Code Generation for Massively Parallel Phase-Field Simulations

Martin Bauer
martin.bauer@fau.de
Chair for System Simulation
Friedrich-Alexander Universität
Erlangen-Nürnberg

Johannes Hötzer*
Institute for Digital Materials
University of Applied Science
Karlsruhe

Dominik Ernst
Professorship for High Performance
Computing & HPC group of RRZE
Friedrich-Alexander Universität
Erlangen-Nürnberg

Julian Hammer
Professorship for High Performance
Computing & HPC group of RRZE
Friedrich-Alexander Universität
Erlangen-Nürnberg

Marco Seiz
Institute for Applied Materials
Karlsruhe Institute of Technology

Henrik Hierl
Institute for Digital Materials
University of Applied Science
Karlsruhe

Jan Hönig
Chair for System Simulation
Friedrich-Alexander Universität
Erlangen-Nürnberg

Harald Köstler
Chair for System Simulation
Friedrich-Alexander Universität
Erlangen-Nürnberg

Gerhard Wellein
Professorship for High Performance
Computing & HPC group of RRZE
Friedrich-Alexander Universität
Erlangen-Nürnberg

Britta Nestler†
Institute for Applied Materials
Karlsruhe Institute of Technology

Ulrich Rüde‡
Chair for System Simulation
Friedrich-Alexander Universität
Erlangen-Nürnberg

ABSTRACT

This article describes the development of automatic program generation technology to create scalable phase-field methods for material science applications. To simulate the formation of microstructures in metal alloys, we employ an advanced, thermodynamically consistent phase-field method. A state-of-the-art large-scale implementation of this model requires extensive, time-consuming, manual code optimization to achieve unprecedented fine mesh resolution. Our new approach starts with an abstract description based on free-energy functionals which is formally transformed into a continuous PDE and discretized automatically to obtain a stencil-based time-stepping scheme. Subsequently, an automatized performance engineering process generates highly optimized, performance-portable code for CPUs and GPUs. We demonstrate the efficiency for real-world simulations on large-scale GPU-based (PizDaint) and CPU-based (SuperMUC-NG) supercomputers. Our technique simplifies program development and optimization for a wide class of models. We further outperform existing, manually optimized implementations as our code can be generated specifically for each phase-field model and hardware configuration.

CCS CONCEPTS

• **Software and its engineering → Massively parallel systems**; **Domain specific languages**; • **Mathematics of computing →** *Partial differential equations*;

ACM Reference Format:
Martin Bauer, Johannes Hötzer, Dominik Ernst, Julian Hammer, Marco Seiz, Henrik Hierl, Jan Hönig, Harald Köstler, Gerhard Wellein, Britta Nestler, and Ulrich Rüde. 2019. Code Generation for Massively Parallel Phase-Field Simulations. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356186

1 INTRODUCTION

Whatever kind of metal you see in your everyday life, it has already been molten and solidified. The solidification conditions determine the inner structure of the material at the micrometer scale. This microstructure determines many properties of the material from its strength to the acoustical and optical properties. In order to develop materials with precisely controlled properties, a deep understanding of the complex, time-dependent, three-dimensional process is needed. To reduce the time and resources spent on developing optimized and new materials, simulations have become a powerful tool in the last decades. Besides molecular dynamics and Monte Carlo simulations at the atomistic scale and finite element simulations at

*Also with Karlsruhe Institute of Technology.
†Also with University of Applied Science Karlsruhe.
‡Also with CERFACS.

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356186

the macroscopic scale, the phase-field method has been established to investigate the solidification process at the mesoscopic scale. Over the last decades, it has been extended by coupling it to models of various other physical phenomena, like fluid flow, mechanical forces, magnetism, chemical reactions, and electrical fields. This leads to a wide range of different models for the investigation of various effects in materials science. In many cases, however, resolving the physics requires representative volume elements that may have billions or more grid points [1]. In order for the simulation to exhibit physically relevant behavior, additionally, long integration times may be necessary. These two aspects create the need for highly parallelized and run-time-efficient solvers on HPC systems.

However, intricate code optimization is in conflict with the flexibility needed for developing new models. Here flexible and extensible frameworks are needed where different models can be implemented and studied easily. To reach both goals, i.e., highest possible performance while maintaining maximal flexibility, we here develop a code generator. It gives the users a powerful and flexible modeling tool while additionally providing techniques to generate highly efficient code for different HPC architectures automatically.

The highest abstraction level to concisely formulate phase-field models is the energy functional

$$\Psi(\boldsymbol{\phi}, \boldsymbol{\mu}, T, \ldots) = \int_V \psi_{int}(\boldsymbol{\phi}) + \psi_{dr}(\boldsymbol{\phi}, \boldsymbol{\mu}, T, \ldots)\, dV, \qquad (1)$$

where $\boldsymbol{\phi}$ describes the vector of local volume fractions of different phases and their orientation. The individual ϕ_α are called phase-field values, $\boldsymbol{\mu}$ is the chemical potential of the components, and T is the temperature. Further terms, e.g., for fluid flow or electromagnetic interactions, can also be included. Nature minimizes this functional on its own; mathematically, this can be represented by variational derivatives

$$\frac{\partial \phi_\alpha}{\partial t} \sim \frac{\delta \Psi}{\delta \phi_\alpha}, \qquad \frac{\partial \mu_i}{\partial t} \sim \frac{\delta \Psi}{\delta \mu_i}, \qquad \frac{\partial T}{\partial t} \sim \frac{\delta \Psi}{\delta T}. \qquad (2)$$

The variational derivative leads to a set of partial differential equations (PDEs) that must be solved by compute-intensive numerical algorithms. Even for the same physical process, there are many variations of the terms in (1) that may be changed to study different effects. For example, solidification can include both dendritic and eutectic structures. The first requires anisotropic terms whereas the latter can be modeled isotropically. However, a generic simulation software typically requires general algorithms. Thus the software in use will often be suboptimal since it can neither be developed to exploit the special model features nor can it be easily optimized for the HPC hardware configuration in use.

In previous work [2], we presented phase-field models for large representative volume elements. The previous general-purpose C implementation was specialized and manually optimized for one particular scenario. In particular, we demonstrated that a huge speedup can be obtained by a meticulous specialization of the code to the model, the geometry, and parameter region. For example, by exploiting the special functional form of the temperature, compute-intensive subexpressions could be pre-computed. Overall these steps have resulted in a speedup of 80 as compared to the original general purpose code. Note that this was achieved by hand-optimizing the solver specifically for the special phase-field model and the specific supercomputer architecture.

Also in many other cases, extremely large scale simulation runs are necessary to obtain physically relevant results. One can then argue that the high cost for manual code optimization can be quickly amortized by the reduced computational cost. However, even if economically justified, this may still be unsatisfactory. As soon as the application scientist wishes to alter the model in any nontrivial form, almost the complete manual optimization process must be repeated. In practice, this leads to a situation where a flexible, but slow implementation is used for model development and for performing small scale simulations. As soon as the application scientist is then satisfied with the model and its parameterization, large scale simulations have to be conducted. Therefore the specific instance is handed to the computer scientist for optimization and tuning to the hardware of the HPC system, e.g., by developing a CUDA version of the code. This binds precious human resources and slows down the scientific development cycle.

2 STATE OF THE ART / RELATED WORK

The idea of the phase-field method to treat the interface between two phases as diffuse area dates back to van der Waals [3] in the 1870s. The Ginzburg-Landau theory [4] extended this idea by introducing an order-parameter. Allen and Cahn [5], as well as Cahn and Hilliard [6], used the Ginzburg-Landau functional to derive their PDEs. Based on these ideas, many solidification models were developed as summarized in [7–9]. In these, the wide range of available phase-field models for solidification coupled with various physical phenomena is described. Large-scale application of one of these models was shown by the ACM Gordon Bell winners Shimokawabe et al. [10]: They showed the first petascale phase-field simulation of dendritic solidification in a $768 \times 1\,632 \times 3\,264$ voxel cells domain using 1 156 GPUs of the TSUBAME 2.0 supercomputer. Two years later, competitive dendritic directional solidification in a $4\,096 \times 4\,104 \times 4\,096$ voxel cell domain using 768 GPUs for 12 days was shown by Takaki et al. [1]. Bauer et al. [2] presented a highly optimized phase-field solver to investigate the directional solidification of a ternary eutectic alloy. Through the optimizations, a peak performance of 25 % on the node level was achieved. Also, the solver was scaled on different hardware architectures with up to 262 144 cores using 1 048 576 MPI processes on the Blue Gene/Q machine JUQUEEN [2]. Based on this solver, the directional solidification in domains with $2\,420 \times 2\,420 \times 1\,474$ voxel cell using 84 700 CPUs for 7 h [11] and $4\,116 \times 4\,088 \times 4\,325$ voxel cell using 171 696 CPUs for 6 h [12] were investigated. The coarsening dynamics in a $6\,700 \times 6\,700 \times 6\,700$ voxel cells domain is reported by Zhang et al. [13] using 10 235 160 cores. On the Sunway TaihuLight, they reached a peak performance of 50 579 TFlops on 10.6 million cores. All the above simulations were done by hand-optimizing specific applications. This is in stark contrast to the goal of combining flexibility with performance as outlined in the introduction. One approach to reach this goal is generating the code on-the-fly from an application scientist's description.

The idea of automatically transforming a high-level problem description into optimized code is already successfully applied

in many fields, e.g., for finite element methods [14], neural networks [15], multigrid methods [16], CFD simulations [17–19], astrophysics [20], or in computational chemistry [21]. There also exist many stencil compilers that automatize loop transformations like spatial and temporal blocking [22, 23]. However, most of them only provide the "lower" part of the necessary toolchain, i.e., from stencil representation to code.

In this article, we extend these concepts in several ways. In particular, we include the application-specific physical modeling into the automatic generation process. Material scientists can now develop new solidification models in terms of energy functionals. The systematic, but tedious derivation of the resulting partial differential equations is then performed automatically. In this sense, our code generation framework is a co-design effort that builds on decades of research leading to extensive experience and best practice know-how specifically for phase field models for complex solidification simulations.

Our approach of an embedded domain-specific language (DSL) in Python constitutes a flexible extensions of the code generation process. While there exist also Python-based stencil code generation packages [24–26], they are restricted in their inter-node parallelization capabilities, i.e., they lack dynamic load balancing and functionality for handling complex geometries. Our modular approach allows for easy integration of the generated compute kernels into existing frameworks to handle complex domain shapes, dynamic load balancing, in-situ analysis and efficient I/O capabilities at large scales.

3 ABSTRACTION LAYERS

Our code generation pipeline is organized into several abstraction layers (Fig. 1). Starting from the top, the first two layers allow the application scientist to formulate the model using a continuous mathematical notation. Automatic discretization of the continuous evolution equations gives a stencil formulation of the problem. At the intermediate representation layer, an algorithmic description is used and loop transformations can be applied. Finally, backends generate performance-portable C or CUDA code. A detailed description is beyond the scope and the space restrictions of this article and thus we refer to documentation for waLBerla[1] and the open source repository[2] on github that includes the technical documentation of the full code generation pipeline.

3.1 Energy functional layer

Phase-field models are based on an energy functional as given in (1). Time evolution equations can be obtained by computing variational derivatives of this functional, resulting in multiple coupled PDEs. Our code generation pipeline enables the user to facilitate this concise, elegant formalism to define the phase-field model. In this work the functional is chosen following [11, 27] as

$$\Psi(\boldsymbol{\phi}, \boldsymbol{\mu}, T) = \int_V \left(\epsilon a(\boldsymbol{\phi}, \nabla\boldsymbol{\phi}) + \frac{1}{\epsilon}\omega(\boldsymbol{\phi}) \right) + \psi(\boldsymbol{\phi}, \boldsymbol{\mu}, T)\, dV \qquad (3)$$

[1]www.walberla.net
[2]https://github.com/mabau/pystencils/

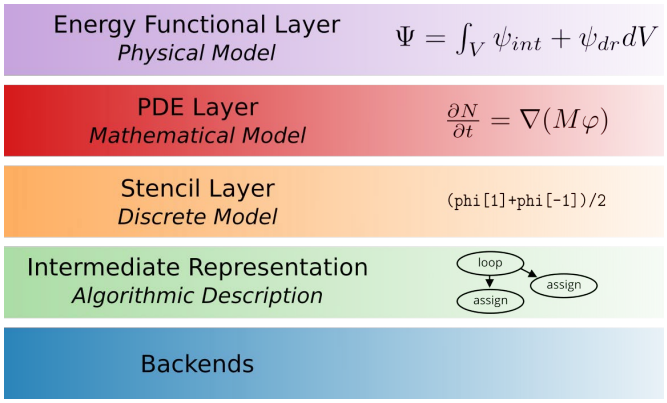

Figure 1: Abstraction layers of code generation tool

with a gradient energy density a, obstacle potential ω, and driving force ψ. The gradient energy density is

$$a(\boldsymbol{\phi}, \nabla\boldsymbol{\phi}) = \sum_{\alpha<\beta}^{N} \gamma_{\alpha\beta} A_{\alpha\beta}(\mathbf{R}\, q_{\alpha\beta})^2 \left| q_{\alpha\beta} \right|^2 \qquad (4)$$

with the interface energy $\gamma_{\alpha\beta}$, the generalized gradient $q_{\alpha\beta} = \phi_\alpha \nabla\phi_\beta - \phi_\beta \nabla\phi_\alpha$ and the anisotropic term $A_{\alpha\beta}$ depending on the generalized gradient rotated by the unitary matrix $\mathbf{R}$. The obstacle potential is defined as

$$\omega(\boldsymbol{\phi}) = \frac{16}{\pi^2} \sum_{\substack{\alpha,\beta=1 \\ (\alpha<\beta)}}^{N,N} \gamma_{\alpha\beta}\phi_\alpha\phi_\beta + \sum_{\substack{\alpha,\beta,\delta=1 \\ (\alpha<\beta<\delta)}}^{N,N,N} \gamma_{\alpha\beta\delta}\phi_\alpha\phi_\beta\phi_\delta \qquad (5)$$

with a higher order term $\gamma_{\alpha\beta\delta}$ to suppress spurious third-phases. These two terms are sufficient to describe simple problems such as mean curvature flow. In order to include additional effects, a driving force term is required. For the case of solidification, the driving force is the difference of grand potentials, making the grand potential density ψ the natural choice. This potential can be calculated via thermodynamic CALPHAD databases, which use computationally expensive formulations. Instead of calling these databases, parabolic fits of the form

$$\psi_\alpha(\boldsymbol{\mu}, T) = \boldsymbol{\mu} \cdot \boldsymbol{A}(T)\boldsymbol{\mu} + \boldsymbol{B}(T) \cdot \boldsymbol{\mu} + C(T) \qquad (6)$$

are employed for each phase α. It is assumed that $\boldsymbol{A}$, $\boldsymbol{B}$, C are affine linear in T. With this, the grand potential density can be defined as $\psi(\boldsymbol{\phi}, \boldsymbol{\mu}, T) = \sum_{\alpha=0}^{N} \psi_\alpha(\boldsymbol{\mu}, T) h_\alpha(\boldsymbol{\phi})$, with the interpolation function h_α having zero gradient at $\phi_\alpha \in \{0, 1\}$ and mapping 0 to 0 as well as 1 to 1. The highest abstraction layer of our code generation system directly employs this mathematical description to formulate the model in a Python-embedded DSL, based on the computer algebra system *sympy* [28]. We extend *sympy* by special constructs to represent differential operators that support variational derivatives. Above description of the energy functional can be written as

```
φ, μ = fields("phi, mu: double[3D]")
a = sum( γ[α,β] * (φ[α]*grad(φ[β]) - φ[β]*grad(φ[α]))**2
         for β in range(N) for α in range(β) )
# (...) similarly ω and driving_force are defined
energy_density = a + ω + driving_force
```

3.2 PDE Layer

On the next lower abstraction layer, the governing PDEs of the system are written down or derived from the energy formulation. The phase variables $\boldsymbol{\phi}$ evolve according to the Allen-Cahn equation

$$\tau\epsilon\frac{\partial\phi_\alpha}{\partial t} = \frac{\delta\Psi}{\delta\phi_\alpha} + \Lambda + \xi(\boldsymbol{\phi}). \tag{7}$$

The left hand side describes the time evolution with the interfacial relaxation coefficient τ and the interface scaling parameter ϵ. On the right hand side, the variational derivative of the functional is corrected by a Lagrange multiplier $\Lambda = \frac{1}{N}\sum_{\alpha=0}^{N}\frac{\delta\Psi}{\delta\phi_\alpha}$. Finally, a fluctuating term can be added to promote the growth of side branches in dendritic growth. On the PDE layer, this can be written down as

```
dΨ_dφ = [func_diff(energy_density, φ[α]) for α in range(N)]
τ_ip = interpolate(τ, phases=φ)
fluctuation = amplitude * random(-1, 1, kind='philox')
φ_pdes = [- τ_ip * ε * transient( φ[α] ) + dΨ_dφ[α] \
          + sum(dΨ_dφ) / N + fluctuation
         for α in range(N)]
```

Note that variational derivatives can be automatically computed by the code generation system. As shown in the code example, the local kinetic coefficient τ is not chosen constant but computed from local phase-field values and pairwise kinetic coefficients $\tau_{\alpha\beta}$. While lengthy expressions as the variational derivatives are automatically derived from the abstraction layer above, the user has the full flexibility to adjust the model on lower layers as well. One example of this flexibility is the extension of the model by a fluctuation term by adding a single expression to the PDE.

For the model considered here, the driving force depends on chemical potential. While it is possible to use the variational approach for the $\boldsymbol{\mu}$ evolution equation, it has been shown by Karma [29] that a non-variational approach provides higher computational efficiency. Hence we construct the $\boldsymbol{\mu}$ evolution equation directly on the PDE level:

$$\begin{aligned}\frac{\partial\boldsymbol{\mu}}{\partial t} = \left[\left(\frac{\partial\boldsymbol{c}}{\partial\boldsymbol{\mu}}\right)\right]^{-1}\Bigg(&\nabla\cdot\Big(\boldsymbol{M}(\boldsymbol{\phi},\boldsymbol{\mu},T)\nabla\boldsymbol{\mu} - \boldsymbol{J}_{at}(\boldsymbol{\phi},\boldsymbol{\mu},T)\Big)\\ &- \left(\frac{\partial\boldsymbol{c}}{\partial\boldsymbol{\phi}}_{T,\boldsymbol{\mu}}\right)\frac{\partial\varphi}{\partial t} - \left(\frac{\partial\boldsymbol{c}}{\partial T}_{\boldsymbol{\phi},\boldsymbol{\mu}}\right)\frac{\partial T}{\partial t}\Bigg).\end{aligned} \tag{8}$$

The derivatives are based on thermodynamic functions and are calculated automatically by *sympy* as soon as the functional dependence of $\boldsymbol{c}$ on $\boldsymbol{\mu}$ is defined. The divergence term consists of fluxes due to a chemical potential gradient corrected by a so-called anti-trapping current $\boldsymbol{J}_{at}$. The mobility $\boldsymbol{M}$ depending on the diffusion coefficient $\boldsymbol{D}_\alpha$ is defined as:

$$\boldsymbol{M}(\boldsymbol{\phi},\boldsymbol{\mu},T) = \sum_{\alpha=1}^{N}\boldsymbol{D}_\alpha\frac{\partial\boldsymbol{c}_\alpha(\boldsymbol{\mu},T)}{\partial\boldsymbol{\mu}}g_\alpha(\boldsymbol{\phi}). \tag{9}$$

Herein, the main difference to a variational approach is revealed: The mobility is not interpolated with h_α, but rather with a simpler interpolation function g_α. Finally, the anti-trapping current is defined as [27]:

$$\begin{aligned}\boldsymbol{J}_{at} = \frac{\pi\epsilon}{4}\sum_{\substack{\alpha=1,\\ \alpha\neq l}}^{N}\frac{g_\alpha(\boldsymbol{\phi})h_l(\boldsymbol{\phi})}{\sqrt{\phi_\alpha\phi_l}}\frac{\partial\phi_\alpha}{\partial t}\left(\frac{\nabla\phi_\alpha}{|\nabla\phi_\alpha|}\cdot\frac{\nabla\phi_\ell}{|\nabla\phi_\ell|}\right)\\ \left(\left(\boldsymbol{c}^l(\boldsymbol{\mu}) - \boldsymbol{c}^\alpha(\boldsymbol{\mu})\right)\otimes\frac{\nabla\phi_\alpha}{|\nabla\phi_\alpha|}\right).\end{aligned} \tag{10}$$

Being able to use a high-level functional description allows the user to define complex models in a flexible manner. We have deliberately not chosen an external DSL but use an embedded DSL in Python. This approach gives the user full flexibility to introduce their own abstractions that might be useful for the specific problem at hand. This means that, for example, also the parameterization of the model can be automated. Parameters like γ or grand potential density coefficients can be obtained from a database and adapted to the model, making use of the vast ecosystem of scientific packages in Python.

3.3 Discretization Layer

At this stage we have a set of PDEs represented as expression trees with continuous differential operators acting on symbolic fields. The discretization layer automatically transforms these PDEs into a stencil representation applying finite differences for spatial derivatives and an explicit time stepping scheme.

Our tool provides functionality to automatically derive finite difference stencils of a given order. For the specific class of phase-field methods that are currently used in our application domain, we can use standard second-order finite differences. In this setting it is possible to approximate first order derivatives by central differences, but it is essential that second derivatives are split in a "divergence-of-fluxes" representation, such that the fluxes can be evaluated at staggered positions. To illustrate this, consider the example term $\partial_x\left(p(x)\,\partial_x f + \partial_y f\right)$, where $p(x)$ is an analytical expression depending on spatial coordinates, f is a field, and ∂_x, ∂_y denote spatial derivatives. This expression could be part of an automatically derived PDE or can be explicitly written as

```
p = function("p")(x)
f = fields("f: double[2D]")
pde_rhs = diff(p * diff(f, 0) + diff(f, 1), 0)
discretization_strategy = fd.Discretization(order=2)
stencil = discretization_strategy(pde_rhs)
```

The expression in the outermost derivative is first evaluated at left and right staggered positions S_l and S_r to yield the final value $(S_r - S_l)/dx$. Quantities not available at staggered positions are automatically interpolated. The full value of S_r is thus computed as

$$\begin{aligned}S_r \leftarrow p(x+dx/2)\left(\frac{f_{(1,0)} - f_{(0,0)}}{dx}\right) +\\ \frac{1}{2}\left(\frac{f_{(0,1)} - f_{(0,-1)}}{2dy} + \frac{f_{(1,1)} - f_{(1,-1)}}{2dy}\right)\end{aligned} \tag{11}$$

where the subscripts denote indices into the array relative to the current cell. In a naive implementation, these staggered values would be computed twice, since e.g., the left staggered value of a cell is the right staggered value of its left neighbor. Our system can generate a separate kernel for pre-computation of these staggered values.

