true memory requirements are not represented here, but it is safe to assume that

1. the curves for low levels of parallelization will only be valid for the higher numbers of processes (as otherwise there will not be enough total memory available), and
2. high memory footprints strongly correlate with high runtimes, such that balancing runtimes will also balance memory usage to some extent.

We can observe that loads can be estimated well by using data-driven techniques for load balancing, despite the rather data-scarce setting. Furthermore, the data-driven approaches outperform the baseline based on expert knowledge not only in regions with plenty of data points, but also in those parameter regions where extrapolation dominates: for large numbers of processes. Still, the data-driven approaches excel only if enough data is at hand. It is therefore essential to keep track of GENE runtimes and the simulation parameters and metadata, such as system architecture and GENE version.

This could potentially pay off even more when using process groups of different sizes [17], which could be an interesting subject of study in the future.

5. Conclusion

In this paper, we studied the data-based prediction of runtimes for load balancing. This enabled us to obtain good load balances for the massively parallel sparse grid combination technique with GENE. While the expert knowledge-based model used until now is reasonable, it can be outperformed by purely data-driven methods such as neural networks, given enough data and automated selection of network parameters.

Based on this insight, it is now feasible to collect run time data for GENE simulations on the job, improving data-driven load models along the way. Especially with respect

to different resolutions, at least knowledge about a good ordering between tasks should be possible even across compute systems. Let us note that data-driven approaches for load balancing are most suited for situations where the concrete code is considered a black-box. If, however, the behavior of the compute and communication systems, as well as the algorithm and its implementation are well-understood and stable, it will be more beneficial to use model-based approaches. In all other cases, one should use carefully selected data-driven approaches, especially when a lot of compute time is at stake – such as with the massively parallel sparse grid combination technique employed with `GENE`.

Acknowledgements We would like to thank Raphael Leiteritz and Tilman Dannert for suggestions and fruitful discussions. This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

[1] F. Jenko, D. Told, T. Görler, et al. Global and local gyrokinetic simulations of high-performance discharges in view of ITER. *Nucl. Fusion*, 53(7):073003, 2013.

[2] M. Heene, A. P. Hinojosa, M. Obersteiner, H.-J. Bungartz, and D. Pflüger. EXAHD: An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *High Performance Computing in Science and Engineering '17*, pages 513–529. Springer, 2018.

[3] M. Heene. *A Massively Parallel Combination Technique for the Solution of High-Dimensional PDEs*. PhD thesis, Universität Stuttgart, 2018.

[4] M. Heene, C. Kowitz, and D. Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 574–583, 2014.

[5] T. Görler. *Multiscale effects in plasma microturbulence*. PhD thesis, Universität Ulm, 2009.

[6] H. Doerk and F. Jenko. Towards optimal explicit time-stepping schemes for the gyrokinetic equations. *Computer Physics Communications*, 185(7):1938–1946, 2014.

[7] M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. The combination technique for parallel sparse-grid-preconditioning or -solution of PDEs on workstation networks. In *Parallel Processing: CONPAR 92 VAPP V*, 1992.

[8] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[9] S. Hirschmann, C. W. Glass, and D. Pflüger. Enabling unstructured domain decompositions for inhomogeneous short-range molecular dynamics in ESPResSo. *The European Physical Journal Special Topics*, 227(14):1779–1788, 2019.

[10] A. Totounferoush, N. Ebrahimi Pour, J. Schröder, S. Roller, and M. Mehl. A new load balancing approach for coupled multi-physics simulations. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019.

[11] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[12] F. Pedregosa, G. Varoquaux, et al. Scikit-learn: Machine Learning in Python. *JMLR*, 12:2825–2830, 2011.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer-Verlag, 2 edition, 2009.

[14] M. Abadi, A. Agarwal, P. Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. URL: `https://www.tensorflow.org/`, 2015.

[15] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37, 199206.

[16] R. Miikkulainen, J. Liang, E. Meyerson, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Academic Press, 2019.

[17] M. Molzer. Implementation of a parallel sparse grid combination technique for variable process group sizes. Bachelor's thesis, TU München, 2018.