This discretization strategy is specific to the application field and can be seen as a best practice in the application field, see [11, 27, 30] and the references therein. Thus, currently explicit finite difference discretizations are used despite their obvious numerical limitations in terms of accuracy and time step restrictions. We note, however, that using better numerical schemes, as they may be developed in the future, will be much simpified. With the code generation approach, such improved schemes can be realised in a well identified specific code generation module. Thus any such improvement would automatically still use all the other optimizations in the workflow and would thus immediately benefit all applications using the framework.

The resulting stencil description of the problem is rewritten to further reduce the amount of necessary floating point operations per cell update. Terms are simplified individually by expansion or factoring. At this stage, the symbolic parameters which remain fixed during a simulation run are substituted by numeric values. This constant folding step on expression level also helps subsequent simplification passes, since the total size of the expression trees is significantly reduced. After every term has been simplified individually, a global common subexpression elimination (CSE) step is done across all terms.

In the discretization layer fluctuating terms are replaced by random number generators. We use the fast counter-based random number generator (RNG) Philox [31]. This RNG is stateless, i.e., no seed state has to be loaded from memory. The global cell index and current time step are used as counters/keys such that no data dependencies between cell updates are introduced.

3.4 Intermediate representation layer

In this layer, the stencil description is transformed into an algorithmic description. The stencil representation consists of a list of assignments with instructions to be executed for every cell. These assignments contain references to arrays using relative indexing. Optionally, assignments may refer to cell indices and the current time step to express analytic dependencies on space and time. Left-hand sides of assignments may either be array writes or assignments to temporary symbols, to be used in later assignments. Assignments to temporary symbols are considered constant, thus this representation is in static single assignment (SSA) form. The SSA form simplifies the formulation of further optimizations. Any symbols not defined before, i.e., not occurring as the left-hand side of an assignment before, become arguments to the generated kernel function. Since the abstraction layers above are built on top of the computer algebra system *sympy*, symbols are not required to carry any type information. The first transformation on this layer thus ensures that all expressions are properly typed and inserts casts where necessary. Then loop nodes are built around the assignment list, representing the iteration over the full or rectangular sub-region of the domain. The memory layout of the multidimensional arrays is fixed at this stage. The loop order is determined depending on the memory layout to ensure spatial locality of memory accesses. Array accesses are resolved at this stage and rewritten as a base pointer with linear index. Base pointers and other subexpressions that are constant w.r.t. to the current iteration are pulled before loops. In combination with CSE, this step is crucial to automatically exploit special functional forms of the temperature. For example, if the temperature depends on one spatial coordinate only, the loop over this coordinate is chosen as the outermost loop and all temperature-dependent subexpressions are pulled out of the inner loops.

The finite difference discretization of second order derivatives contains evaluations of first-order derivatives at staggered positions. We implemented two options to handle these flux values. They can either be recomputed in every cell or be precomputed and cached in a temporary staggered field. The precomputation approach eliminates the necessity to compute flux values twice, however, another pass over the domain is necessary. In [2] a different buffering technique was used where flux values for only two slices of the domain have to be stored. However, in this technique cell updates are not independent anymore, making OpenMP parallelization and straightforward GPU implementations impossible. Kernels updating staggered fields have a different iteration pattern than update kernels for cell-centered values. A 2D block of size $N_x \times N_y$ with one ghost layer in all directions has $(N_x + 1) \times N_y$ values at x staggered positions but $N_x \times (N_y + 1)$ at y staggered positions. This could be solved by iterating separately over x and y staggered values. Because in both iterations usually the same or nearby values are read from memory, overall performance can be increased by fusing both iterations. Due to the difference in loop bounds, this transformation is non-trivial. To optimize CPU code, we make use of the integer set library [32] to get an optimal iteration pattern.

3.5 Backends for C and CUDA

In the final step of the code generation pipeline, our intermediate representation (IR) is transformed into C or CUDA code. We generate C/CUDA code instead of LLVM-IR [33] because according to our experience compilers developed by hardware vendors (Intel C compiler, CUDA compiler) give slightly better performance in most cases.

For CPUs, we generate an OpenMP parallel code that is also explicitly vectorized using SIMD intrinsics. This can be done without any additional analysis steps since our pipeline guarantees that loop iterations are independent and that there are no conditionals inside the loop body. However, we allow expressions containing piecewise-defined functions with a mandatory fallback case, since they can be efficiently mapped to `blend` vector instructions. Our tool supports the SSE, AVX, and AVX512 SIMD instruction sets. Vectorization is done on the intermediate representation by unrolling the loop by the vector length and generating a tear-down loop for remaining cells. Arrays are allocated and padded such that the beginning of each line is sufficiently aligned. Thus aligned reads and writes can be issued for all array accesses that have no offset in the "fastest" coordinate, i.e., the coordinate that is stored consecutively in memory. We choose to explicitly vectorize with intrinsics instead of relying on the auto-vectorization of the compiler to have full control over the process. The user of the code generation pipeline can, for example, choose costly operations like division or square roots to be evaluated in a faster but approximate way. The backend then uses corresponding intrinsics, if available.

We use for example `rqrt14` intrinsics to approximate reciprocal square roots on AVX512 architectures.

Similarly, operations that have been marked for approximate evaluation by the user are treated differently in the CUDA backend as well. Approximate divisions are computed using the `fdividef` intrinsic function that uses 32-bit floating point values instead of the usual 64 bit double values. Approximate (inverse) square roots are also computed in single precision using `frsqrt`. Since some of the kernels contain many of these comparatively expensive operations, using approximations can be worthwile. To generate CUDA code, the backend first has to strip away loop nodes of the intermediate representation and replace loop counters by index expressions using CUDA's special variables for the thread and block indices. For the mapping of CUDA threads to domain cells several strategies are implemented. A transformation then applies the selected strategy by replacing all array accesses accordingly. This allows for a clean modular structure where the thread-to-cell mapping code is fully separated from the stencil implementation and can be exchanged easily. An auto-tuning mechanism can then test and optimize different mappings.

Furthermore, the CUDA backend implements several GPU-specific optimization passes. An important set of transformations aims to reduce the register usage of kernels. There are several factors that contribute to high register usage: The CSE finds many small common expressions, that are reused in multiple assignments, which creates many intermediates that are alive for a long time. Furthermore, the compiler tries to move as many loads as possible to the beginning of a block, so that they can overlap with each other and independent computations. Lastly, the compiler also reduces its efforts for instruction reordering for register count minimization if the number of statements becomes too large.

The first transformation to address this problem reorders the IR level assignments. This also results in reordering the statements in the generated CUDA code in a way that minimizes the number of intermediates that are alive at any point. We assume that some of this order is preserved in the internal representation of the nvcc compiler and that it will also persist in the generated machine code even after the compiler-specifc additional transformations.

We adapted a scheduling method described in [34]: A breadth-first search generates all possible instruction schedules starting from the roots of the dependency graph, but after each scheduling step deduplicates partial schedules that would have the same path forward for the rest of the instructions. While this algorithm finds the optimal schedule, the original publication from 1998 notes that graphs with more than 50 nodes are impossible but unrealistic. Because our kernels easily contain 1000s of nodes, we convert the algorithm to a heuristic by only keeping a fixed number of the so far best schedules in each step. This tunable parameter allows a balance between a greedy and full breadth-first search. The second transformation addresses the problem of many small, long-lived intermediates found by the CSE. It essentially takes back some effects of the CSE, by rematerializing expressions that are cheap to compute. Some of the tunable, considered properties of assignments are whether the used terms are at the top of the dependency graph like constants and field accesses, how many operations are in the expression and how often the computed expression is used in other expressions. While experimenting with manual spilling into shared memory, we found that marking the shared memory buffers as volatile not just ensured that data would actually be reloaded, but also reduces the amount of reordering of instructions by the compiler, in order to preserve the load/store semantics. Since this can be beneficial to avoid compiler mechanisms that have negative impact in this case, there is a transformation that inserts `__threadfence()` statements, which have the same effect. The described transformations shift the compiler generated balance of several performance mechanics that work in different directions. Different kernels require a different balance, and all transformations might have negative performance impacts on some kernels. Additionally, the effects of multiple transformations do not add up linearly but can decrease or amplify each other. To deal with this non-convex, multi-dimensional, non-smooth fitness landscape, we use an evolutionary optimization algorithm to tune a sequence of transformations with their parameters for each kernel. The optimization potentially discovers sequences that would have been elusive to reasoning and manual experiments.

3.6 Automatic Performance Modeling

Performance modeling is a valuable tool to understand and improve the performance of a compute kernel. To assist automatic optimizations in the code generation process, we use an automatic performance modeling approach. While the complexity of modern processors makes a full model of the CPU impossible, a performance model tries to capture the most essential aspects of a compute system to make reasonably accurate predictions for the behavior of a piece of software.

Automating the performance modeling process is important in our cases because some of the kernels considered here have more than 3 000 assembly instructions and 110 array references, which makes manual analysis tedious, error-prone or simply not feasible.

We use the intermediate representation of loop kernels to gather all necessary input for the execution-cache-memory (ECM) [35] model. Therefore our code generation pipeline is coupled to the analysis tool *Kerncraft* [36] that automatically constructs an ECM model for the kernel. In addition to this analytic performance model, we can also compile a benchmark executable and perform measurements of actual performance characteristics using *likwid* [37]. Performance modeling and benchmark results are then fed back as input for further optimization, e.g., to determine sizes for spatial blocking.

Since all modern mainstream architectures are still based on a von-Neumann architecture, the two main processor components that have to be modeled are the arithmetic logical unit (ALU) and the memory hierarchy, which comprises various caches and the main memory. To predict which subsystem limits the performance of a compute kernel we need at least two code characteristics: First, the *compute throughput* i.e., how many cycles the CPU needs to execute a number of loop iterations, assuming that all required data is available in the fastest cache. Secondly, the *data throughput* has to be determined, i.e., how many cycles it takes to transfer the data through the memory hierarchy. Throughput is considered in cycles per cache line of results or eight lattice site updates.

For a first estimate of the compute throughput, the number and type of floating point operations in a kernel can be counted and

combined with the throughput information for these instructions. Instruction throughput information is often provided by the CPU vendor. Floating-point operations (FLOPs) are counted by traversing the fully optimized intermediate representation, where constants are already folded and common subexpressions are extracted.

For a more accurate and automated analysis on Intel architectures, the kernel throughput can be determined using the Intel Architecture Code Analyser (IACA). This tool conducts a static analysis on the compiled kernel and uses a detailed CPU model to more accurately estimate the compute throughput, assuming all data is available in the L1 cache. For IACA to find the kernel code in the compiled executable, special assembly markers have to be inserted at the beginning and the end of the kernel loop, which is done automatically by *Kerncraft*. The modified representation is then assembled and run through IACA.

To determine the data throughput of a kernel, data structures together with number and offset of field accesses are reported to *Kerncraft*. Then two approaches can be used to determine the actual amount of data transferred between caches: analytical layer conditions or a cache hierarchy simulator. For details on these models, we refer to [36]. The current Skylake architecture poses some issues for the prediction since Intel has made the last-level cache a non-inclusive victim cache with unpublished heuristics. These unknown heuristics introduce some additional uncertainties in the performance models.

4 DISTRIBUTED MEMORY PARALLELIZATION

To run the generated kernels on distributed-memory systems we use the multi-physics HPC framework waLBerla.

4.1 waLBerla Framework

waLBerla employs a block-structured domain partitioning approach with support for dynamic load balancing [38, 39]. It is carefully designed as an HPC framework, making sure that all data structures are fully distributed, such that the memory consumption of one process does not increase with the total number of processes. The block-structured domain partitioning is flexible enough to handle complex domains [40] while, at the same time, the structured grid within each block allows for efficient implementations of stencil-based algorithms. We chose the waLBerla framework since it has been successfully used to run large-scale phase-field simulations before [2]. It offers postprocessing and I/O capabilities specifically developed for phase-field simulations, including the creation and distributed coarsening of surface meshes to reduce the amount of data that has to be written out. Additionally, waLBerla offers a Python interface [41] for in-situ evaluation and computational steering. This interface also helps to couple our Python-based code-generation approach to the framework.

4.2 Integration of Generated Kernels

There are two ways to combine our Python code generation approach with the waLBerla C++ framework. The first option is designed to be used in an interactive workflow where Python is used as the driving language. Generated kernels are compiled into Python C-extension modules which are loaded at runtime. They operate on objects implementing the Python buffer protocol, e.g., `numpy` [42] arrays, and are thus seamlessly integrated into Python's large ecosystem of scientific packages. Optionally, waLBerla can be loaded as a Python module if distributed-memory runs are required. Our code generation can be used in conjunction with technologies like interactive Jupyter notebooks [43] to combine code, visualization and documentation. Parameterizations and model variants can be quickly explored and tested in this flexible, interactive environment. While the driving code is written in a slow scripting language, the overall performance is not significantly impacted because most of the runtime is spent in the generated and compiled compute kernels. For production runs a second coupling option is available. Here the specific model and its parameters are chosen before compilation and compute kernels are generated by the CMake build system of waLBerla. This approach has the advantage that the application can be fully compiled and linked before running, such that no compiler installation is required on the compute nodes.

After discretizing the phase-field model using second order finite differences and an explicit Euler scheme, two compute kernels are obtained [2]. One kernel to update the phase-field and one for updating the chemical potential. Two arrays are stored for each variable representing two adjacent time steps at t and $t + \Delta t$: two destination arrays labeled ϕ_{dst} and ϕ_{src} store the state at $t + \Delta t$ and two source arrays labeled ϕ_{src} and μ_{src} represent the state at the current time step t. One time step is shown in Algorithm 1. Superscripts denote neighbor access patterns of the kernels, DdCn refers to a d dimensional stencil accessing n cells, the center cell and $n-1$ neighbors. Each kernel can optionally be split into two

Algorithm 1 Timestep

1: $\phi_{\text{dst}} \leftarrow \phi\text{-kernel}\left(\phi_{\text{src}}^{\text{D3C7}}, \mu_{\text{src}}^{D3C1}\right)$ using "ϕ-full" or "ϕ-split"
2: ϕ_{dst} communication and boundary handling
3: $\mu_{\text{dst}} \leftarrow \mu\text{-kernel}\left(\mu_{\text{src}}^{\text{D3C7}}, \phi_{\text{src}}^{\text{D3C19}}, \phi_{\text{dst}}^{\text{D3C19}}\right)$ using "μ-full" or "μ-split"
4: μ_{dst} communication and boundary handling
5: Swap $\phi_{\text{src}} \leftrightarrow \phi_{\text{dst}}$ and $\mu_{\text{src}} \leftrightarrow \mu_{\text{dst}}$

parts to prevent re-computation of staggered values. Then, in a first pass over the domain, flux quantities at staggered positions are cached in a temporary array and used in the second iteration pass to update the destination array. We label the splitted kernel variants "ϕ-split" and "μ-split", and the single-pass versions "ϕ-full" and "μ-full".

4.3 Communication

To enable distributed-memory parallelization of stencil codes, waLBerla provides ghost-layer synchronization functions. These communication routines have been only available for CPU simulations. For GPU simulations we extend the waLBerla communication stack with a new implementation that can make use of CUDA-enabled MPI and the GPUDirect technology. The ghost layer exchange is broken down into two parts. First, the ghost-layers are packed into a separate buffer that is stored contiguously in memory. Then, this buffer is sent to the neighboring process in a single message using asynchronous MPI functions. The packing step can either be implemented with `cudaMemcpy3D` calls or by a manually

written kernel. While the first option is simpler to realize, the latter option was chosen, since it gives better performance results. If a CUDA-enabled MPI library is available, we pack the ghost layers into a buffer in device memory to take advantage of the GPUDirect technology. The buffer can be transferred directly to the network adaptor with remote direct memory access (RDMA). On systems where no CUDA-enabled MPI is available the buffers are staged in host memory. To utilize the GPU as efficiently as possible, we schedule the kernels on multiple CUDA streams to overlap independent kernel executions. Additionally, asynchronous MPI communication is used to hide communication operations behind computations. Since the ϕ kernel does not access neighboring values of the μ array, the ghost layer exchange of μ can be overlapped with the time evolution of ϕ. The communication of ϕ cannot be hidden in the same straightforward way since the μ update accesses the ghost layer of ϕ. Thus μ is first updated in the inner part of the block, which does not require any ghost values. In a second step, the missing outer layer is computed, as soon as the communication of the ϕ ghost layers is completed.

5 RESULTS AND DISCUSSION

In this section we discuss the benefits of the code generation approach for phase-field simulation codes, present performance and scaling results on CPU and GPU clusters, and show large-scale simulation results of two different physical systems.

5.1 Benefits of Code Generation Approach

Our meta-programming approach enables us to generate highly efficient code for systems with arbitrary number of phases and components. All optimizations are written in the form of transformations, making them applicable to newly developed models as well. The central benefit of our approach is that all available model information, like configuration parameters, can already be used at compile time. Our phase-field model in its full generality requires a large number of these configuration parameters. Assuming an affine linear dependency on temperature, the specific form of the driving force (6) requires $2(N^2 + N + 1)$ configuration parameters. Phase-dependent mobility matrices $\mathbf{M}$ increase this value by $N \cdot (K-1)^2$, with K being the number of components. For a model with 4 phases, 3 components (e.g., ternary eutectic solidification), more than 50 material-dependent quantities are required for configuration. When developing an application framework without code generation techniques, these values have to be read from a configuration file at runtime. The application has to be written in a generic way to simulate systems of arbitrary parameterization. In our code generation approach we already insert the numeric values for all material-dependent values at compile time. A subsequent optimization pass pre-computes values and automatically simplifies the model. If simplified configuration values are chosen, for example symmetric diffusivity matrices or a constant temperature, this special setup can be exploited through further automatic simplification. A generic application without code generation would have to spend FLOPs to compute unnecessary expressions in these cases. Since we fix the parametrization at compile time, each change of options requires recompilation. A full recompilation of all compute kernels takes about 30 to 60 seconds. This is no problem for production runs with compute times of several hours. During model development or for parameter studies, multiple recompilations can slow down the workflow. If this is a concern, the user may choose a set of parameters that will remain variables at runtime. During the compilation process these parameters then remain symbolic and become function arguments to the generated C or CUDA kernels.

To verify that our automatic optimization approach is working, we reproduce the setup that was manually optimized by [2]. It is a setup with 4 phases, 3 components, isotropic gradient energy density ($A_{\alpha\beta} = 1$), and an analytic temperature gradient depending on time and one spatial coordinate. For later reference, we label this parameterization $P1$. Table 1 shows the number of operations required for the ϕ and μ kernels. Additions and multiplications are counted as one operation, divisions as 16, approximate square roots as 10, and approx. inverse square roots are counted as 2 FLOPs, which approximately matches their throughput on the Skylake architecture [44]. The μ-split kernel requires almost only half of the operations compared to the full version, indicating that most FLOPs are spent computing staggered values. In [2] the special functional dependency of the temperature was manually exploited by rearranging expressions such that temperature-dependent quantities could be pulled before the innermost loop. Our pipeline conducts these optimizations automatically, such that for the μ-split kernel only 1 328 normalized FLOPs are necessary (Table 1). Our automatic simplifications even slightly outperforms the result reported in [2], who report 1 384 FLOPs for their manually optimized μ-kernel.

A second parameterization, termed $P2$, is chosen to demonstrate how apparently small changes in the model can lead to vastly different performance characteristics of the generated kernels. For $P2$, the number of phases is reduced to 3, the number of components to 2. However, this time an anisotropic gradient energy density $A_{\alpha\beta}$ is chosen. This drastically increases the amount of computation required for the evolution of ϕ. Without code generation techniques a complete re-implementation of the kernel would have been necessary in this case.

6 PERFORMANCE RESULTS

In the following section we present single node performance results as well as scaling experiments on Europe's fastest supercomputing systems, the CPU-based SuperMUC-NG cluster located at Leibnitz Supercomputing Center (LRZ) in Munich and the GPU-based Piz Daint system located at the Swiss National Computing Centre in Lugano. These systems are currently rank 8 and 5 on the TOP500 list [45]. The SuperMUC-NG system is equipped with 6 480 compute nodes containing Intel Xeon Platinum 8174 processors. Every compute node has 2 sockets with 24 cores each. The nodes are bundled into eight islands forming a fat tree network topology. SuperMUC-NG is not in normal production operation yet. We are grateful to the LRZ computing centre for giving us early access to the system. For GPU simulations the Piz Daint system was used. It consists of 5 704 compute nodes. Each node is equipped with one NVIDIA Tesla P100 GPU. PizDaint is interconnected with an "Aries" network in dragonfly topology.

	P1				P2			
	μ full	μ partial	φ full	φ partial	μ full	μ partial	φ full	φ partial
loads	112	84 + 22	30	16 + 54	79	60 + 13	58	48 + 40
stores	2	6 + 2	4	12 + 4	1	3 + 1	3	9 + 3
adds	542	256 + 75	334	66 + 202	293	142 + 26	1087	364 + 368
muls	788	389 + 90	526	124 + 282	488	248 + 46	2081	792 + 557
divs	19	6 + 11	9	0 + 9	18	6 + 9	50	18 + 14
sqrts	42	21 + 0	0	0 + 0	6	3 + 0	0	0 + 0
rsqrts	36	18 + 0	0	0 + 0	24	12 + 0	0	0 + 0
norm. FLOPS	2126	1328	1004	818	1177	756	3968	2593

Table 1: Number of floating point operations (additions, multiplications, divisions, square roots, and inverse square roots) for all compute kernels for one lattice cell. Loads and stores count the number of double precision values loaded/stored in each cell. For split/partial kernels the number left of the "+" refers to the kernel for staggered value precomputation, the number right of the "+" to the kernel computing the final, cell-centered value. The last row shows normalized FLOPS, a weighted sum of all FLOPS, according to their throughput on Skylake.

6.1 CPUs: SuperMUC-NG

We employ *Kerncraft* [36] with the ECM model to predict and measure the single node performance behavior on SuperMUC-NG. All kernels are neither communication nor I/O bound, so the relevant effects take place within a socket. Thus, we restrict our analysis to a single socket (24 cores) and can safely assume that any results will scale to the second socket. As we use MPI parallelization for production runs, NUMA effects may be neglected for our discussion. The generated kernels exploit spatial blocking to reduce main memory traffic. We derive suitable blocking factors using the layer condition [36, 46]: For configuration $P1$, the most demanding kernel μ-full has a cache storage demand of $232 \cdot N^2$ Bytes to fulfill the 3D layer condition, assuming a loop length of N for the two innermost loops. Applying it to Skylake's 1 MB L2 cache, we find suitable blocking sizes of $N < 67$ which minimize main memory traffic. For the single node performance analysis and scaling experiments on SuperMUC-NG we thus use block sizes of 60^3. The major challenge in code generation and performance optimizing transformations is identifying and selecting the fastest variant. We use *Kerncraft's* automated performance modeling capability to provide a performance rating of the candidates. The ECM model predictions and single node measurements using *Kerncraft* of the two μ kernel variants are presented in Figure 2 on the left. Performance results are reported in million lattice updates per second (MLUP/s).

Comparing μ-split and μ-full kernel variants, we notice two different single-socket scaling behaviors: μ-split's performance per core decreases with the number of cores and μ-full's performance per core stays constant regardless of the number of cores, reflecting the fact that the full version recomputes values instead of loading them from memory. Both effects show in the benchmark measurements as well as ECM model predictions. The decreasing performance of μ-split indicates that it is influenced by data transfers. Its scalability limit is predicted at 32 cores (per socket). The constant performance of μ-full means that it is compute bound by core local resources up to at least 24 cores and is predicted to scale to 83 cores. While the qualitative scaling trend is correctly predicted by *Kerncraft*, comparing the absolute ECM prediction and measurement we see a discrepancy of up to a factor of 2. Due to the complexity of the underlying kernels, which contain more than 3 000 instructions and close to 24 kilobyte of binary op code, the ECM model gives only a light-speed prediction. Predictions became more accurate when the kernels are bound by data transfers (μ-split in the second half of the plot), as the effects are better understood. Selection of the faster kernel variant, based on the ECM model, would choose μ-split over μ-full for full-socket runs, due to the cross-over point at 16 cores. This effect however does not yet show in the measurements with 24 cores, but by visual extrapolation would occur at 26 cores. This delay is due to the more accurate upper-bound prediction of μ-split's performance with larger core counts. When using larger domain sizes, not mimicking the spatial blocking, we have seen the same behavior within the bounds of a single socket. To show that different high-level model configurations for the same kernel produce very distinct performance behaviors, we model and measure ϕ-split and ϕ-full variants for the $P1$ and $P2$ configuration, as shown in Figure 2 in the middle. As predicted by the model, for $P1$ the full version performs better, while for $P2$ the ϕ-split kernel is the faster choice. Overall, we can say that even though the ECM performance model as implemented in *Kerncraft* did not give precise predictions of absolute runtime, it correctly predicted the trends and upper bounds and allowed a relative comparison and informed selection of the most suitable variant. The issues with the compute bound kernels are largely due to the dependency of *Kerncraft* on IACA for the compute throughput analysis, a closed source blackbox utility, as well as unknown and undisclosed internal workings of the underlying microarchitecture.

For the $P1$ setup, the fastest option is to use the μ-split kernel combined with the ϕ-full kernel. To allow comparison of the single node analysis with multi node runs, we combine the measurements of pure kernel execution to get 7 MLUP/s. This number does not yet include any communication overhead, boundary handling and tooling necessary for multi node runs. Including these components as well, our generated application for the $P1$ setup achieves a sustained performance of about 6 MLUP/s per core (Fig. 3, left). For comparison we also run the manual implementation developed in previous work [2]. Our automatically generated version outperforms the old, already highly optimized manual version by about

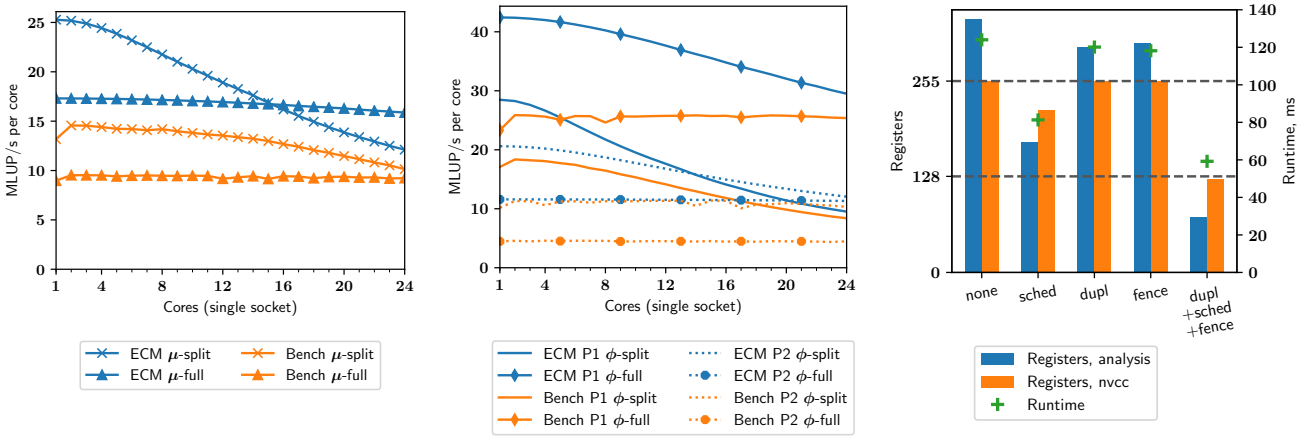

Figure 2: Single socket model and runtime comparison on SuperMUC-NG with $P1$ for μ kernels (left), and comparison of ϕ-kernels for $P1$ and $P2$ (middle). Right: Effectivity of GPU-specific transformation sequences for an exemplary μ-full kernel. Analysis counts alive intermediates (multiplied by 2 since double precision values occupy 2 registers), nvcc registers is the actual compiler allocated register count

20 %. This somewhat surprising result is due to the fact that the manually developed application was specifically optimized for the AVX2 instruction set, whereas our newly generated application optimizes for AVX512. This highlights again the important issue of performance-portability: highly specialized code needs to be adapted to new hardware architectures, a step that is fully automated in our toolchain.

When studying complex solidification scenarios, the mesh size and time step are dictated by physical considerations. However, the system behavior that is under study may manifest itself only, when large enough domains can be simulated [2]. In this case, weak scaling is the relevant metric. Figure 3 on the left shows weak scaling experiments conducted on SuperMUC-NG. Here the process count is increased keeping the workload per process constant. On SuperMUC-NG we scale to half of the machine, i.e., to all 3 168 nodes that have been available to us during the early access phase. The performance per core stays the same, indicating near perfect scalability. In other applications, also the strong scaling behavior may be relevant. Therefore, Figure 3 on the right presents a strong scaling study on the same machine, where the total domain size is fixed at $512 \times 256 \times 256$ cells. Again, good scaling behavior can be observed. The decreasing block size in this setup leads to small differences in the performance per core. For example, slightly better performance is obtained in cases where the size of the fastest dimension of a block is a multiple of the SIMD width or when cubic blocks sizes can be chosen. The execution with 48 cores already achieves a good 0.2 time steps per second. However, if necessary, we can time step the relatively small domain also on 152, 064 cores and can thus reduce the compute time to an excellent 460 time steps per second.

6.2 GPUs: PizDaint

We first evaluate the GPU-specific transformations to reduce register usage. For the exemplary μ-full kernel, Figure 2 on the right shows that rescheduling of statements is the most effective GPU register usage transformation on its own, as it manages to reduce both the number of alive intermediates and allocated registers below 255. This eliminates spilling, which increases performance by 50 %. Some of that effect can already be seen for a reordering search breadth of one, effectively a greedy search, and there is no consistent improvement for values above 20. The other two transformations, reduplication of expressions and thread fences, show only small improvements on their own, but are effective if used in combination with rescheduling. In this case, the allocated register count drops below 128, which doubles the occupancy, for a total performance improvement of a factor of 2. The GPU register usage transformations are particularly effective for the larger full kernels, and have less effect on the split kernels. The use of approximations for square roots and divisions results in a speedup of 25 – 35% for the μ kernels, which contain many of these operations.

The NVIDIA profiler reports utilization levels of 55 % to 65 % for the double precision units and 35 % to 75 % of the memory interface for the kernels run in Figure 3. Higher utilization is hindered by latency and low occupancy, and the fact that high utilization of both resources simultaneously is particularly difficult. Next, we measured the impact of the communication hiding techniques and the utilization of GPUDirect technology using 128 nodes. Table 2 shows that by fully overlapping the compute kernels with communication routines employing asynchronous MPI functionality as well as parallel CUDA streams, the overall performance of a full time step is increased by around 6 % from 395 to 422 MLUP/s. Further analysis with the `nvprof` profiler confirmed that communication latencies are hidden fully behind computations. Making use of the CUDA-enabled MPI available on the system further improves the performance from 422 to 440 MLUP/s, since in this case message buffers do not have to be staged in host memory. With this advanced communication strategy we can achieve perfect scalability

overlap	GPUDirect	MLUP/s per GPU
no	no	395
no	yes	403
yes	no	422
yes	yes	440

Table 2: Communication options on Piz Daint using 128 GPUs (right). All experiments conducted with setup $P1$.

on the PizDaint system as well (Fig.3, middle) utilizing the 2 400 nodes to which we had access.

7 SIMULATION RESULTS

Based on the different models, exemplary simulation results of setups $P1$ and $P2$ are shown. A large-scale simulation result of setup $P1$ is shown in Fig. 4 (left) based on data from [12]. Many features also found in experimental investigations [47, 48] can be found in this simulation: A two-phase chain-like structure embedded in a third-phase matrix, chain ends, rings as well as junctions. Furthermore, even though the structure looks static when viewed from the top, a side view reveals that the top view is given by a complex evolving fiber structure as studied in [49, 50]. The case of $P2$ describes dendritic solidification; a typical real-world example is shown in Fig. 4 (middle and right) for a binary aluminium-copper alloy under directional solidification. The image shows dendrites growing with different orientations relative to the temperature gradient. For the simulation, nine seeds with three different orientations were placed at the bottom of the domain. The temperature gradient was oriented towards the top of the domain, with one orientation along this axis (teal) and the other two (green and purple) misoriented relative to this first orientation. Both the experiment and the simulation show the qualitative features of dendritic growth: The dendrite grows with a parabolic tip followed by a wider trunk. On the dendritic trunk, sidearms grow further away from the dendrite tip. Furthermore, the competitive nature can be seen from dendrites of one orientation being overgrown by dendrites of another orientation, as the red circle in Fig. 4 shows.

8 CONCLUSION

In this article, we have developed a meta-programming methodology, to generate highly efficient, scalable codes for phase-field simulations in material science. The application scientist can define the physical model using a flexible and concise free energy formalism. Automatic transformations are employed to reformulate the problem on successively lower abstraction layers, based on the PDE, on stencils, an intermediate representation, and finally C or CUDA code. The user can extend the description on each level e.g., by coupling additional equations on the PDE level or by adding fluctuating terms. Optimizing transformations rewrite the model for fast execution on CPUs and GPUs. With this methodology, we can automatically perform optimizations that had previously required tedious manual work. As shown before in [2], a generic C implementation could be accelerated by a factor of 80 by elaborate manual optimizations. Exploiting application knowledge, the specific parameterization, and its geometric properties are crucial. This is essential to simulate large domain sizes with fine enough resolution, as it is often required to obtain physically meaningful results. Because of the large development effort, this step could previously only be performed for a few important model instances. The new code generation method presented in this article automates the time-consuming optimization process and provides a flexible, modular tool for implementing a wide class of phase-field models. The development and manual optimization for a single model instance as in [2] requires more than one person-year of development time of a highly specialized computer scientist, who must operate in close collaboration with the team of application scientists. This high effort can be reduced by our code generation approach by at least one order of magnitude. Since the metaprogramming approach allows for better separation-of-concerns and increases the reuseability of components, we expect that the development times can be decreased even further in the future.

We have separated the model description from the work of coding non-trivial optimized kernels via several stages of automatized transformations. This separation of concerns allows computer scientists to develop new hardware-dependent optimizations without having to understand all the details of the model. On the other side, a material scientist can formulate the physical model in a natural way without having to worry about algorithmic optimization or specific details of the target supercomputer architecture. We could achieve similar performance as the quite involved manual optimization of [2]. Due to performance-portability reasons, we can even outperform them on the latest HPC systems. This becomes possible since we can use all available information, including the full parametrization already at compile time. The highly optimized, automatically generated kernels are eventually integrated into the waLBerla framework that provides all functionality for the scalable and efficient distributed memory parallel execution.

A new communication strategy for GPU-based clusters has been developed that packs message buffers directly on the device and uses the GPUDirect technology for direct transfer from the GPU to the message passing network. In summary, our code shows excellent scaling results on the CPU-based SuperMUC-NG system as well as the GPU-based PizDaint supercomputer. Note that maintaining such excellent scalability is significantly harder and more challenging for our fast node-optimized kernels than if the node-performance had not been optimized. We can nevertheless show near-perfect scalability. For future work, we plan to develop and integrate further spatial and temporal discretization options. Additionally, we are going to apply and generalize our code generation pipeline to include also other stencil-based methods, e.g., lattice Boltzmann schemes for multiphase CFD simulations.

9 ACKNOWLEDGMENTS

We thank Mareike Wegener from the German Aerospace Center (DLR) for providing the experimental image. This research has been funded by the Federal Ministry of Education and Research of Germany (BMBF) through the SKAMPY and METACCA projects. We are grateful to the Leibniz Rechenzentrum Garching and to the Swiss National Supercomputing Centre for providing computational resources.

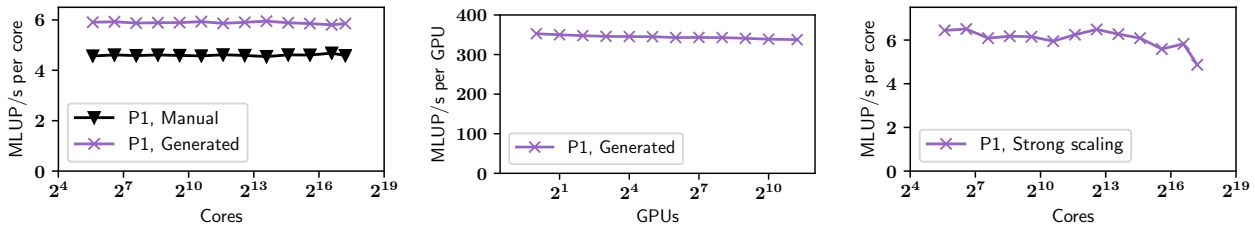

Figure 3: Weak Scaling on SuperMUC-NG with 60^3 block per core (left) and Piz Daint with 400^3 block per GPU (middle). Strong scaling on SuperMUC-NG with total domain size of $512 \times 256 \times 256$ cells (right).

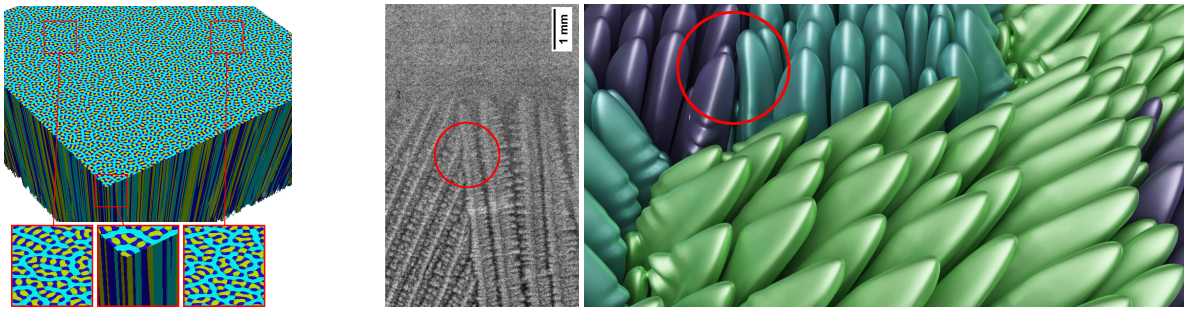

Figure 4: Large-scale phase-field simulation result of ternary eutectic directional solidification $P1$[12] (left). Experimental in-situ observation of directional solidification of an Al-Cu alloy from Mareike Wegener at the DLR [51] (middle) and a simulation with similar conditions using parameterization $P2$ (right).

REFERENCES

[1] T. Takaki, T. Shimokawabe, M. Ohno, A. Yamanaka, and T. Aoki. Unexpected selection of growing dendrites by very-large-scale phase-field simulation. *Journal of Crystal Growth*, 382:21–25, 2013.

[2] M. Bauer, J. Hötzer, M. Jainta, P. Steinmetz, M. Berghoff, F. Schornbaum, C. Godenschwager, H. Köstler, B. Nestler, and U. Rüde. Massively parallel phase-field simulations for ternary eutectic directional solidification. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 8. ACM, 2015.

[3] J. S. Rowlinson. Translation of J. D. van der Waals'"the thermodynamik theory of capillarity under the hypothesis of a continuous variation of density". *Journal of Statistical Physics*, 20(2):197–200, 1979.

[4] L. D. Landau and I. M. Khalatikow. The selected works of l. d. landau (engl. transl.), 1963.

[5] S. M. Allen and J. W. Cahn. Coherent and incoherent equilibria in iron-rich iron-aluminum alloys. *Acta Metallurgica*, 23(9):1017 – 1026, 1975. ISSN 0001-6160. doi: http://dx.doi.org/10.1016/0001-6160(75)90106-6. URL http://www.sciencedirect.com/science/article/pii/0001616075901066.

[6] J. E. Hilliard and J. W. Cahn. On the nature of the interface between a solid metal and its melt. *Acta Metallurgica*, 6(12):772 – 774, 1958. ISSN 0001-6160. doi: http://dx.doi.org/10.1016/0001-6160(58)90052-X. URL http://www.sciencedirect.com/science/article/pii/000161605890052X.

[7] U. Hecht, L. Gránásy, T. Pusztai, B. Böttger, M. Apel, V. Witusiewicz, L. Ratke, J. De Wilde, L. Froyen, D. Camel, et al. Multiphase solidification in multicomponent alloys. *Materials Science and Engineering: R: Reports*, 46(1):1–49, 2004.

[8] M. Asta, C. Beckermann, A. Karma, W. Kurz, R. Napolitano, M. Plapp, G. Purdy, M. Rappaz, and R. Trivedi. Solidification microstructures and solid-state parallels: Recent developments, future directions. *Acta Materialia*, 57(4):941–971, 2009.

[9] J. Hötzer*, M. Kellner*, P. Steinmetz*, and B. Nestler (*equal authors). Applications of the phase-field method for the solidification of microstructures in multi-component systems. *Journal of the Indian Institute of Science*, 2016.

[10] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[11] J. Hötzer, M. Jainta, P. Steinmetz, B. Nestler, A. Dennstedt, A. Genau, M. Bauer, H. Köstler, and U. Rüde. Large scale phase-field simulations of directional ternary eutectic solidification. *Acta Materialia*, 2015.

[12] J. Hötzer. Massiv-parallele und großskalige phasenfeldsimulationen zur untersuchung der mikrostrukturentwicklung, 2017.

[13] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, and Z. Liu. Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Press, 2016.

[14] A. Logg, K. Mardal, and G. Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[15] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[16] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

[17] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer. Towards green aviation with python at petascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1. IEEE Press, 2016.

[18] M. Januszewski and M. Kostur. Sailfish: A flexible multi-gpu implementation of the lattice boltzmann method. *Computer Physics Communications*, 185(9): 2350–2368, 2014.

[19] S. Kuckuk, G. Haase, D. A. Vasco, and H. Köstler. Towards generating efficient flow solvers with the exastencils approach. *Concurrency and Computation: Practice and Experience*, 29(17):e4062, 2017.

[20] M. Fernando, D. Neilsen, H. Lim, E. Hirschmann, and H. Sundar. Massively parallel simulations of binary black hole intermediate-mass-ratio inspirals. *SIAM Journal on Scientific Computing*, 41(2):C97–C138, 2019. doi: 10.1137/18M1196972. URL https://doi.org/10.1137/18M1196972.

[21] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, So. Hirata, S. Krishnamoorthy, et al. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005.

[22] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, et al. Simulations of below-ground dynamics of fungi: 1.184 pflops attained by automated generation and autotuning of temporal blocking codes. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 23–33. IEEE, 2016.

[23] C. Yount, J. Tobin, A. Breuer, and A. Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39, Nov 2016. doi: 10.1109/WOLFHPC.2016.08.

[24] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. Devito: towards a generic finite difference dsl using symbolic python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 67–75. IEEE, 2016.

[25] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman. Devito: Automated fast finite difference computation. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 11–19. IEEE, 2016.

[26] J. E. Guyer, D. Wheeler, and J. A. Warren. Fipy: Partial differential equations with python. *Computing in Science & Engineering*, 11(3):6–15, 2009.

[27] A. Choudhury and B. Nestler. Grand-potential formulation for multicomponent phase transformations combined with thin-interface asymptotics of the double-obstacle potential. *Phys. Rev. E*, 85:021602, Feb 2012. doi: 10.1103/PhysRevE.85.021602. URL https://link.aps.org/doi/10.1103/PhysRevE.85.021602.

[28] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

[29] A. Karma and W. Rappel. Quantitative phase-field modeling of dendritic growth in two and three dimensions. *Phys. Rev. E*, 57:4323–4349, Apr 1998. doi: 10.1103/PhysRevE.57.4323. URL https://link.aps.org/doi/10.1103/PhysRevE.57.4323.

[30] B. Nestler, H. Garcke, and B. Stinner. Multicomponent alloy solidification: Phase-field modeling and simulations. *PHYSICAL REVIEW E 71, 041609*, 2005.

[31] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 16. ACM, 2011.

[32] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010. ISBN 978-3-642-15581-9.

[33] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[34] Christoph W. Kessler. Scheduling expression dags for minimal register need. *Computer Languages*, 24(1):33 – 53, 1998. ISSN 0096-0551. doi: https://doi.org/10.1016/S0096-0551(98)00002-2. URL http://www.sciencedirect.com/science/article/pii/S0096055198000022.

[35] H. Stengel, J. Treibig, G. Hager, and G. Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. *Proceedings of the 29th ACM on International Conference on Supercomputing - ICS '15*, 2015. doi: 10.1145/2751205.2751240. URL http://dx.doi.org/10.1145/2751205.2751240.

[36] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein. Kerncraft: A tool for analytic performance modeling of loop kernels. In C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, editors, *Tools for High Performance Computing 2016*, pages 1–22, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56702-0.

[37] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[38] F. Schornbaum and U. Rüde. Extreme-scale block-structured adaptive mesh refinement. *SIAM Journal on Scientific Computing*, 40(3):C358–C387, 2018.

[39] F. Schornbaum and U. Rüde. Massively parallel algorithms for the lattice boltzmann method on nonuniform grids. *SIAM Journal on Scientific Computing*, 38(2): C96–C126, 2016.

[40] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2013.

[41] M. Bauer, F. Schornbaum, C. Godenschwager, M. Markl, D. Anderl, H. Köstler, and U. Rüde. A python extension for the massively parallel multiphysics simulation framework walberla. *International Journal of Parallel, Emergent and Distributed Systems*, 31(6):529–542, 2016.

[42] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2): 22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37.

[43] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P Ivanov, D. Avila, S. Abdalla, and C. Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[44] A. Fog. 4. instruction tables. URL https://www.agner.org/optimize/instruction_tables.pdf.

[45] TOP500 List. http://www.top500.org/lists/2018/11/, 2019. [Online; accessed April-2019].

[46] Layer Conditions. https://rrze-hpc.github.io/layer-condition/, 2018. [Online; accessed April-2019].

[47] A. Dennstedt and L. Ratke. Microstructures of directionally solidified al–ag–cu ternary eutectics. *Transactions of the Indian Institute of Metals*, 65(6):777–782, 2012.

[48] A. Genau and L. Ratke. Morphological characterization of the al–ag–cu ternary eutectic. *International Journal of Materials Research*, 103(4):469–475, 2012.

[49] M. Kellner, W. Kunz, P. Steinmetz, J. Hötzer, and B. Nestler. Phase-field study of dynamic velocity variations during directional solidification of eutectic nial-34cr. *Computational Materials Science*, 145:291–305, 2018.

[50] J. Hötzer, P. Steinmetz, A. Dennstedt, A. Genau, M. Kellner, Irmak Sargin, and B. Nestler. Influence of growth velocity variations on the pattern formation during the directional solidification of ternary eutectic Al-Ag-Cu. 136:335–346, 2017.

[51] M. Wegener. German Aerospace Center (DLR). private communication, 2019.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran our generated phase-field application, as described in the paper, on the SuperMUC-NG cluster using the Intel compiler in version 2019.3.199. GPU experiments have been run on the PizDaint system using the CUDA toolkit in version 9.1. Special environment variables have been set to make use of the CUDA-enabled MPI implementation:

MPICH_RDMA_ENABLED_CUDA=1 CRAY_CUDA_MPS=1 MPICH_G2G_PIPELINE=256

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

```
www.walberla.net
https://i10git.cs.fau.de/walberla/walberla
github.com/RRZE-HPC/kerncraft
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Supermuc-NG (April, 2019), CPU: Intel(R) Xeon(R) Platinum 8174 CPU @ 3.10GHz; Piz Daint(April, 2019), CPU: Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, Accelerator: Tesla P100

Operating systems and versions: SUSE Linux Enterprise Server 12 (x86_64) running Linux Kernel 4.4.126-94.22-default; SUSE Linux Enterprise Server 12 SP3 running Linux 4.4.103-6.38_4.0.153-cray_ari_c

Compilers and versions: intel/19; cudatoolkit9.1

Applications and versions: waLBerla, git version e91468474c5f4d1014abc6c37d867cceb297ff6e

Libraries and versions: mpi.intel/2019; mpich-gnu/7.1

Output from scripts that gathers execution environment information.

```
SuperMUC-NG:
MPI_F90_LIB=-L/lrz/sys/intel/studio2019_u3/impi/2019
↪  .3.199/intel64/lib -Xlinker --enable-new-dtags
↪  -Xlinker -rpath -Xlinker /lrz/sys/intel/studio20
↪  19_u3/impi/2019.3.199/intel64/lib
↪  -lmpifort
MKLROOT=/lrz/sys/intel/studio2019_u3/compilers_and_l
↪  ibraries_2019.3.199/linux/mkl
MPI_DEBUG_LIB=-L/lrz/sys/intel/studio2019_u3/impi/20
↪  19.3.199/intel64/lib/debug -Xlinker
↪  --enable-new-dtags -Xlinker -rpath -Xlinker
↪  /lrz/sys/intel/studio2019_u3/impi/2019.3.199/int
↪  el64/lib/debug -lmpi -ldl -lrt
↪  -lpthread
MODULE_VERSION_STACK=4.0.0
LESSKEY=/etc/lesskey.bin
SLURM_NODELIST=i01r02c02s02
SLURM_CHECKPOINT_IMAGE_DIR=/var/slurm/checkpoint
NNTPSERVER=news
LRZ_OS_SUBVER=3
MANPATH=/lrz/sys/compilers/gcc/7.3.0/share/man:/lrz/
↪  sys/intel/studio2019_u3/inspector_2019.3.0.59148
↪  4/man:/lrz/sys/intel/studio2019_u3/vtune_amplifi
↪  er_2019.3.0.591499/man:/lrz/sys/intel/studio2019
↪  _u3/advisor_2019.3.0.591490/man:/lrz/sys/intel/s
↪  tudio2019_u3/itac/2019.3.032/man:/lrz/sys/intel/
↪  studio2019_u3/impi/2019.3.199/man:/lrz/sys/intel
↪  /studio2019_u3/documentation_2019/en/debugger/gd
↪  b-ia/man:/lrz/sys/intel/studio2019_u3/documentat
↪  ion_2019/en/debugger/gdb-mic/man:/lrz/sys/intel/
↪  studio2019_u3/documentation_2019/en/debugger/gdb
↪  -igfx/man:/lrz/sys/intel/studio2019_u3/compilers_
↪  and_libraries_2019.3.199/linux/man/common:/lrz/s
↪  ys/share/modules/share/man:/usr/local/man:/usr/l
↪  ocal/share/man:/usr/share/man
SLURM_JOB_NAME=test_job
AMPLIFIER_XE_DOC=/lrz/sys/intel/studio2019_u3/vtune_
↪  amplifier_2019.3.0.591499/documentation/en
MPI_INSTALL_DOC=/lrz/noarch/src/intel/mpi/README_LRZ
VT_MPI=impi4
HOSTNAME=i01r02c02s02
SLURM_NTASKS_PER_NODE=1
SLURM_TOPOLOGY_ADDR=core.i01opa.i01r02c02s02
SLURMD_NODENAME=i01r02c02s02
I_MPI_PIN=1
```

```
_LMFILES__modshare=/lrz/sys/share/modules/files/comp
↪ ilers/gcc/7:1:/lrz/sys/share/modules/files/envir
↪ onment/devEnv/Intel/2019:1:/lrz/sys/share/module
↪ s/files/libraries/mkl/2019:1:/lrz/sys/share/modu
↪ les/files/tools/spack/staging/19.1:1:/lrz/sys/sh
↪ are/modules/files/tools/inspector_xe/2019:1:/lrz
↪ /sys/share/modules/files/parallel/mpi.intel/2019
↪ :1:/lrz/sys/share/modules/files/tools/advisor_xe
↪ /2019:1:/lrz/sys/share/modules/files/environment
↪ /lrz/default:1:/lrz/sys/share/modules/files/comp
↪ ilers/intel/19.0:1:/lrz/sys/share/modules/files/
↪ tools/amplifier_xe/2019:1:/lrz/sys/share/modules
↪ /files/environment/admin/1.0:1:/lrz/sys/share/mo
↪ dules/files/tools/itac/2019:1:/lrz/sys/share/mod
↪ ules/files/tools/cmake/3.6:1:/lrz/sys/share/modu
↪ les/files/environment/tempdir/1.0:1
LRZ_SYSTEM=Supercomputer
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
EAR_GROUP=hpcusers
SLURM_PRIO_PROCESS=0
MODULEPATH_modshare=/lrz/sys/share/modules/files/io_
↪ tools:1:/lrz/sys/share/modules/files/environment
↪ :1:/lrz/sys/share/modules/files/graphics:1:/lrz/
↪ sys/share/modules/files/libraries:1:/lrz/sys/sha
↪ re/modules/files/parallel:1:/lrz/sys/share/modul
↪ es/files/tools:1:/lrz/sys/share/modules/files/co
↪ mpilers:1:/lrz/sys/spack/staging/19.1/modules/x8
↪ 6_avx512/linux-sles12-x86_64:1:/lrz/sys/share/mo
↪ dules/files/applications:1
INTEL_LICENSE_FILE=/lrz/sys/intel/licenses
SLURM_NODE_ALIASES=(null)
CMAKE_SRC=/lrz/noarch/src/tools/cmake/cmake-3.6.3
ADVISOR_XE_WWW=http://www.lrz.de/services/software/p
↪ rogrammierung/intel_advisor/
AMPLIFIER_XE_INSTALL_DOC=/lrz/noarch/src/intel/tools
↪ /README_amplifier_lrz
LRZ_SYSTEM_SEGMENT=Thin_Node
HOST=login03
TERM=xterm-256color
SHELL=/bin/bash
I_MPI_FABRICS=shm:ofi
INSPECTOR_XE_WWW=http://www.lrz-muenchen.de/services
↪ /software/programmierung/inspector_xe
PROFILEREAD=true
HISTSIZE=1000
SLURM_JOB_QOS=test
TMPDIR=/tmp/USER
SSH_CLIENT=131.188.39.70 58274 22
SLURM_TOPOLOGY_ADDR_PATTERN=switch.switch.node
PLG_LST_CTX=SBATCH
CMAKE_BASE=/lrz/sys/tools/cmake/3.6.3
MORE=-sl
LRZ_LOAD_DEFAULTS=yes
MODULE_INCLUDE=/lrz/sys/share/modules/include
GCC_DOC=/lrz/sys/compilers/gcc/7.3.0/share/info
MIC_LD_LIBRARY_PATH=/lrz/sys/intel/studio2019_u3/com
↪ pilers_and_libraries_2019.3.199/linux/mkl/lib/mic
I_MPI_PLATFORM=skx
SSH_TTY=/dev/pts/34
MPI_SRC=/lrz/noarch/src/intel/mpi
MIC_LD_LIBRARY_PATH_modshare=/lrz/sys/intel/studio20
↪ 19_u3/compilers_and_libraries_2019.3.199/linux/m
↪ kl/lib/mic:1
PLG_PATH_PFX=/opt/ear
ADVISOR_XE_INSTALL_DOC=/lrz/noarch/src/intel/tools/R
↪ EADME_advisor_lrz
DEVENV_SRC=/lrz/noarch/src/intel/compilers/
http_proxy=http://localhost:3128
USER=USER
SLURM_NNODES=1
LD_LIBRARY_PATH=/lrz/sys/compilers/gcc/7.3.0/lib64:/
↪ lrz/sys/compilers/gcc/7.3.0/lib:/lrz/sys/intel/s
↪ tudio2019_u3/itac/2019.3.032/slib:/lrz/sys/intel
↪ /studio2019_u3/impi/2019.3.199/intel64/libfabric
↪ /lib:/lrz/sys/intel/studio2019_u3/impi/2019.3.19
↪ 9/intel64/lib/release:/lrz/sys/intel/studio2019_
↪ u3/impi/2019.3.199/intel64/lib:/lrz/sys/intel/st
↪ udio2019_u3/compilers_and_libraries_2019.3.199/l
↪ inux/mkl/lib/intel64:/lrz/sys/intel/studio2019_u
↪ 3/debugger_2019/libipt/intel64/lib:/lrz/sys/inte
↪ l/studio2019_u3/compilers_and_libraries_2019.3.1
↪ 99/linux/compiler/lib/intel64
LS_COLORS=no=00:fi=00:di=01;34:ln=00;36:pi=40;33:so=
↪ 01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=41;33;
↪ 01:ex=00;32:*.cmd=00;32:*.exe=01;32:*.com=01;32:
↪ *.bat=01;32:*.btm=01;32:*.dll=01;32:*.tar=00;31:
↪ *.tbz=00;31:*.tgz=00;31:*.rpm=00;31:*.deb=00;31:
↪ *.arj=00;31:*.taz=00;31:*.lzh=00;31:*.lzma=00;31
↪ :*.zip=00;31:*.zoo=00;31:*.z=00;31:*.Z=00;31:*.g
↪ z=00;31:*.bz2=00;31:*.tb2=00;31:*.tz2=00;31:*.tb
↪ z2=00;31:*.xz=00;31:*.avi=01;35:*.bmp=01;35:*.fl
↪ i=01;35:*.gif=01;35:*.jpg=01;35:*.jpeg=01;35:*.m
↪ ng=01;35:*.mov=01;35:*.mpg=01;35:*.pcx=01;35:*.p
↪ bm=01;35:*.pgm=01;35:*.png=01;35:*.ppm=01;35:*.t
↪ ga=01;35:*.tif=01;35:*.xbm=01;35:*.xpm=01;35:*.d
↪ l=01;35:*.gl=01;35:*.wmv=01;35:*.aiff=00;32:*.au
↪ =00;32:*.mid=00;32:*.mp3=00;32:*.ogg=00;32:*.voc
↪ =00;32:*.wav=00;32:
LRZ_SUB_ARCH=SKX_8186
DSM_LOG=/dss/dsshome1/05/USER
```

```
MKL_LIB=-L/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64 -Wl,--start-group /lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64/libmkl_intel_lp64.a /lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64/libmkl_intel_thread.a /lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64/libmkl_core.a -Wl,--end-group -liomp5 -lpthread -lm
ROMIO_FSTYPE_FORCE=gpfs:
MPI_DEBUG_MT_LIB=-L/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib/debug_mt -Xlinker --enable-new-dtags -Xlinker -rpath -Xlinker /lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib/debug_mt -lmpi -ldl -lrt -lpthread
XNLSPATH=/usr/share/X11/nls
FI_PROVIDER_PATH=/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/libfabric/lib/prov
CLASSPATH_modshare=/lrz/sys/intel/studio2019_u3/itac/2019.3.032/lib:1
MIC_LIBRARY_PATH=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/mic
MODULE_NORELOAD=admin/1.0:tempdir/1.0:lrz/1.0
GCC_SRC=/lrz/noarch/src/gcc
INSPECTOR_XE_INC=-I/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484/include
MPI_CXX_LIB=-L/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib -Xlinker --enable-new-dtags -Xlinker -rpath -Xlinker /lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib -lmpicxx
QEMU_AUDIO_DRV=pa
WORK_LIST=/hppfs/work/ACCOUNT/USER
HOSTTYPE=x86_64
SLURM_JOBID=68195
MKL_VML_SHLIB=-L/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64 -lmkl_vml_avx512 -liomp5 -lpthread -lm
FTP_PROXY=http://localhost:3128
ftp_proxy=http://localhost:3128
MKL_BASE=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl
MPI_LIB=-L/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib/release -Xlinker --enable-new-dtags -Xlinker -rpath -Xlinker /lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lib/release -lmpi -ldl -lrt -lpthread
FROM_HEADER=
SLURM_NTASKS=1
DEVENV_DOC=/lrz/sys/intel/studio2019_u1/https://doku.lrz.de/display/PUBLIC/Software+for+HPC
SALLOC_PARTITION=test
PAGER=less
MODULE_BIN=/lrz/sys/share/modules/bin
CSHEDIT=emacs
GCC_BASE=/lrz/sys/compilers/gcc/7.3.0
XDG_CONFIG_DIRS=/etc/xdg
NLSPATH=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64/locale/en_US/%N:/lrz/sys/intel/studio2019_u3/debugger_2019/gdb/intel64/share/locale/%l_%t/%N:/lrz/sys/intel/studio2019_u3/debugger_2019/gdb/intel64_mic/share/locale/%l_%t/%N:/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/compiler/lib/intel64/locale/en_US/%N
INSPECTOR_XE_DOC=/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484/documentation/en
LIBGL_DEBUG=quiet
MINICOM=-c on
GCC_WWW=http://www.lrz-muenchen.de/services/software/programmierung/gcc
INSPECTOR_XE_BASE=/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484
ADVISOR_XE_DOC=/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490/documentation/en
VT_ADD_LIBS=-ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread
LRZ_OS=SUSE Linux Enterprise Server 12 (x86_64)
TEMPDIR_NOCHECK=yes
OPT_TMP=/hppfs/scratch/05/USER
MODULE_VERSION=4.0.0
MAIL=/var/mail/USER
PATH=/dss/dsshome1/05/USER/grandchem/miniconda/bin:/lrz/sys/tools/cmake/3.6.3/bin:/lrz/sys/compilers/gcc/7.3.0/bin:/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484/bin64:/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499/bin64:/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490/bin64:/lrz/sys/intel/studio2019_u3/itac/2019.3.032/bin:/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lrzbin:/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/bin:/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/bin/intel64:/lrz/sys/share/modules/bin:/lrz/sys/opt/lenovo/onecli:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
SLURM_TASKS_PER_NODE=1
LRZ_NOCHECK=yes
CPU=x86_64
SLURM_WORKING_CLUSTER=supermucng:172.16.226.80:6817:8448
MKL_INSTALL_DOC=/lrz/noarch/src/intel/libraries/mkl/README_lrz
INTEL_DOC=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/Documentation
```

```
LRZ_OS_VER=12
SSH_SENDS_LOCALE=yes
_=/usr/bin/env
SLURM_JOB_ID=68195
MPI_INC=-I/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/include
MKL_LIBDIR=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64
INPUTRC=/etc/inputrc
PWD=/dss/dsshome1/05/USER
SLURM_JOB_USER=USER
_LMFILES_=/lrz/sys/share/modules/files/environment/admin/1.0:/lrz/sys/share/modules/files/environment/tempdir/1.0:/lrz/sys/share/modules/files/tools/spack/staging/19.1:/lrz/sys/share/modules/files/environment/lrz/default:/lrz/sys/share/modules/files/compilers/intel/19.0:/lrz/sys/share/modules/files/libraries/mkl/2019:/lrz/sys/share/modules/files/parallel/mpi.intel/2019:/lrz/sys/share/modules/files/tools/itac/2019:/lrz/sys/share/modules/files/tools/advisor_xe/2019:/lrz/sys/share/modules/files/tools/amplifier_xe/2019:/lrz/sys/share/modules/files/tools/inspector_xe/2019:/lrz/sys/share/modules/files/environment/devEnv/Intel/2019:/lrz/sys/share/modules/files/compilers/gcc/7:/lrz/sys/share/modules/files/tools/cmake/3.6
MKL_SRC=/lrz/noarch/src/intel/libraries/mkl
LRZ_ARCH=x86_64_intel
PTMP=
SPACK_DOC=https://spack.readthedocs.io/
MPI_BASE=/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64
INSPECTOR_XE_INSTALL_DOC=/lrz/noarch/src/intel/tools/README_inspector_lrz
GRID_ENV=LRZ
LANG=en_US
MKL_F90_INC=-I/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/include/intel64/lp64
ADVISOR_XE_SRC=/lrz/noarch/src/intel/tools
PYTHONSTARTUP=/etc/pythonstart
MODULEPATH=/lrz/sys/share/modules/files/applications:/lrz/sys/share/modules/files/compilers:/lrz/sys/share/modules/files/environment:/lrz/sys/share/modules/files/graphics:/lrz/sys/share/modules/files/libraries:/lrz/sys/share/modules/files/io_tools:/lrz/sys/share/modules/files/parallel:/lrz/sys/share/modules/files/tools:/lrz/sys/spack/staging/19.1/modules/x86_avx512/linux-sles12-x86_64
VT_LIB_DIR=/lrz/sys/intel/studio2019_u3/itac/2019.3.032/lib
DEVENV_BASE=/lrz/sys/intel/studio2019_u1
LOADEDMODULES=admin/1.0:tempdir/1.0:spack/staging/19.1:lrz/default:intel/19.0:mkl/2019:mpi.intel/2019:itac/2019:advisor_xe/2019:amplifier_xe/2019:inspector_xe/2019:devEnv/Intel/2019:gcc/7:cmake/3.6
EAR_USER=USER
SLURM_JOB_UID=3948295
INTEL_INSTALL_DOC=/lrz/noarch/src/intel/compilers/README_lrz
MKL_DOC=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/../documentation/en/mkl
SLURM_NODEID=0
INSPECTOR_XE_SRC=/lrz/noarch/src/intel/tools
VT_FLUSH_PREFIX=/hppfs/scratch/05/USER
WORK_LIST_modshare=/hppfs/work/ACCOUNT/USER:1
SLURM_SUBMIT_DIR=/dss/dsshome1/05/USER
INTEL_LIBDIR=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/compiler/lib/intel64
AMPLIFIER_XE_WWW=http://www.lrz.de/services/software/programmierung/intel_amplifier/
VT_ROOT=/lrz/sys/intel/studio2019_u3/itac/2019.3.032
ADVISOR_XE_2019_DIR=/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490
SLURM_NPROCS=1
SLURM_TASK_PID=744695
GCC_INSTALL_DOC=/lrz/noarch/src/gcc/README_lrz.txt
I_MPI_HYDRA_IFACE=ib0
HTTPS_PROXY=http://localhost:3128
https_proxy=http://localhost:3128
SLURM_CPUS_ON_NODE=96
CXX=icpc
INTEL_SRC=/lrz/noarch/src/intel/compilers
ENVIRONMENT=BATCH
SLURM_PROCID=0
INTEL_PYTHONHOME=/lrz/sys/intel/studio2019_u3/debugger_2019/python/intel64/
MKL_F90_LIB=-L/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64 -lmkl_blas95_lp64 -lmkl_lapack95_lp64
GPG_TTY=/dev/pts/34
AUDIODRIVER=pulseaudio
SLURM_JOB_NODELIST=i01r02c02s02
CMAKE_WWW=http://www.cmake.org/
FI_PROVIDER=psm2
MKL_INC=-I/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/include -I/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/include/intel64/lp64
I_MPI_HYDRA_PMI_CONNECT=lazy-cache
```

```
PATH_modshare=/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484/bin64:1:/usr/bin:1:/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/bin:1:/lrz/sys/intel/studio2019_u3/impi/2019.3.199/intel64/lrzbin:1:/usr/local/bin:1:/lrz/sys/bin:1:/opt/lenovo/onecli:1:/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/bin/intel64:1:/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490/bin64:1:/lrz/sys/share/modules/bin:1:/bin:1:/lrz/sys/intel/studio2019_u3/itac/2019.3.032/bin:1:/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499/bin64:1:/usr/bin/X11:1:/lrz/sys/compilers/gcc/7.3.0/bin:1:/lrz/sys/tools/cmake/3.6.3/bin:1:/usr/games:1
QT_SYSTEM_DIR=/usr/share/desktop-data
SHLVL=3
HOME=/dss/dsshome1/05/USER
ALSA_CONFIG_PATH=/etc/alsa-pulse.conf
SDL_AUDIODRIVER=pulse
LESS_ADVANCED_PREPROCESSOR=no
OSTYPE=linux
SLURM_LOCALID=0
VT_INC=-I/lrz/sys/intel/studio2019_u3/itac/2019.3.032/include
AMPLIFIER_XE_BASE=/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499
KMP_AFFINITY=granularity=thread,compact,1,0
INTEL_BASE=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199
LS_OPTIONS=-N --color=tty -T 0
XCURSOR_THEME=DMZ
EAR_LIBRARY=0
SLURM_CLUSTER_NAME=supermucng
SLURM_JOB_CPUS_PER_NODE=96
SLURM_JOB_GID=3930044
VT_SLIB_DIR=/lrz/sys/intel/studio2019_u3/itac/2019.3.032/slib
VT_ARCH=intel64
HTTP_PROXY=http://localhost:3128
WINDOWMANAGER=env GNOME_SHELL_SESSION_MODE=classic SLE_CLASSIC_MODE=1 gnome-session --session gnome-classic
MANPATH_modshare=/usr/local/share/man:1:/lrz/sys/intel/studio2019_u3/impi/2019.3.199/man:1:/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490/man:1:/lrz/sys/intel/studio2019_u3/documentation_2019/en/debugger/gdb-mic/man:1:/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499/man:1:/lrz/sys/share/modules/share/man:1:/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/man/common:1:/usr/local/man:1:/lrz/sys/compilers/gcc/7.3.0/share/man:1:/lrz/sys/intel/studio2019_u3/inspector_2019.3.0.591484/man:1:/lrz/sys/intel/studio2019_u3/documentation_2019/en/debugger/gdb-ia/man:1:/lrz/sys/intel/studio2019_u3/itac/2019.3.032/man:1:/usr/share/man:1:/lrz/sys/intel/studio2019_u3/documentation_2019/en/debugger/gdb-igfx/man:1
SLURM_SUBMIT_HOST=login03
SLURM_GTIDS=0
NLSPATH_modshare=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/compiler/lib/intel64/locale/en_US/%N:1:/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/lib/intel64/locale/en_US/%N:1:/lrz/sys/intel/studio2019_u3/debugger_2019/gdb/intel64/share/locale/%l_%t/%N:1:/lrz/sys/intel/studio2019_u3/debugger_2019/gdb/intel64_mic/share/locale/%l_%t/%N:1
AMPLIFIER_XE_LIBDIR=/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499/lib64
MKL_INCDIR=/lrz/sys/intel/studio2019_u3/compilers_and_libraries_2019.3.199/linux/mkl/include
SLURM_JOB_PARTITION=test
PYTHONPATH=/dss/dsshome1/05/USER/grandchem/code/walberla/python:
ADVISOR_XE_INC=-I/lrz/sys/intel/studio2019_u3/advisor_2019.3.0.591490/include
MPI_DOC=/lrz/sys/intel/studio2019_u3/impi/2019.3.199/doc
G_FILENAME_ENCODING=@locale,UTF-8,ISO-8859-15,CP1252
OPERATING_SYSTEM=Linux
ADMIN_NOCHECK=yes
LESS=-M -I -R
MACHTYPE=x86_64-suse-linux
LOGNAME=USER
BOOST_ROOT=/dss/dsshome1/05/USER/pe/boost
MPMATH_NOGMPY=1
AMPLIFIER_XE_INC=-I/lrz/sys/intel/studio2019_u3/vtune_amplifier_2019.3.0.591499/include
```

```
LD_LIBRARY_PATH_modshare=/lrz/sys/intel/studio2019_u
↪ 3/impi/2019.3.199/intel64/lib/release:1:/lrz/sys
↪ /intel/studio2019_u3/impi/2019.3.199/intel64/lib
↪ :1:/lrz/sys/compilers/gcc/7.3.0/lib:1:/lrz/sys/i
↪ ntel/studio2019_u3/impi/2019.3.199/intel64/libfa
↪ bric/lib:1:/lrz/sys/intel/studio2019_u3/itac/201
↪ 9.3.032/slib:1:/lrz/sys/intel/studio2019_u3/comp
↪ ilers_and_libraries_2019.3.199/linux/compiler/li
↪ b/intel64:1:/lrz/sys/compilers/gcc/7.3.0/lib64:1
↪ :/lrz/sys/intel/studio2019_u3/compilers_and_libr
↪ aries_2019.3.199/linux/mkl/lib/intel64:1:/lrz/sy
↪ s/intel/studio2019_u3/debugger_2019/libipt/intel
↪ 64/lib:1
MKL_WWW=http://www.lrz-muenchen.de/services/software
↪ /programmierung/intel_libs
WORK_ACCOUNT=/hppfs/work/ACCOUNT/USER
CVS_RSH=ssh
DEVENV_INSTALL_DOC=/lrz/noarch/src/intel/compilers//
↪ README_lrz
CLASSPATH=/lrz/sys/intel/studio2019_u3/itac/2019.3.0
↪ 32/lib
MIC_LIBRARY_PATH_modshare=/lrz/sys/intel/studio2019_
↪ u3/compilers_and_libraries_2019.3.199/linux/mkl/
↪ lib/mic:1
XDG_DATA_DIRS=/usr/share
MODULE_NORELOAD_modshare=lrz/1.0:1:admin/1.0:1:tempd
↪ ir/1.0:1
SSH_CONNECTION=131.188.39.70 58274 129.187.1.3 22
PLG_PATH_TMP=/var/ear
SLURM_JOB_ACCOUNT=ACCOUNT
I_MPI_HYDRA_BOOTSTRAP=slurm
MODULESHOME=/lrz/sys/share/modules
SLURM_JOB_NUM_NODES=1
OMP_NUM_THREADS=1
LESSOPEN=lessopen.sh %s
MPI_WWW=https://doku.lrz.de/display/PUBLIC/Intel+MPI
INSPECTOR_XE_LIBDIR=/lrz/sys/intel/studio2019_u3/ins
↪ pector_2019.3.0.591484/lib64
LOADEDMODULES_modshare=gcc/7:1:intel/19.0:1:lrz/defa
↪ ult:1:spack/staging/19.1:1:inspector_xe/2019:1:a
↪ dmin/1.0:1:advisor_xe/2019:1:amplifier_xe/2019:1
↪ :tempdir/1.0:1:mpi.intel/2019:1:itac/2019:1:devE
↪ nv/Intel/2019:1:cmake/3.6:1:mkl/2019:1
INFOPATH_modshare=/lrz/sys/intel/studio2019_u3/docum
↪ entation_2019/en/debugger/gdb-mic/info:1:/lrz/sy
↪ s/compilers/gcc/7.3.0/info:1:/lrz/sys/intel/stud
↪ io2019_u3/documentation_2019/en/debugger/gdb-ia/
↪ info:1:/lrz/sys/intel/studio2019_u3/documentatio
↪ n_2019/en/debugger/gdb-igfx/info:1
INFOPATH=/lrz/sys/compilers/gcc/7.3.0/info:/lrz/sys/
↪ intel/studio2019_u3/documentation_2019/en/debugg
↪ er/gdb-ia/info:/lrz/sys/intel/studio2019_u3/docu
↪ mentation_2019/en/debugger/gdb-mic/info:/lrz/sys
↪ /intel/studio2019_u3/documentation_2019/en/debug
↪ ger/gdb-igfx/info
CC=icc
CMAKE_INSTALL_DOC=/lrz/noarch/src/tools/cmake/README
↪ _lrz
ADVISOR_XE_BASE=/lrz/sys/intel/studio2019_u3/advisor
↪ _2019.3.0.591490
DEVENV_WWW=https://doku.lrz.de/display/PUBLIC/Softwa
↪ re+for+HPC
AMPLIFIER_XE_SRC=/lrz/noarch/src/intel/tools
ADVISOR_XE_LIBDIR=/lrz/sys/intel/studio2019_u3/advis
↪ or_2019.3.0.591490/lib64
INTEL_WWW=http://www.lrz-muenchen.de/services/softwa
↪ re/programmierung/intel_compilers/
MPI_MT_LIB=-L/lrz/sys/intel/studio2019_u3/impi/2019.
↪ 3.199/intel64/lib/release_mt -Xlinker
↪ --enable-new-dtags -Xlinker -rpath -Xlinker
↪ /lrz/sys/intel/studio2019_u3/impi/2019.3.199/int
↪ el64/lib/release_mt -lmpi -ldl -lrt
↪ -lpthread
LRZ_INSTRSET=x86_avx512
LESSCLOSE=lessclose.sh %s %s
CMAKE_DOC=/lrz/sys/tools/cmake/3.6.3/doc
G_BROKEN_FILENAMES=1
SCRATCH=/hppfs/scratch/05/USER
EAR_PLUGIN=1
SLURM_MEM_PER_NODE=81920
I_MPI_ROOT=/lrz/sys/intel/studio2019_u3/impi/2019.3.
↪ 199
MKL_SHLIB=-L/lrz/sys/intel/studio2019_u3/compilers_a
↪ nd_libraries_2019.3.199/linux/mkl/lib/intel64
↪ -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core
↪ -liomp5 -lpthread -lm
COLORTERM=1
BASH_FUNC_moduleraw%%=() {  eval `/usr/bin/tclsh
↪ $MODULE_BIN/modulecmd.tcl bash $*`
}
BASH_FUNC_module%%=() {  moduleraw $* 2>&1
}
+ lsb_release -a
LSB Version:    core-5.0-amd64:core-5.0-noarch
Distributor ID: SUSE
Description:    SUSE Linux Enterprise Server 12 SP3
Release:        12.3
Codename:       n/a
+ uname -a
Linux i01r02c02s02.sng.lrz.de 4.4.126-94.22-default
↪ #1 SMP Wed Apr 11 07:45:03 UTC 2018 (9649989)
↪ x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-95
Thread(s) per core:    2
Core(s) per socket:    24
Socket(s):             2
NUMA node(s):          2
```

```
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8174
↪  CPU @ 3.10GHz
Stepping:              4
CPU MHz:               2300.000
CPU max MHz:           2701.0000
CPU min MHz:           1200.0000
BogoMIPS:              5387.37
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              33792K
NUMA node0 CPU(s):     0-23,48-71
NUMA node1 CPU(s):     24-47,72-95
Flags:                 fpu vme de pse tsc msr pae mce
↪  cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪  dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
↪  pdpe1gb rdtscp lm constant_tsc art arch_perfmon
↪  pebs bts rep_good nopl xtopology nonstop_tsc
↪  aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
↪  ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
↪  pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
↪  tsc_deadline_timer aes xsave avx f16c rdrand
↪  lahf_lm abm 3dnowprefetch ida arat epb
↪  invpcid_single pln pts dtherm intel_pt rsb_ctxsw
↪  spec_ctrl stibp retpoline kaiser tpr_shadow vnmi
↪  flexpriority ept vpid fsgsbase tsc_adjust bmi1
↪  hle avx2 smep bmi2 erms invpcid rtm cqm mpx
↪  avx512f avx512dq rdseed adx smap clflushopt clwb
↪  avx512cd avx512bw avx512vl xsaveopt xsavec
↪  xgetbv1 cqm_llc cqm_occup_llc pku ospke
+ cat /proc/meminfo
MemTotal:       98434112 kB
MemFree:        90273348 kB
MemAvailable:   89809124 kB
Buffers:               0 kB
Cached:          1522524 kB
SwapCached:            0 kB
Active:           646212 kB
Inactive:        1454692 kB
Active(anon):     646212 kB
Inactive(anon):  1454692 kB
Active(file):          0 kB
Inactive(file):        0 kB
Unevictable:     4194320 kB
Mlocked:         4194320 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:       4772816 kB
Mapped:           240452 kB
Shmem:           1522524 kB
Slab:             526928 kB
SReclaimable:      39344 kB
SUnreclaim:       487584 kB
KernelStack:       34640 kB
PageTables:        17024 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    49217056 kB
Committed_AS:    6464700 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:   4532224 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:     1370496 kB
DirectMap2M:    73797632 kB
DirectMap1G:    27262976 kB
+ inxi -F -c0
collect_environment.sh: line 14: inxi: command not
↪  found
+ lsblk -a
+ lsscsi -s
collect_environment.sh: line 16: lsscsi: command not
↪  found
+ module list
+ moduleraw list
++ /usr/bin/tclsh
↪  /lrz/sys/share/modules/bin/modulecmd.tcl bash
↪  list
Currently Loaded Modulefiles:
 1) admin/1.0            6) mkl/2019           11)
 ↪  inspector_xe/2019
 2) tempdir/1.0          7) mpi.intel/2019     12)
 ↪  devEnv/Intel/2019
 3) spack/staging/19.1   8) itac/2019          13)
 ↪  gcc/7
 4) lrz/default          9) advisor_xe/2019    14)
 ↪  cmake/3.6
 5) intel/19.0          10) amplifier_xe/2019
+ eval
+ nvidia-smi
collect_environment.sh: line 18: nvidia-smi: command
↪  not found
+ lshw -short -quiet -sanitize
+ cat
collect_environment.sh: line 19: lshw: command not
↪  found
+ lspci
collect_environment.sh: line 19: lspci: command not
↪  found
```

```
Pitz Daint:
SLURM_CHECKPOINT_IMAGE_DIR=/var/slurm/checkpoint
SLURM_NODELIST=nid04277
LESSKEY=/etc/lesskey.bin
MODULE_VERSION_STACK=3.2.10.6
KSH_AUTOLOAD=1
PE_LIBSCI_VOLATILE_PRGENV=CRAY GNU INTEL
PE_SMA_DEFAULT_PKGCONFIG_VARIABLES=PE_SMA_COMPFLAG_@
↪ prgenv@
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_mic_knl=160
EBEXTSLISTJUPYTER=tornado-5.0.2,pyzmq-17.0.0,jupyter
↪ _core-4.4.0,jupyter_client-5.2.3,ipykernel-4.8.2
↪ ,webencodings-0.5.1,html5lib-1.0.1,bleach-2.1.3,
↪ jsonschema-2.6.0,nbformat-4.4.0,MarkupSafe-1.0,J
↪ inja2-2.10,testpath-0.3.1,entrypoints-0.2.3,pand
↪ ocfilters-1.4.2,nbconvert-5.3.1,qtconsole-4.3.1,
↪ terminado-0.8.1,Send2Trash-1.5.0,notebook-5.5.0,
↪ widgetsnbextension-3.2.1,ipywidgets-7.2.1,jupyte
↪ r_console-5.2.0,jupyter-1.0.0
EBVERSIONCONFIGURABLEMINHTTPMINPROXY=3.1.1
CRAY_CUDATOOLKIT_VERSION=9.1.85_3.18-6.0.7.0_5.1__g2
↪ eb7c52
EBVERSIONZLIB=1.2.11
SLURM_JOB_NAME=gpu_test
MANPATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybui
↪ ld/software/zlib/1.2.11-CrayGNU-18.08/share/man:
↪ /apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/s
↪ oftware/bzip2/1.0.6-CrayGNU-18.08/man:/opt/nvidi
↪ a/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c
↪ 52/doc/man:/opt/cray/pe/libsci_acc/18.07.1/man:/
↪ apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/so
↪ ftware/nodejs/8.9.4-CrayGNU-18.08/share/man:/app
↪ s/daint/UES/jenkins/6.0.UP07/gpu/easybuild/softw
↪ are/IPython/5.7.0-CrayGNU-18.08-python3/share/ma
↪ n:/opt/python/3.6.5.1/share/man:/opt/cray/pe/per
↪ ftools/7.0.3/man:/opt/cray/pe/papi/5.6.0.3/share
↪ /pdoc/man:/opt/cray/pe/atp/2.1.2/man:/opt/cray/a
↪ lps/6.6.43-6.0.7.0_26.4__ga796da3.ari/man:/opt/c
↪ ray/job/2.2.3-6.0.7.0_44.1__g6c4e934.ari/man:/op
↪ t/cray/pe/pmi/5.0.14/man:/opt/cray/pe/libsci/18.
↪ 07.1/man:/opt/cray/pe/man/csmlversion:/opt/cray/
↪ pe/craype/2.5.15/man:/opt/gcc/6.2.0/snos/share/m
↪ an:/opt/slurm/17.11.12.cscs/share/man:/opt/cray/
↪ pe/mpt/7.7.2/gni/man/mpich:/opt/cray/pe/modules/
↪ 3.2.10.6/share/man:/opt/slurm/default/share/man:
↪ /usr/local/man:/usr/share/man:/opt/cray/share/ma
↪ n:/opt/cray/pe/man
NNTPSERVER=news
PE_PAPI_DEFAULT_ACCEL_FAMILY_LIBS_nvidia=,-lcupti,-l
↪ cudart,-lcuda
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
PE_PETSC_DEFAULT_GENCOMPS_CRAY_skylake=86
PE_TPSL_DEFAULT_GENCOMPS_INTEL_x86_skylake=160
PE_CXX_PKGCONFIG_LIBS=mpichcxx
PE_MPICH_GENCOMPILERS_PGI=15.3
EBDEVELJUPYTERLAB=/apps/daint/UES/jenkins/6.0.UP07/g
↪ pu/easybuild/software/jupyterlab/0.35.2-CrayGNU-
↪ 18.08/easybuild/jupyterlab-0.35.2-CrayGNU-18.08-
↪ easybuild-devel
GNU_VERSION=6.2.0
PE_LIBSCI_ACC_MODULE_NAME=cray-libsci_acc/18.07.1
PE_LIBSCI_ACC_NV_SUFFIX_nvidia20=nv20
XDG_SESSION_ID=2407
XALT_ETC_DIR=/apps/daint/UES/xalt/0.7.6/etc
PE_HDF5_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
CRAY_LIBSCI_ACC_BASE_DIR=/opt/cray/pe/libsci_acc/18.
↪ 07.1
SLURMD_NODENAME=nid04277
SLURM_TOPOLOGY_ADDR=s21.s11.nid04277
SLURM_NTASKS_PER_NODE=1
HOSTNAME=nid04277
CRAY_UDREG_INCLUDE_OPTS=-I/opt/cray/udreg/2.3.2-6.0.
↪ 7.0_33.18__g5196236.ari/include
PE_FFTW_DEFAULT_TARGET_mic_knl=mic_knl
PE_LIBSCI_ACC_DEFAULT_PKGCONFIG_VARIABLES=PE_LIBSCI_
↪ ACC_DEFAULT_NV_SUFFIX_@accelerator@
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_interlagos=160
PE_TRILINOS_DEFAULT_GENCOMPS_CRAY_x86_64=86
RCLOCAL_BASEOPTS=true
SLURM_PRIO_PROCESS=0
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
CRAY_SITE_LIST_DIR=/etc/opt/cray/pe/modules
LIBRARYMODULES=acml:alps:cray-dwarf:cray-fftw:cray-g
↪ a:cray-hdf5:cray-hdf5-parallel:cray-libsci:cray-
↪ libsci_acc:cray-mpich:cray-mpich2:cray-mpich-abi
↪ :cray-netcdf:cray-netcdf-hdf5parallel:cray-paral
↪ lel-netcdf:cray-petsc:cray-petsc-complex:cray-sh
↪ mem:cray-tpsl:cray-trilinos:cudatoolkit:fftw:ga:
↪ hdf5:hdf5-parallel:iobuf:libfast:netcdf:netcdf-h
↪ df5parallel:ntk:onesided:papi:petsc:petsc-comple
↪ x:pmi:tpsl:trilinos:xt-libsci:xt-mpich2:xt-mpt:x
↪ t-papi
PE_NETCDF_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/
↪ pe/netcdf/4.6.1.2/@PRGENV@/@PE_NETCDF_DEFAULT_GE
↪ NCOMPS@/lib/pkgconfig
PE_PARALLEL_NETCDF_DEFAULT_VOLATILE_PKGCONFIG_PATH=/
↪ opt/cray/pe/parallel-netcdf/1.8.1.3/@PRGENV@/@PE
↪ _PARALLEL_NETCDF_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_SMA_DEFAULT_COMPFLAG_GNU=-fcray-pointer
PE_TRILINOS_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cra
↪ y/pe/trilinos/12.12.1.0/@PRGENV@/@PE_TRILINOS_DE
↪ FAULT_GENCOMPS@/@PE_TRILINOS_DEFAULT_TARGET@/lib
↪ /pkgconfig
SLURM_SRUN_COMM_PORT=41155
PE_ENV=GNU
PE_HDF5_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
PE_MPICH_ALTERNATE_LIBS_dpm=_dpm
PE_SMA_DEFAULT_COMPFLAG=
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
```

```
SHELL=/usr/local/bin/bash
TERM=xterm-256color
HOST=daint105
PE_TPSL_DEFAULT_GENCOMPS_CRAY_x86_skylake=86
PKGCONFIG_ENABLED=1
EBROOTPYEXTENSIONS=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/PyExtensions/3.6.5.1-CrayGNU-18.08
SLURM_JOB_QOS=daint_debug
HISTSIZE=1000
PROJECT=/project/ACCOUNT/USER
PROFILEREAD=true
PE_PETSC_DEFAULT_GENCOMPS_CRAY_sandybridge=86
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_x86_skylake=7.1 6.1
TMPDIR=/tmp
SLURM_TOPOLOGY_ADDR_PATTERN=switch.switch.node
SSH_CLIENT=148.187.1.11 41298 22
CRAYPE_DIR=/opt/cray/pe/craype/2.5.15
CRAY_UGNI_POST_LINK_OPTS=-L/opt/cray/ugni/6.0.14.0-6.0.7.0_23.1__gea11d3d.ari/lib64
CRAY_XPMEM_POST_LINK_OPTS=-L/opt/cray/xpmem/2.2.15-6.0.7.1_5.10__g7549d06.ari/lib64
PE_NETCDF_DEFAULT_VOLATILE_PRGENV=GNU
PE_PARALLEL_NETCDF_DEFAULT_VOLATILE_PRGENV=GNU
PE_PETSC_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_INTEL_haswell=160
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_x86_skylake=160
PE_TPSL_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_TPSL_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
PE_TRILINOS_DEFAULT_VOLATILE_PRGENV=CRAY GNU INTEL
PE_MPICH_DIR_PGI_DEFAULT64=64
SLURM_CSCS=yes
EBROOTCRAYGNU=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/CrayGNU/18.08
PERL5LIB=/opt/slurm/17.11.12.cscs//lib/perl5/site_perl/5.18.2/x86_64-linux-thread-multi:/opt/slurm/default/lib/perl5/site_perl/5.18.2/x86_64-linux-thread-multi:
PE_FFTW_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/fftw/3.3.6.5/@PE_FFTW_DEFAULT_TARGET@/lib/pkgconfig
PE_HDF5_DEFAULT_VOLATILE_PRGENV=GNU
PE_HDF5_PARALLEL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/hdf5-parallel/1.10.2.0/@PRGENV@/@PE_HDF5_PARALLEL_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_NETCDF_HDF5PARALLEL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/netcdf-hdf5parallel/4.6.1.2/@PRGENV@/@PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPS@/lib/pkgconfig
PE_PETSC_DEFAULT_GENCOMPS_CRAY_interlagos=86
CRAY_MPICH2_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1
ALT_LINKER=/apps/daint/UES/xalt/0.7.6/bin/ld
LIBRARY_PATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/Boost/1.67.0-CrayGNU-18.08-python3/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/zlib/1.2.11-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/bzip2/1.0.6-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/jupyterlab/0.35.2-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/jupyterhub/0.9.4-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/configurable-http-proxy/3.1.1-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/nodejs/8.9.4-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/jupyter/1.0.0-CrayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/IPython/5.7.0-CrayGNU-18.08-python3/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/software/PyExtensions/3.6.5.1-CrayGNU-18.08/lib
PMI_CONTROL_PORT=20447
PE_GA_DEFAULT_VOLATILE_PRGENV=GNU
PE_LIBSCI_DEFAULT_GENCOMPS_GNU_x86_64=71 61 51 49
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_TPSL_DEFAULT_GENCOMPS_CRAY_mic_knl=86
CRAY_CUDATOOLKIT_POST_LINK_OPTS=-L/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/lib64 -L/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/extras/CUPTI/lib64 -Wl,--as-needed -Wl,-lcupti -Wl,-lcudart -Wl,--no-as-needed -L/opt/cray/nvidia/default/lib64 -lcuda
CRAY_LIBSCI_ACC_DIR=/opt/cray/pe/libsci_acc/18.07.1
SLURM_CPU_BIND_VERBOSE=quiet
MORE=-sl
FPATH=:/opt/cray/pe/modules/3.2.10.6/init/sh_funcs/no_redirect:/opt/cray/pe/modules/3.2.10.6/init/sh_funcs/no_redirect
PERFTOOLS_VERSION=7.0.3
PE_LIBSCI_ACC_DEFAULT_GENCOMPS_CRAY_x86_64=85
PE_LIBSCI_ACC_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
PE_MPICH_DEFAULT_GENCOMPILERS_GNU=7.1 5.1 4.9
PE_PKGCONFIG_PRODUCTS=PE_LIBSCI_ACC:PE_LIBSCI:PE_MPICH
PE_TPSL_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_MPICH_GENCOMPS_GNU=71 51 49
EBVERSIONJUPYTERHUB=0.9.4
EBVERSIONJUPYTERLAB=0.35.2
PE_PAPI_DEFAULT_ACCEL_LIBS_nvidia35=,-lcupti,-lcudart,-lcuda
PE_PETSC_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI:PE_HDF5_PARALLEL:PE_TPSL
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_haswell=86
```

```
PE_TPSL_64_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray
↪ /pe/tpsl/18.06.1/@PRGENV@64/@PE_TPSL_64_DEFAULT_
↪ GENCOMPS@/@PE_TPSL_64_DEFAULT_TARGET@/lib/pkgcon
↪ fig
SLURM_JOB_GPUS=0
PE_CRAY_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/pa
↪ rallel-netcdf/1.8.1.3/CRAY/8.6/lib/pkgconfig:/op
↪ t/cray/pe/netcdf-hdf5parallel/4.6.1.2/CRAY/8.6/l
↪ ib/pkgconfig:/opt/cray/pe/netcdf/4.6.1.2/CRAY/8.
↪ 6/lib/pkgconfig:/opt/cray/pe/hdf5-parallel/1.10.
↪ 2.0/CRAY/8.6/lib/pkgconfig:/opt/cray/pe/hdf5/1.1
↪ 0.2.0/CRAY/8.6/lib/pkgconfig:/opt/cray/pe/ga/5.3
↪ .0.8/CRAY/8.6/lib/pkgconfig
PE_TRILINOS_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_LIBSCI_ACC_NV_SUFFIX_nvidia60=nv60
SLURM_SPANK_SHIFTER_GID=32035
SSH_TTY=/dev/pts/32
PE_LIBSCI_DEFAULT_OMP_REQUIRES_openmp=_mp
PE_PETSC_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
PE_FORTRAN_PKGCONFIG_LIBS=mpichf90
EBEXTSLISTPYEXTENSIONS=Cython-0.28.4,six-1.11.0,matp
↪ lotlib-2.2.2,pandas-0.23.3,Mako-1.0.8
EBVERSIONJUPYTER=1.0.0
SLURM_CPU_BIND_LIST=0xFFFFFF
MPICH_G2G_PIPELINE=256
CRAYPAT_ALPS_COMPONENT=/opt/cray/pe/perftools/7.0.3/
↪ sbin/pat_alps
CRAYPAT_LD_LIBRARY_PATH=/opt/cray/pe/gcc-libs:/opt/c
↪ ray/gcc-libs:/opt/cray/pe/perftools/7.0.3/lib64
PE_SMA_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/
↪ mpt/7.7.2/gni/sma@PE_SMA_DEFAULT_DIR_DEFAULT64@/
↪ lib64/pkgconfig
ALLINEA_QUEUE_DLL=/opt/cray/pe/mpt/7.7.2/gni/mpich-g
↪ nu/7.1/lib/libtvmpich.so.3.0.1
MPICH_RDMA_ENABLED_CUDA=1
ALPS_APP_ID=12837021
PE_LIBSCI_ACC_DEFAULT_VOLATILE_PRGENV=CRAY GNU
PE_TRILINOS_DEFAULT_GENCOMPS_INTEL_x86_64=160
CRAY_MPICH_BASEDIR=/opt/cray/pe/mpt/7.7.2/gni
EBROOTNODEJS=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/nodejs/8.9.4-CrayGNU-18.08
SLURM_NNODES=1
USER=USER
JRE_HOME=/usr/lib64/jvm/java/jre
PE_HDF5_PARALLEL_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3
↪ 4.9
PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPILERS_GNU=7.1
↪ 6.1 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_x86_skylake=86
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_haswell=160
SLURM_LOG_ACTIONS=yes
LS_COLORS=no=00:fi=00:di=01;34:ln=00;36:pi=40;33:so=
↪ 01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=41;33;
↪ 01:ex=00;32:*.cmd=00;32:*.exe=01;32:*.com=01;32:
↪ *.bat=01;32:*.btm=01;32:*.dll=01;32:*.tar=00;31:
↪ *.tbz=00;31:*.tgz=00;31:*.rpm=00;31:*.deb=00;31:
↪ *.arj=00;31:*.taz=00;31:*.lzh=00;31:*.lzma=00;31
↪ :*.zip=00;31:*.zoo=00;31:*.z=00;31:*.Z=00;31:*.g
↪ z=00;31:*.bz2=00;31:*.tb2=00;31:*.tz2=00;31:*.tb
↪ z2=00;31:*.xz=00;31:*.avi=01;35:*.bmp=01;35:*.fl
↪ i=01;35:*.gif=01;35:*.jpg=01;35:*.jpeg=01;35:*.m
↪ ng=01;35:*.mov=01;35:*.mpg=01;35:*.pcx=01;35:*.p
↪ bm=01;35:*.pgm=01;35:*.png=01;35:*.ppm=01;35:*.t
↪ ga=01;35:*.tif=01;35:*.xbm=01;35:*.xpm=01;35:*.d
↪ l=01;35:*.gl=01;35:*.wmv=01;35:*.aiff=00;32:*.au
↪ =00;32:*.mid=00;32:*.mp3=00;32:*.ogg=00;32:*.voc
↪ =00;32:*.wav=00;32:
LD_LIBRARY_PATH=/opt/nvidia/cudatoolkit9.1/9.1.85_3.
↪ 18-6.0.7.0_5.1__g2eb7c52/lib64:/opt/nvidia/cudat
↪ oolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/extr
↪ as/CUPTI/lib64:/opt/cray/pe/libsci_acc/18.07.1/G
↪ NU/4.9/x86_64/lib:/opt/cray/pe/mpt/7.7.2/gni/mpi
↪ ch-gnu/7.1/lib:/opt/cray/pe/perftools/7.0.3/lib6
↪ 4:/opt/cray/rca/2.2.18-6.0.7.0_33.3__g2aa4f39.ar
↪ i/lib64:/opt/cray/alps/6.6.43-6.0.7.0_26.4__ga79
↪ 6da3.ari/lib64:/opt/cray/xpmem/2.2.15-6.0.7.1_5.
↪ 10__g7549d06.ari/lib64:/opt/cray/dmapp/7.1.1-6.0
↪ .7.0_34.3__g5a674e0.ari/lib64:/opt/cray/pe/pmi/5
↪ .0.14/lib64:/opt/cray/ugni/6.0.14.0-6.0.7.0_23.1
↪ __gea11d3d.ari/lib64:/opt/cray/udreg/2.3.2-6.0.7
↪ .0_33.18__g5196236.ari/lib64:/opt/cray/pe/libsci
↪ /18.07.1/GNU/6.1/x86_64/lib:/apps/daint/UES/jenk
↪ ins/6.0.UP07/gpu/easybuild/software/Boost/1.67.0
↪ -CrayGNU-18.08-python3/lib:/apps/daint/UES/jenkin
↪ s/6.0.UP07/gpu/easybuild/software/zlib/1.2.11-Cr
↪ ayGNU-18.08/lib:/apps/daint/UES/jenkins/6.0.UP07
↪ /gpu/easybuild/software/bzip2/1.0.6-CrayGNU-18.0
↪ 8/lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easyb
↪ uild/software/jupyterlab/0.35.2-CrayGNU-18.08/li
↪ b:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild
↪ /software/jupyterhub/0.9.4-CrayGNU-18.08/lib:/ap
↪ ps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/soft
↪ ware/configurable-http-proxy/3.1.1-CrayGNU-18.08
↪ /lib:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybu
↪ ild/software/nodejs/8.9.4-CrayGNU-18.08/lib:/app
↪ s/daint/UES/jenkins/6.0.UP07/gpu/easybuild/softw
↪ are/jupyter/1.0.0-CrayGNU-18.08/lib:/apps/daint/
↪ UES/jenkins/6.0.UP07/gpu/easybuild/software/IPyt
↪ hon/5.7.0-CrayGNU-18.08-python3/lib:/apps/daint/
↪ UES/jenkins/6.0.UP07/gpu/easybuild/software/PyEx
↪ tensions/3.6.5.1-CrayGNU-18.08/lib:/opt/cray/pe/
↪ gcc-libs:/opt/cray/pe/papi/5.6.0.3/lib64:/opt/cr
↪ ay/job/2.2.3-6.0.7.0_44.1__g6c4e934.ari/lib64:/o
↪ pt/gcc/6.2.0/snos/lib64
PE_FFTW_DEFAULT_TARGET_interlagos=interlagos
PE_LIBSCI_DEFAULT_VOLATILE_PRGENV=CRAY GNU INTEL
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
```

```
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_TPSL_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_TRILINOS_DEFAULT_GENCOMPILERS_GNU_x86_64=71 53 49
PE_TRILINOS_DEFAULT_GENCOMPILERS_INTEL_x86_64=160
EBDEVELCONFIGURABLEMINHTTPMINPROXY=/apps/daint/UES/j
↪ enkins/6.0.UP07/gpu/easybuild/software/configura
↪ ble-http-proxy/3.1.1-CrayGNU-18.08/easybuild/con
↪ figurable-http-proxy-.3.1.1-CrayGNU-18.08-easybu
↪ ild-devel
EBDEVELPYEXTENSIONS=/apps/daint/UES/jenkins/6.0.UP07
↪ /gpu/easybuild/software/PyExtensions/3.6.5.1-Cra
↪ yGNU-18.08/easybuild/PyExtensions-3.6.5.1-CrayGN
↪ U-18.08-easybuild-devel
EBVERSIONNODEJS=8.9.4
CRAYPE_LINK_TYPE=dynamic
PE_LIBSCI_ACC_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/l
↪ ibsci_acc/18.07.1/@PRGENV@/@PE_LIBSCI_ACC_GENCOM
↪ PS@/@PE_LIBSCI_ACC_TARGET@/lib/pkgconfig
EBDEVELZLIB=/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/software/zlib/1.2.11-CrayGNU-18.08/easybu
↪ ild/zlib-.1.2.11-CrayGNU-18.08-easybuild-devel
CRAY_IAA_INFO_FILE=/tmp/cray_iaa_info.12837021
SINFO_FORMAT=%9P %5a %8s %.10l %.6c %.6z %.7D %10T %N
CRAY_RCA_POST_LINK_OPTS=-L/opt/cray/rca/2.2.18-6.0.7
↪ .0_33.3__g2aa4f39.ari/lib64
↪ -lrca
PE_LIBSCI_PKGCONFIG_VARIABLES=PE_LIBSCI_OMP_REQUIRES
↪ _@openmp@:PE_SCI_EXT_LIBPATH:PE_SCI_EXT_LIBNAME
PE_PETSC_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU
↪ GNU64 INTEL INTEL64
PE_PKGCONFIG_LIBS=cray-cudatoolkit:libsci_acc:AtpSig
↪ Handler:cray-rca:libsci_mpi:libsci:mpich
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1
↪ 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
PE_MPICH_FIXED_PRGENV=INTEL
XNLSPATH=/usr/share/X11/nls
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_skylake=16.0
PE_PETSC_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_INTEL_interlagos=160
PE_PETSC_DEFAULT_GENCOMPS_INTEL_sandybridge=160
PE_TPSL_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
EBDEVELJUPYTER=/apps/daint/UES/jenkins/6.0.UP07/gpu/
↪ easybuild/software/jupyter/1.0.0-CrayGNU-18.08/e
↪ asybuild/jupyter-1.0.0-CrayGNU-18.08-easybuild-d
↪ evel
CRAY_ACCEL_TARGET=nvidia60
LIBSCI_ACC_EXAMPLES_DIR=/opt/cray/pe/libsci_acc/18.0
↪ 7.1/examples
SLURM_STEP_NUM_NODES=1
MPICH_ABORT_ON_ERROR=1
PE_LIBSCI_DEFAULT_GENCOMPS_CRAY_x86_64=86
PE_PAPI_DEFAULT_PKGCONFIG_VARIABLES=PE_PAPI_ACCEL_LI
↪ BS_@accelerator@
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
PE_PETSC_DEFAULT_GENCOMPS_GNU_mic_knl=53
PE_PETSC_DEFAULT_GENCOMPS_INTEL_mic_knl=160
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1
↪ 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_sandybridge=160
MPICH_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/7.1
SRUN_DEBUG=3
CRAY_CUDA_MPS=1
SLURM_JOBID=12837021
SSH_AUTH_SOCK=/tmp/ssh-BGSHB3Fhjt/agent.30331
HOSTTYPE=x86_64
ATP_POST_LINK_OPTS=-Wl,-L/opt/cray/pe/atp/2.1.2/libA
↪ pp/
PE_FFTW_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
PE_FFTW_DEFAULT_TARGET_sandybridge=sandybridge
PE_HDF5_PARALLEL_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
PE_NETCDF_HDF5PARALLEL_DEFAULT_REQUIRED_PRODUCTS=PE_
↪ HDF5_PARALLEL
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16.0
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
PE_MPICH_FORTRAN_PKGCONFIG_LIBS=mpichf90
CPATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild
↪ /software/Boost/1.67.0-CrayGNU-18.08-python3/inc
↪ lude:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybu
↪ ild/software/zlib/1.2.11-CrayGNU-18.08/include:/
↪ apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/so
↪ ftware/bzip2/1.0.6-CrayGNU-18.08/include:/apps/d
↪ aint/UES/jenkins/6.0.UP07/gpu/easybuild/software
↪ /nodejs/8.9.4-CrayGNU-18.08/include
CRAY_PRGENVGNU=loaded
EBDEVELBOOST=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/Boost/1.67.0-CrayGNU-18.08-pyth
↪ on3/easybuild/Boost-1.67.0-CrayGNU-18.08-python3
↪ -easybuild-devel
EBROOTCMAKE=/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/software/CMake/3.12.0
TMOUT=259200
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_mic_knl=5.3
RCLOCAL_PRGENV=true
SLURM_NTASKS=1
APPS=/apps/daint
FROM_HEADER=
CHPL_CG_CPP_LINES=1
OFFLOAD_INIT=on_start
PE_LIBSCI_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
PE_LIBSCI_GENCOMPS_INTEL_x86_64=160
PE_PRODUCT_LIST=CRAY_LIBSCI_ACC:CRAY_RCA:CRAY_ALPS:D
↪ VS:CRAY_XPMEM:CRAY_DMAPP:CRAY_PMI:CRAY_UGNI:CRAY
↪ _UDREG:CRAY_LIBSCI:CRAYPE:CRAYPE_HASWELL:GNU:GCC
↪ :PERFTOOLS:CRAYPAT:CRAY_ACCEL
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_TPSL_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
GCC_VERSION=6.2.0
```

```
gcc_already_loaded=0
SLURM_LAUNCH_NODE_IPADDR=148.187.48.214
ALPS_LLI_STATUS_OFFSET=1
PAGER=less
PE_MPICH_DEFAULT_GENCOMPS_PGI=153
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3 4.9
PE_TPSL_DEFAULT_GENCOMPS_GNU_x86_skylake=71 61
CRAY_MPICH_ROOTDIR=/opt/cray/pe/mpt/7.7.2
SLURM_STEP_ID=0
ALPS_APP_PE=0
CSHEDIT=emacs
PE_LIBSCI_GENCOMPILERS_GNU_x86_64=7.1 6.1 5.1 4.9
PE_PETSC_DEFAULT_GENCOMPS_GNU_skylake=61
PE_PETSC_DEFAULT_GENCOMPS_INTEL_skylake=160
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
PE_MPICH_GENCOMPILERS_CRAY=8.6
PE_MPICH_MODULE_NAME=cray-mpich
XDG_CONFIG_DIRS=/etc/xdg
CRAYPAT_ROOT=/opt/cray/pe/perftools/7.0.3
PE_LIBSCI_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.6
PE_LIBSCI_GENCOMPS_CRAY_x86_64=86
PE_MPICH_DEFAULT_VOLATILE_PRGENV=CRAY GNU PGI
PE_MPICH_TARGET_VAR_nvidia20=-lcudart
PE_TPSL_64_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_LIB
↪ SCI
PE_TPSL_DEFAULT_GENCOMPS_CRAY_haswell=86
PE_TPSL_DEFAULT_GENCOMPS_CRAY_sandybridge=86
JUPYTER_PATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/jupyter/1.0.0-CrayGNU-18.08
CRAY_LIBSCI_ACC_VERSION=18.07.1
MINICOM=-c on
LIBGL_DEBUG=quiet
USERMODULES=acml:alps:apprentice:apprentice2:atp:blc
↪ r:cce:chapel:cray-ccdb:cray-fftw:cray-ga:cray-hd
↪ f5:cray-hdf5-parallel:cray-lgdb:cray-libsci:cray
↪ -libsci_acc:cray-mpich:cray-mpich2:cray-mpich-com
↪ pat:cray-netcdf:cray-netcdf-hdf5parallel:cray-pa
↪ rallel-netcdf:craypat:craype:cray-petsc:cray-pet
↪ sc-complex:craypkg-gen:cray-shmem:cray-snplauncher
↪ er:cray-tpsl:cray-trilinos:cudatoolkit:ddt:fftw:
↪ ga:gcc:hdf5:hdf5-parallel:intel:iobuf:java:lgdb:
↪ libfast:libsci_acc:mpich1:netcdf:netcdf-hdf5para
↪ llel:netcdf-nofsync:netcdf-nofsync-hdf5parallel:
↪ ntk:onesided:papi:parallel-netcdf:pathscale:perf
↪ tools:perftools-lite:petsc:petsc-complex:pgi:pmi
↪ :PrgEnv-cray:PrgEnv-gnu:PrgEnv-intel:PrgEnv-path
↪ scale:PrgEnv-pgi:stat:totalview:tpsl:trilinos:xt
↪ -asyncpe:xt-craypat:xt-lgdb:xt-libsci:xt-mpich2:x
↪ t-mpt:xt-papi:xt-shmem:xt-totalview
CRAY_DMAPP_INCLUDE_OPTS=-I/opt/cray/dmapp/7.1.1-6.0.
↪ 7.0_34.3__g5a674e0.ari/include
↪ -I/opt/cray/gni-headers/5.0.12.0-6.0.7.0_24.1__g
↪ 3b1768f.ari/include
CRAY_LIBSCI_BASE_DIR=/opt/cray/pe/libsci/18.07.1
CRAY_LIBSCI_DIR=/opt/cray/pe/libsci/18.07.1
DVS_VERSION=0.9.0
PE_LIBSCI_ACC_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/c
↪ ray/pe/libsci_acc/18.07.1/@PRGENV@/@PE_LIBSCI_AC
↪ C_DEFAULT_GENCOMPS@/@PE_LIBSCI_ACC_DEFAULT_TARGE
↪ T@/lib/pkgconfig
PE_LIBSCI_PKGCONFIG_LIBS=libsci_mpi:libsci
PE_NETCDF_DEFAULT_GENCOMPS_GNU=
PE_PARALLEL_NETCDF_DEFAULT_GENCOMPS_GNU=51 49
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_mic_knl=71 53
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
EBDEVELIPYTHON=/apps/daint/UES/jenkins/6.0.UP07/gpu/
↪ easybuild/software/IPython/5.7.0-CrayGNU-18.08-p
↪ ython3/easybuild/IPython-.5.7.0-CrayGNU-18.08-py
↪ thon3-easybuild-devel
EBVERSIONPYEXTENSIONS=3.6.5.1
SLURM_STEP_LAUNCHER_PORT=41155
SLURM_TASKS_PER_NODE=1
PATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/
↪ software/CMake/3.12.0/bin:/apps/daint/UES/jenkin
↪ s/6.0.UP07/gpu/easybuild/software/bzip2/1.0.6-Cr
↪ ayGNU-18.08/bin:/opt/nvidia/cudatoolkit9.1/9.1.8
↪ 5_3.18-6.0.7.0_5.1__g2eb7c52/bin:/opt/nvidia/cud
↪ atoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/li
↪ bnvvp:/apps/daint/UES/jenkins/6.0.UP07/gpu/easyb
↪ uild/software/jupyterlab/0.35.2-CrayGNU-18.08/bi
↪ n:/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild
↪ /software/jupyterhub/0.9.4-CrayGNU-18.08/bin:/ap
↪ ps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/soft
↪ ware/configurable-http-proxy/3.1.1-CrayGNU-18.08
↪ :/apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/
↪ software/configurable-http-proxy/3.1.1-CrayGNU-1
↪ 8.08/bin:/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/nodejs/8.9.4-CrayGNU-18.08/bin:
↪ /apps/daint/UES/jenkins/6.0.UP07/gpu/easybuild/s
↪ oftware/jupyter/1.0.0-CrayGNU-18.08/bin:/apps/da
↪ int/UES/jenkins/6.0.UP07/gpu/easybuild/software/
↪ IPython/5.7.0-CrayGNU-18.08-python3/bin:/apps/da
↪ int/UES/jenkins/6.0.UP07/gpu/easybuild/software/
↪ PyExtensions/3.6.5.1-CrayGNU-18.08/bin:/opt/pyth
↪ on/3.6.5.1/bin:/opt/cray/pe/perftools/7.0.3/bin:
↪ /opt/cray/pe/papi/5.6.0.3/bin:/opt/cray/rca/2.2.
↪ 18-6.0.7.0_33.3__g2aa4f39.ari/bin:/opt/cray/alps
↪ /6.6.43-6.0.7.0_26.4__ga796da3.ari/sbin:/opt/cra
↪ y/job/2.2.3-6.0.7.0_44.1__g6c4e934.ari/bin:/opt/
↪ cray/pe/craype/2.5.15/bin:/opt/gcc/6.2.0/bin:/ap
↪ ps/daint/UES/xalt/0.7.6/bin:/opt/slurm/17.11.12.
↪ cscs/bin:/opt/cray/pe/mpt/7.7.2/gni/bin:/opt/cra
↪ y/pe/modules/3.2.10.6/bin:/opt/slurm/default/bin
↪ :/apps/daint/system/bin:/apps/common/system/bin:
↪ /users/USER/bin:/usr/local/bin:/usr/bin:/bin:/us
↪ r/bin/X11:/usr/lib/mit/bin:/usr/lib/mit/sbin:/op
↪ t/cray/pe/bin:/opt/cray/nvidia/default/bin
MAIL=/var/mail/USER
MODULE_VERSION=3.2.10.6
```

```
PAT_REPORT_PRUNE_NAME=_cray$mt_execute_,_cray$mt_sta
↪ rt_,__cray_hwpc_,f_cray_hwpc_,cstart,__pat_,pat_
↪ region_,PAT_,OMP.slave_loop,slave_entry,_new_sla
↪ ve_entry,_thread_pool_slave_entry,THREAD_POOL_jo
↪ in,__libc_start_main,_start,__start,start_thread
↪ ,__wrap_,UPC_ADIO_,_upc_,upc_,__caf_,__pgas_,sys
↪ call,__device_stub
PE_HDF5_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe
↪ /hdf5/1.10.2.0/@PRGENV@/@PE_HDF5_DEFAULT_GENCOMP
↪ S@/lib/pkgconfig
PE_PKGCONFIG_DEFAULT_PRODUCTS=PE_TRILINOS:PE_TPSL_64
↪ :PE_TPSL:PE_PETSC:PE_PARALLEL_NETCDF:PE_NETCDF_H
↪ DF5PARALLEL:PE_NETCDF:PE_MPICH:PE_LIBSCI_ACC:PE_
↪ LIBSCI:PE_HDF5_PARALLEL:PE_HDF5:PE_GA:PE_FFTW
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3 4.9
PE_TPSL_DEFAULT_GENCOMPS_CRAY_interlagos=86
PE_MPICH_GENCOMPILERS_GNU=7.1 5.1 4.9
EBROOTJUPYTERHUB=/apps/daint/UES/jenkins/6.0.UP07/gp
↪ u/easybuild/software/jupyterhub/0.9.4-CrayGNU-18
↪ .08
PE_LIBSCI_ACC_PKGCONFIG_VARIABLES=PE_LIBSCI_ACC_NV_S
↪ UFFIX_@accelerator@
PMI_CRAY_NO_SMP_ORDER=0
SLURM_WORKING_CLUSTER=daint:daintsl01:6817:8192
CPU=x86_64
CSCS_CUSTOM_ENV=true
XTPE_NETWORK_TARGET=aries
ATP_IGNORE_SIGTERM=1
PE_FFTW_DEFAULT_TARGET_abudhabi=abudhabi
PE_MPICH_DEFAULT_DIR_PGI_DEFAULT64=64
PE_NETCDF_DEFAULT_GENCOMPILERS_GNU=7.1 6.1 5.3 4.9
PE_PARALLEL_NETCDF_DEFAULT_GENCOMPILERS_GNU=5.1 4.9
PE_PETSC_DEFAULT_GENCOMPS_CRAY_mic_knl=86
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_x86_skylake=7.1
↪ 6.1
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3 4.9
EBROOTCONFIGURABLEMINHTTPMINPROXY=/apps/daint/UES/je
↪ nkins/6.0.UP07/gpu/easybuild/software/configurab
↪ le-http-proxy/3.1.1-CrayGNU-18.08
_=/usr/bin/env
PMI_NO_FORK=1
SLURM_JOB_ID=12837021
SSH_SENDS_LOCALE=yes
JAVA_BINDIR=/usr/lib64/jvm/java/bin
SQUEUE_SORT=-t,e,S
PE_HDF5_PARALLEL_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_HDF5_PARALLEL_DEFAULT_GENCOMPS_GNU=
PE_NETCDF_HDF5PARALLEL_DEFAULT_FIXED_PRGENV=CRAY PGI
↪ INTEL
PE_NETCDF_HDF5PARALLEL_DEFAULT_GENCOMPS_GNU=
PE_SMA_DEFAULT_DIR_CRAY_DEFAULT64=64
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_x86_skylake=8.6
LIBSCI_ACC_VERSION=18.07.1
SLURM_STEP_GPUS=0
CRAY_UDREG_POST_LINK_OPTS=-L/opt/cray/udreg/2.3.2-6.
↪ 0.7.0_33.18__g5196236.ari/lib64
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_sandybridge=86
PE_TPSL_64_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU
↪ GNU64 INTEL INTEL64
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
PE_TPSL_DEFAULT_GENCOMPS_INTEL_interlagos=160
EBVERSIONBOOST=1.67.0
SLURM_STEPID=0
SLURM_JOB_USER=USER
PWD=/users/USER/run
INPUTRC=/users/USER/.inputrc
CRAYPE_VERSION=2.5.15
CRAY_ALPS_POST_LINK_OPTS=-L/opt/cray/alps/6.6.43-6.0
↪ .7.0_26.4__ga796da3.ari/lib64
PE_TPSL_DEFAULT_GENCOMPS_GNU_mic_knl=71 53
PE_MPICH_VOLATILE_PRGENV=CRAY GNU PGI
PE_LIBSCI_ACC_GENCOMPS_CRAY_x86_64=85
SLURM_SRUN_COMM_HOST=148.187.48.214
CUDA_VISIBLE_DEVICES=0
JAVA_HOME=/usr/lib64/jvm/java
TARGETMODULES=craype-abudhabi:craype-abudhabi-cu:cra
↪ ype-accel-host:craype-accel-nvidia20:craype-acce
↪ l-nvidia30:craype-accel-nvidia35:craype-barcelon
↪ a:craype-broadwell:craype-haswell:craype-hugepag
↪ es128K:craype-hugepages128M:craype-hugepages16M:
↪ craype-hugepages256M:craype-hugepages2M:craype-h
↪ ugepages32M:craype-hugepages4M:craype-hugepages5
↪ 12K:craype-hugepages512M:craype-hugepages64M:cra
↪ ype-hugepages8M:craype-intel-knc:craype-interlag
↪ os:craype-interlagos-cu:craype-istanbul:craype-i
↪ vybridge:craype-mc12:craype-mc8:craype-mic-knl:c
↪ raype-network-aries:craype-network-gemini:craype
↪ -network-infiniband:craype-network-none:craype-ne
↪ twork-seastar:craype-sandybridge:craype-shanghai
↪ :craype-target-compute_node:craype-target-local_
↪ host:craype-target-native:craype-xeon:xtpe-barce
↪ lona:xtpe-interlagos:xtpe-interlagos-cu:xtpe-ist
↪ anbul:xtpe-mc12:xtpe-mc8:xtpe-network-gemini:xtp
↪ e-network-seastar:xtpe-shanghai:xtpe-target-nati
↪ ve:xtpe-xeon
```

```
_LMFILES_=/opt/cray/pe/modulefiles/modules/3.2.10.6:
↪ /opt/cray/pe/modulefiles/cray-mpich/7.7.2:/opt/m
↪ odulefiles/slurm/17.11.12.cscs-1:/apps/daint/UES
↪ /easybuild/modulefiles/xalt/daint-2016.11:/apps/
↪ daint/UES/easybuild/modulefiles/daint-gpu:/opt/m
↪ odulefiles/gcc/6.2.0:/opt/cray/pe/craype/2.5.15/
↪ modulefiles/craype-haswell:/opt/cray/pe/craype/2
↪ .5.15/modulefiles/craype-network-aries:/opt/cray
↪ /pe/modulefiles/craype/2.5.15:/opt/cray/pe/modul
↪ efiles/cray-libsci/18.07.1:/opt/cray/ari/modulef
↪ iles/udreg/2.3.2-6.0.7.0_33.18__g5196236.ari:/op
↪ t/cray/ari/modulefiles/ugni/6.0.14.0-6.0.7.0_23.
↪ 1__gea11d3d.ari:/opt/cray/pe/modulefiles/pmi/5.0
↪ .14:/opt/cray/ari/modulefiles/dmapp/7.1.1-6.0.7.
↪ 0_34.3__g5a674e0.ari:/opt/cray/ari/modulefiles/g
↪ ni-headers/5.0.12.0-6.0.7.0_24.1__g3b1768f.ari:/
↪ opt/cray/ari/modulefiles/xpmem/2.2.15-6.0.7.1_5.
↪ 10__g7549d06.ari:/opt/cray/ari/modulefiles/job/2
↪ .2.3-6.0.7.0_44.1__g6c4e934.ari:/opt/cray/ari/mo
↪ dulefiles/dvs/2.7_2.2.113-6.0.7.1_7.6__g1bbc03e:
↪ /opt/cray/ari/modulefiles/alps/6.6.43-6.0.7.0_26
↪ .4__ga796da3.ari:/opt/cray/ari/modulefiles/rca/2
↪ .2.18-6.0.7.0_33.3__g2aa4f39.ari:/opt/cray/pe/mo
↪ dulefiles/atp/2.1.2:/opt/cray/pe/modulefiles/per
↪ ftools-base/7.0.3:/opt/cray/pe/modulefiles/PrgEn
↪ v-gnu/6.0.4:/apps/daint/UES/jenkins/6.0.UP07/gpu
↪ /easybuild/modules/all/CrayGNU/.18.08:/opt/modul
↪ efiles/cray-python/3.6.5.1:/apps/daint/UES/jenki
↪ ns/6.0.UP07/gpu/easybuild/modules/all/PyExtensio
↪ ns/3.6.5.1-CrayGNU-18.08:/apps/daint/UES/jenkins
↪ /6.0.UP07/gpu/easybuild/modules/all/IPython/.5.7
↪ .0-CrayGNU-18.08-python3:/apps/daint/UES/jenkins
↪ /6.0.UP07/gpu/easybuild/modules/all/jupyter/1.0.
↪ 0-CrayGNU-18.08:/apps/daint/UES/jenkins/6.0.UP07
↪ /gpu/easybuild/modules/all/nodejs/.8.9.4-CrayGNU
↪ -18.08:/apps/daint/UES/jenkins/6.0.UP07/gpu/easyb
↪ uild/modules/all/configurable-http-proxy/.3.1.1-
↪ CrayGNU-18.08:/apps/daint/UES/jenkins/6.0.UP07/g
↪ pu/easybuild/modules/all/jupyterhub/0.9.4-CrayGN
↪ U-18.08:/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/modules/all/jupyterlab/0.35.2-CrayGNU-18.
↪ 08:/opt/cray/pe/modulefiles/cray-libsci_acc/18.0
↪ 7.1:/opt/cray/modulefiles/cudatoolkit/9.1.85_3.1
↪ 8-6.0.7.0_5.1__g2eb7c52:/opt/cray/pe/craype/2.5.
↪ 15/modulefiles/craype-accel-nvidia60:/apps/daint
↪ /UES/jenkins/6.0.UP07/gpu/easybuild/modules/all/
↪ bzip2/.1.0.6-CrayGNU-18.08:/apps/daint/UES/jenki
↪ ns/6.0.UP07/gpu/easybuild/modules/all/zlib/.1.2.
↪ 11-CrayGNU-18.08:/apps/daint/UES/jenkins/6.0.UP0
↪ 7/gpu/easybuild/modules/all/Boost/1.67.0-CrayGNU
↪ -18.08-python3:/apps/daint/UES/jenkins/6.0.UP07/g
↪ pu/easybuild/modules/all/CMake/.3.12.0:/opt/modu
↪ lefiles/Base-opts/2.4.135-6.0.7.0_38.1__g718f891
↪ .ari
PE_LIBSCI_DEFAULT_OMP_REQUIRES=
PE_MPICH_DEFAULT_GENCOMPS_CRAY=86
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1
↪ 5.3 4.9
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
XALT_TRANSMISSION_STYLE=directdb
SLURM_CPU_BIND_TYPE=mask_cpu:
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia20=nv20
PE_LIBSCI_MODULE_NAME=cray-libsci/18.07.1
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_skylake=8.6
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_mic_knl=7.1 5.3
LANG=en_US
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_x86_skylake=71 61
PE_INTEL_FIXED_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.7.2
↪ /gni/mpich-intel/16.0/lib/pkgconfig
EBDEVELCRAYGNU=/apps/daint/UES/jenkins/6.0.UP07/gpu/
↪ easybuild/software/CrayGNU/18.08/easybuild/CrayG
↪ NU-.18.08-easybuild-devel
SLURM_UMASK=0022
PYTHONSTARTUP=/etc/pythonstart
MODULEPATH=/opt/cray/pe/perftools/7.0.3/modulefiles:
↪ /opt/cray/pe/craype/2.5.15/modulefiles:/apps/dai
↪ nt/UES/jenkins/6.0.UP07/gpu/easybuild/tools/modu
↪ les/all:/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/modules/all:/apps/daint/modulefiles:/apps
↪ /daint/system/modulefiles:/apps/daint/UES/easybu
↪ ild/modulefiles:/apps/common/UES/modulefiles:/ap
↪ ps/common/system/modulefiles:/opt/cray/pe/module
↪ files:/opt/cray/modulefiles:/opt/modulefiles:/op
↪ t/cray/ari/modulefiles:/opt/cray/pe/ari/modulefi
↪ les
PE_LIBSCI_GENCOMPILERS_CRAY_x86_64=8.6
PE_MPICH_NV_LIBS_nvidia20=-lcudart
PE_MPICH_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/mpt/7.
↪ 7.2/gni/mpich-@PRGENV@@PE_MPICH_DIR_DEFAULT64@/@
↪ PE_MPICH_GENCOMPS@/lib/pkgconfig
EBROOTJUPYTERLAB=/apps/daint/UES/jenkins/6.0.UP07/gp
↪ u/easybuild/software/jupyterlab/0.35.2-CrayGNU-1
↪ 8.08
SLURM_JOB_UID=23945
SDK_HOME=/usr/lib64/jvm/java
TZ=Europe/Zurich
```

```
LOADEDMODULES=modules/3.2.10.6:cray-mpich/7.7.2:slur
↪ m/17.11.12.cscs-1:xalt/daint-2016.11:daint-gpu:g
↪ cc/6.2.0:craype-haswell:craype-network-aries:cra
↪ ype/2.5.15:cray-libsci/18.07.1:udreg/2.3.2-6.0.7
↪ .0_33.18__g5196236.ari:ugni/6.0.14.0-6.0.7.0_23.
↪ 1__gea11d3d.ari:pmi/5.0.14:dmapp/7.1.1-6.0.7.0_3
↪ 4.3__g5a674e0.ari:gni-headers/5.0.12.0-6.0.7.0_2
↪ 4.1__g3b1768f.ari:xpmem/2.2.15-6.0.7.1_5.10__g75
↪ 49d06.ari:job/2.2.3-6.0.7.0_44.1__g6c4e934.ari:d
↪ vs/2.7_2.2.113-6.0.7.1_7.6__g1bbc03e:alps/6.6.43
↪ -6.0.7.0_26.4__ga796da3.ari:rca/2.2.18-6.0.7.0_33
↪ .3__g2aa4f39.ari:atp/2.1.2:perftools-base/7.0.3:
↪ PrgEnv-gnu/6.0.4:CrayGNU/.18.08:cray-python/3.6.
↪ 5.1:PyExtensions/3.6.5.1-CrayGNU-18.08:IPython/.
↪ 5.7.0-CrayGNU-18.08-python3:jupyter/1.0.0-CrayGN
↪ U-18.08:nodejs/.8.9.4-CrayGNU-18.08:configurable
↪ -http-proxy/.3.1.1-CrayGNU-18.08:jupyterhub/0.9.4
↪ -CrayGNU-18.08:jupyterlab/0.35.2-CrayGNU-18.08:cr
↪ ay-libsci_acc/18.07.1:cudatoolkit/9.1.85_3.18-6.
↪ 0.7.0_5.1__g2eb7c52:craype-accel-nvidia60:bzip2/
↪ .1.0.6-CrayGNU-18.08:zlib/.1.2.11-CrayGNU-18.08:
↪ Boost/1.67.0-CrayGNU-18.08-python3:CMake/.3.12.0
↪ :Base-opts/2.4.135-6.0.7.0_38.1__g718f891.ari
SHMEM_ABORT_ON_ERROR=1
EBVERSIONCRAYGNU=18.08
LIBSCI_ACC_BASE_DIR=/opt/cray/pe/libsci_acc/18.07.1
PE_LIBSCI_ACC_GENCOMPS_GNU_x86_64=49
SLURM_NODEID=0
CRAY_DMAPP_POST_LINK_OPTS=-L/opt/cray/dmapp/7.1.1-6.
↪ 0.7.0_34.3__g5a674e0.ari/lib64
PE_FFTW_DEFAULT_TARGET_ivybridge=ivybridge
PE_FFTW_DEFAULT_TARGET_share=share
PE_FFTW_DEFAULT_TARGET_x86_skylake=x86_skylake
PE_PKG_CONFIG_PATH=/opt/cray/pe/cti/1.0.7/lib/pkgcon
↪ fig:/opt/cray/pe/cti/1.0.6/lib/pkgconfig:/opt/cr
↪ ay/pe/cti/1.0.4/lib/pkgconfig
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_interlagos=71 53 49
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_mic_knl=16.0
PE_LIBSCI_ACC_GENCOMPILERS_CRAY_x86_64=8.5
SLURM_STEP_RESV_PORTS=20447
SLURM_SUBMIT_DIR=/users/USER/run
CRAY_RCA_INCLUDE_OPTS=-I/opt/cray/rca/2.2.18-6.0.7.0
↪ _33.3__g2aa4f39.ari/include
↪ -I/opt/cray/krca/2.2.4-6.0.7.1_5.27__g8505b97.ar
↪ i/include
↪ -I/opt/cray-hss-devel/8.0.0/include
PAT_BUILD_PAPI_BASEDIR=/opt/cray/pe/papi/5.6.0.3
PE_LIBSCI_OMP_REQUIRES_openmp=_mp
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_skylake=6.1
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_x86_skylake=8.6
EBROOTJUPYTER=/apps/daint/UES/jenkins/6.0.UP07/gpu/e
↪ asybuild/software/jupyter/1.0.0-CrayGNU-18.08
SLURM_TASK_PID=14849
SLURM_NPROCS=1
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_mic_knl=86
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
CRAY_MPICH_DIR=/opt/cray/pe/mpt/7.7.2/gni/mpich-gnu/
↪ 7.1
PE_MPICH_CXX_PKGCONFIG_LIBS=mpichcxx
EBDEVELJUPYTERHUB=/apps/daint/UES/jenkins/6.0.UP07/g
↪ pu/easybuild/software/jupyterhub/0.9.4-CrayGNU-1
↪ 8.08/easybuild/jupyterhub-0.9.4-CrayGNU-18.08-ea
↪ sybuild-devel
SLURM_DISTRIBUTION=cyclic
SLURM_CPUS_ON_NODE=24
SQUEUE_FORMAT=%.8i %.8u %.7a %.14j %.3t %9r %19S
↪ %.10M %.10L %.5D %.4C
PE_LIBSCI_ACC_DEFAULT_GENCOMPILERS_GNU_x86_64=4.9
PE_LIBSCI_DEFAULT_GENCOMPS_INTEL_x86_64=160
PE_MPICH_PKGCONFIG_VARIABLES=PE_MPICH_NV_LIBS_@accel
↪ erator@:PE_MPICH_ALTERNATE_LIBS_@multithreaded@:
↪ PE_MPICH_ALTERNATE_LIBS_@dpm@
PE_LIBSCI_ACC_PKGCONFIG_LIBS=libsci_acc
EBROOTBOOST=/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/software/Boost/1.67.0-CrayGNU-18.08-pytho
↪ n3
EBDEVELCMAKE=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/CMake/3.12.0/easybuild/CMake-.3
↪ .12.0-easybuild-devel
SLURM_PROCID=0
ENVIRONMENT=BATCH
APP2_STATE=7.0.3
CRAY_PMI_POST_LINK_OPTS=-L/opt/cray/pe/pmi/5.0.14/li
↪ b64
PE_HDF5_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_TPSL_64_DEFAULT_GENCOMPILERS_CRAY_mic_knl=8.6
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_x86_skylake=16.0
PE_TPSL_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe
↪ /tpsl/18.06.1/@PRGENV@/@PE_TPSL_DEFAULT_GENCOMPS
↪ @/@PE_TPSL_DEFAULT_TARGET@/lib/pkgconfig
CRAY_MPICH2_VER=7.7.2
PE_MPICH_PKGCONFIG_LIBS=mpich
PE_LIBSCI_ACC_NV_SUFFIX_nvidia35=nv35
SLURM_JOB_NODELIST=nid04277
GPG_TTY=not a tty
PE_GA_DEFAULT_GENCOMPILERS_GNU=5.3 4.9
PE_LIBSCI_ACC_DEFAULT_GENCOMPS_GNU_x86_64=49
PE_LIBSCI_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/libsc
↪ i/18.07.1/@PRGENV@/@PE_LIBSCI_GENCOMPS@/@PE_LIBS
↪ CI_TARGET@/lib/pkgconfig
PE_MPICH_ALTERNATE_LIBS_multithreaded=_mt
PE_NETCDF_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_PARALLEL_NETCDF_DEFAULT_FIXED_PRGENV=CRAY PGI
↪ INTEL
EBDEVELNODEJS=/apps/daint/UES/jenkins/6.0.UP07/gpu/e
↪ asybuild/software/nodejs/8.9.4-CrayGNU-18.08/eas
↪ ybuild/nodejs-.8.9.4-CrayGNU-18.08-easybuild-dev
↪ el
JUPYTERLAB_DIR=/apps/daint/UES/jenkins/6.0.UP07/gpu/
↪ easybuild/software/jupyterlab/0.35.2-CrayGNU-18.
↪ 08/share/jupyter/lab/
CRAY_TCMALLOC_MEMFS_FORCE=1
```

```
EBDEVELBZIP2=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/bzip2/1.0.6-CrayGNU-18.08/easyb
↪ uild/bzip2-.1.0.6-CrayGNU-18.08-easybuild-devel
EBVERSIONCMAKE=3.12.0
HOME=/users/USER
SHLVL=4
JDK_HOME=/usr/lib64/jvm/java
QT_SYSTEM_DIR=/usr/share/desktop-data
CRAY_LIBSCI_VERSION=18.07.1
PE_HDF5_PARALLEL_DEFAULT_VOLATILE_PRGENV=GNU
PE_MPICH_TARGET_VAR_nvidia35=-lcudart
PE_NETCDF_HDF5PARALLEL_DEFAULT_VOLATILE_PRGENV=GNU
PE_PKGCONFIG_PRODUCTS_DEFAULT=PE_PAPI
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_haswell=71 53 49
CUDATOOLKIT_HOME=/opt/nvidia/cudatoolkit9.1/9.1.85_3
↪ .18-6.0.7.0_5.1__g2eb7c52
SLURM_LOCALID=0
OSTYPE=linux
LESS_ADVANCED_PREPROCESSOR=no
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia60=nv60
PE_MPICH_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/p
↪ e/mpt/7.7.2/gni/mpich-@PRGENV@@PE_MPICH_DEFAULT_
↪ DIR_DEFAULT64@/@PE_MPICH_DEFAULT_GENCOMPS@/lib/p
↪ kgconfig
PE_PETSC_DEFAULT_GENCOMPILERS_CRAY_interlagos=8.6
PE_TPSL_DEFAULT_VOLATILE_PRGENV=CRAY CRAY64 GNU GNU64
↪ INTEL INTEL64
SLURM_JOB_GID=32035
SLURM_JOB_CPUS_PER_NODE=24
SLURM_CLUSTER_NAME=daint
XCURSOR_THEME=DMZ
LS_OPTIONS=-N --color=none -T 0
CRAY_PMI_INCLUDE_OPTS=-I/opt/cray/pe/pmi/5.0.14/incl
↪ ude
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_interlagos=86
PE_TPSL_DEFAULT_GENCOMPS_INTEL_sandybridge=160
EBROOTIPYTHON=/apps/daint/UES/jenkins/6.0.UP07/gpu/e
↪ asybuild/software/IPython/5.7.0-CrayGNU-18.08-py
↪ thon3
SLURM_GTIDS=0
SLURM_SUBMIT_HOST=nid04277
WINDOWMANAGER=
PRGENVMODULES=PrgEnv-cray:PrgEnv-gnu:PrgEnv-intel:Pr
↪ gEnv-pathscale:PrgEnv-pgi
CRAYPE_NETWORK_TARGET=aries
ATP_MRNET_COMM_PATH=/opt/cray/pe/atp/2.1.2/libexec/a
↪ tp_mrnet_commnode_wrapper
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_haswell=8.6
PKG_CONFIG_PATH_DEFAULT=/opt/cray/pe/papi/5.6.0.2/li
↪ b64/pkgconfig
PE_MPICH_DIR_CRAY_DEFAULT64=64
GCC_PATH=/opt/gcc/6.2.0
CRAY_CUDATOOLKIT_DIR=/opt/nvidia/cudatoolkit9.1/9.1.
↪ 85_3.18-6.0.7.0_5.1__g2eb7c52
CRAY_CUDATOOLKIT_INCLUDE_OPTS=-I/opt/nvidia/cudatool
↪ kit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52/include
↪ -I/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0
↪ _5.1__g2eb7c52/extras/CUPTI/include
↪ -I/opt/nvidia/cudatoolkit9.1/9.1.85_3.18-6.0.7.0
↪ _5.1__g2eb7c52/extras/Debugger/include
SLURM_JOB_PARTITION=debug
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3 4.9
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_mic_knl=7.1 5.3
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1 5.3
↪ 4.9
PE_TPSL_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16.0
LOGNAME=USER
MACHTYPE=x86_64-suse-linux
LESS=-M -I -R
G_FILENAME_ENCODING=@locale,UTF-8,ISO-8859-15,CP1252
CRAY_GNI_HEADERS_INCLUDE_OPTS=-I/opt/cray/gni-header
↪ s/5.0.12.0-6.0.7.0_24.1__g3b1768f.ari/include
CRAY_LIBSCI_PREFIX_DIR=/opt/cray/pe/libsci/18.07.1/G
↪ NU/6.1/x86_64
PE_HDF5_DEFAULT_GENCOMPS_GNU=
PE_MPICH_NV_LIBS=
PE_NETCDF_DEFAULT_REQUIRED_PRODUCTS=PE_HDF5
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_haswell=7.1 5.3
↪ 4.9
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_sandybridge=16
↪ .0
PE_TPSL_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_TRILINOS_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH:PE_HD
↪ F5_PARALLEL:PE_NETCDF_HDF5PARALLEL:PE_LIBSCI:PE_
↪ TPSL
PYTHONPATH=/users/USER/code/pystencils/pystencils:/u
↪ sers/USER/code/pystencils/pygrandchem:/users/USE
↪ R/code/pystencils/pystencils_walberla:/users/USE
↪ R/code/walberla/python
SLURM_STEP_NUM_TASKS=1
CVS_RSH=ssh
DMAPP_ABORT_ON_ERROR=1
PE_LIBSCI_OMP_REQUIRES=
PE_MPICH_DEFAULT_GENCOMPILERS_CRAY=8.6
PE_TRILINOS_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_MPICH_GENCOMPS_CRAY=86
BOOST_ROOT=/apps/daint/UES/jenkins/6.0.UP07/gpu/easy
↪ build/software/Boost/1.67.0-CrayGNU-18.08-python3
GPU_DEVICE_ORDINAL=0
SLURM_JOB_ACCOUNT=ACCOUNT
SSH_CONNECTION=148.187.1.11 41298 148.187.26.68 22
XDG_DATA_DIRS=/usr/share
TOOLMODULES=apprentice:apprentice2:atp:chapel:cray-l
↪ gdb:craypat:craypkg-gen:cray-snplauncher:ddt:gdb
↪ :iobuf:papi:perftools:perftools-lite:stat:totalv
↪ iew:xt-craypat:xt-lgdb:xt-papi:xt-totalview
DVS_INCLUDE_OPTS=-I/opt/cray/dvs/2.7_2.2.113-6.0.7.1
↪ _7.6__g1bbc03e/include
PE_LIBSCI_ACC_DEFAULT_GENCOMPILERS_CRAY_x86_64=8.5
PE_LIBSCI_ACC_DEFAULT_NV_SUFFIX_nvidia35=nv35
```

```
PE_LIBSCI_DEFAULT_REQUIRED_PRODUCTS=PE_MPICH
PE_MPICH_DEFAULT_FIXED_PRGENV=INTEL
PE_MPICH_DEFAULT_GENCOMPS_GNU=71 51 49
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_interlagos=16.0
PE_TPSL_DEFAULT_GENCOMPILERS_CRAY_sandybridge=8.6
EBEXTSLISTJUPYTERHUB=pamela-0.3.0,tornado-5.1.1,deco
↪ rator-4.3.0,ipython_genutils-0.2.0,traitlets-4.3
↪ .2,python-oauth2-1.1.0,SQLAlchemy-1.2.12,MarkupS
↪ afe-1.0,Mako-1.0.7,python-editor-1.0.3,python-da
↪ teutil-2.7.3,alembic-1.0.0,prometheus_client-0.4
↪ .1,async_generator-1.10,chardet-3.0.4,certifi-20
↪ 18.8.24,idna-2.7,urllib3-1.23,requests-2.19.1,Ji
↪ nja2-2.10,jupyterhub-0.9.4
SLURM_JOB_NUM_NODES=1
MODULESHOME=/opt/cray/pe/modules/3.2.10.6
PE_GA_DEFAULT_FIXED_PRGENV=CRAY PGI INTEL
PE_LIBSCI_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/
↪ pe/libsci/18.07.1/@PRGENV@/@PE_LIBSCI_DEFAULT_GE
↪ NCOMPS@/@PE_LIBSCI_DEFAULT_TARGET@/lib/pkgconfig
PE_TPSL_DEFAULT_GENCOMPILERS_GNU_sandybridge=7.1 5.3
↪ 4.9
PE_LIBSCI_ACC_VOLATILE_PRGENV=CRAY GNU
LESSOPEN=lessopen.sh %s
PKG_CONFIG_PATH=/apps/daint/UES/jenkins/6.0.UP07/gpu
↪ /easybuild/software/zlib/1.2.11-CrayGNU-18.08/li
↪ b/pkgconfig:/apps/daint/UES/jenkins/6.0.UP07/gpu
↪ /easybuild/software/bzip2/1.0.6-CrayGNU-18.08/li
↪ b/pkgconfig:/opt/nvidia/cudatoolkit9.1/9.1.85_3.
↪ 18-6.0.7.0_5.1__g2eb7c52/lib64/pkgconfig:/opt/cr
↪ ay/rca/2.2.18-6.0.7.0_33.3__g2aa4f39.ari/lib64/p
↪ kgconfig:/opt/cray/alps/6.6.43-6.0.7.0_26.4__ga7
↪ 96da3.ari/lib64/pkgconfig:/opt/cray/xpmem/2.2.15
↪ -6.0.7.1_5.10__g7549d06.ari/lib64/pkgconfig:/opt/
↪ cray/gni-headers/5.0.12.0-6.0.7.0_24.1__g3b1768f
↪ .ari/lib64/pkgconfig:/opt/cray/dmapp/7.1.1-6.0.7
↪ .0_34.3__g5a674e0.ari/lib64/pkgconfig:/opt/cray/
↪ pe/pmi/5.0.14/lib64/pkgconfig:/opt/cray/ugni/6.0
↪ .14.0-6.0.7.0_23.1__gea11d3d.ari/lib64/pkgconfig
↪ :/opt/cray/udreg/2.3.2-6.0.7.0_33.18__g5196236.a
↪ ri/lib64/pkgconfig:/opt/cray/pe/craype/2.5.15/pk
↪ g-config:/opt/cray/pe/iobuf/2.0.8/lib/pkgconfig:
↪ /opt/slurm/17.11.12.cscs/lib64/pkgconfig:/opt/sl
↪ urm/default/lib64/pkgconfig:/opt/cray/pe/atp/2.1
↪ .2/lib/pkgconfig
SLURM_TIME_FORMAT=relative
PE_MPICH_NV_LIBS_nvidia35=-lcudart
PE_PETSC_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/p
↪ e/petsc/3.8.4.0/complex/@PRGENV@/@PE_PETSC_DEFAU
↪ LT_GENCOMPS@/@PE_PETSC_DEFAULT_TARGET@/lib/pkgco
↪ nfig
PELOCAL_PRGENV=true
CRAY_NUM_COOKIES=2
SLURM_STEP_TASKS_PER_NODE=1
CRAYPAT_OPTS_EXECUTABLE=sbin/pat-opts
LIBSCI_BASE_DIR=/opt/cray/pe/libsci/18.07.1
PE_TPSL_64_DEFAULT_GENCOMPS_INTEL_x86_64=160
EBEXTSLISTIPYTHON=decorator-4.3.0,wcwidth-0.1.7,prom
↪ pt_toolkit-1.0.15,pickleshare-0.7.4,parso-0.2.1,
↪ jedi-0.12.0,ptyprocess-0.6.0,pexpect-4.6.0,ipyth
↪ on_genutils-0.2.0,traitlets-4.3.2,backcall-0.1.0
↪ ,simplegeneric-0.8.1,Pygments-2.2.0,ipython-5.7.0
PE_LIBSCI_ACC_GENCOMPILERS_GNU_x86_64=4.9
EBROOTBZIP2=/apps/daint/UES/jenkins/6.0.UP07/gpu/eas
↪ ybuild/software/bzip2/1.0.6-CrayGNU-18.08
CRAY_COOKIES=1384382464,1384448000
SLURM_STEP_NODELIST=nid04277
LIBSCI_VERSION=18.07.1
PE_LIBSCI_DEFAULT_PKGCONFIG_VARIABLES=PE_LIBSCI_DEFA
↪ ULT_OMP_REQUIRES_@openmp@:PE_SCI_EXT_LIBPATH:PE_
↪ SCI_EXT_LIBNAME
PE_MPICH_NV_LIBS_nvidia60=-lcudart
PE_TPSL_64_DEFAULT_GENCOMPS_GNU_sandybridge=71 53 49
PE_TPSL_DEFAULT_GENCOMPS_INTEL_mic_knl=160
INFOPATH=/opt/gcc/6.2.0/snos/share/info
CRAY_LIBSCI_ACC_PREFIX_DIR=/opt/cray/pe/libsci_acc/1
↪ 8.07.1/GNU/4.9/x86_64
PE_LIBSCI_ACC_REQUIRED_PRODUCTS=PE_MPICH:PE_LIBSCI
ACLOCAL_PATH=/apps/daint/UES/jenkins/6.0.UP07/gpu/ea
↪ sybuild/software/CMake/3.12.0/share/aclocal
XDG_RUNTIME_DIR=/run/user/23945
CRAY_PRE_COMPILE_OPTS=-hnetwork=aries
CRAY_ALPS_INCLUDE_OPTS=-I/opt/cray/alps/6.6.43-6.0.7
↪ .0_26.4__ga796da3.ari/include
PE_FFTW_DEFAULT_TARGET_broadwell=broadwell
PE_LIBSCI_GENCOMPILERS_INTEL_x86_64=16.0
PE_PGI_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/par
↪ allel-netcdf/1.8.1.3/PGI/15.3/lib/pkgconfig:/opt
↪ /cray/pe/netcdf-hdf5parallel/4.6.1.2/PGI/17.10/l
↪ ib/pkgconfig:/opt/cray/pe/netcdf/4.6.1.2/PGI/17.
↪ 10/lib/pkgconfig:/opt/cray/pe/hdf5-parallel/1.10
↪ .2.0/PGI/17.10/lib/pkgconfig:/opt/cray/pe/hdf5/1
↪ .10.2.0/PGI/17.10/lib/pkgconfig:/opt/cray/pe/ga/
↪ 5.3.0.8/PGI/17.10/lib/pkgconfig
PE_TPSL_64_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 5.3
↪ 4.9
CRAY_CPU_TARGET=haswell
EBVERSIONBZIP2=1.0.6
CRAY_UGNI_INCLUDE_OPTS=-I/opt/cray/ugni/6.0.14.0-6.0
↪ .7.0_23.1__gea11d3d.ari/include
CRAY_XPMEM_INCLUDE_OPTS=-I/opt/cray/xpmem/2.2.15-6.0
↪ .7.1_5.10__g7549d06.ari/include
PE_LIBSCI_REQUIRED_PRODUCTS=PE_MPICH
PE_MPICH_DEFAULT_GENCOMPILERS_PGI=15.3
PE_PAPI_DEFAULT_ACCELL_FAMILY_LIBS=
PE_TPSL_64_DEFAULT_GENCOMPS_CRAY_x86_64=86
craype_already_loaded=0
PE_MPICH_GENCOMPS_PGI=153
XTPE_LINK_TYPE=dynamic
SLURM_CPU_BIND=quiet,mask_cpu:0xFFFFFF
CUDA_CACHE_PATH=/scratch/snx3000/USER/.nv/ComputeCac
↪ he
```

```
PE_LIBSCI_DEFAULT_GENCOMPILERS_GNU_x86_64=7.1 6.1 5.1
↪ 4.9
PE_LIBSCI_GENCOMPS_GNU_x86_64=71 61 51 49
PE_TPSL_DEFAULT_GENCOMPS_INTEL_haswell=160
LESSCLOSE=lessclose.sh %s %s
ATP_HOME=/opt/cray/pe/atp/2.1.2
PE_FFTW_DEFAULT_TARGET_x86_64=x86_64
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_x86_64=16.0
EBEXTSLISTJUPYTERLAB=jupyterlab_server-0.2.0,jupyter
↪ lab-0.35.2
SLURM_MEM_PER_NODE=61000
SCRATCH=/scratch/snx3000/USER
G_BROKEN_FILENAMES=1
CRAY_LD_LIBRARY_PATH=/opt/nvidia/cudatoolkit9.1/9.1.
↪ 85_3.18-6.0.7.0_5.1__g2eb7c52/lib64:/opt/nvidia/
↪ cudatoolkit9.1/9.1.85_3.18-6.0.7.0_5.1__g2eb7c52
↪ /extras/CUPTI/lib64:/opt/cray/pe/libsci_acc/18.0
↪ 7.1/GNU/4.9/x86_64/lib:/opt/cray/pe/mpt/7.7.2/gn
↪ i/mpich-gnu/7.1/lib:/opt/cray/pe/perftools/7.0.3
↪ /lib64:/opt/cray/rca/2.2.18-6.0.7.0_33.3__g2aa4f
↪ 39.ari/lib64:/opt/cray/alps/6.6.43-6.0.7.0_26.4_
↪ _ga796da3.ari/lib64:/opt/cray/xpmem/2.2.15-6.0.7
↪ .1_5.10__g7549d06.ari/lib64:/opt/cray/dmapp/7.1.
↪ 1-6.0.7.0_34.3__g5a674e0.ari/lib64:/opt/cray/pe/
↪ pmi/5.0.14/lib64:/opt/cray/ugni/6.0.14.0-6.0.7.0
↪ _23.1__gea11d3d.ari/lib64:/opt/cray/udreg/2.3.2-
↪ 6.0.7.0_33.18__g5196236.ari/lib64:/opt/cray/pe/l
↪ ibsci/18.07.1/GNU/6.1/x86_64/lib
PE_FFTW_DEFAULT_TARGET_haswell=haswell
PE_GA_DEFAULT_GENCOMPS_GNU=53 49
PE_GA_DEFAULT_VOLATILE_PKGCONFIG_PATH=/opt/cray/pe/g
↪ a/5.3.0.8/@PRGENV@/@PE_GA_DEFAULT_GENCOMPS@/lib/
↪ pkgconfig
PE_INTEL_DEFAULT_FIXED_PKGCONFIG_PATH=/opt/cray/pe/p
↪ arallel-netcdf/1.8.1.3/INTEL/16.0/lib/pkgconfig:
↪ /opt/cray/pe/netcdf-hdf5parallel/4.6.1.2/INTEL/1
↪ 6.0/lib/pkgconfig:/opt/cray/pe/netcdf/4.6.1.2/IN
↪ TEL/16.0/lib/pkgconfig:/opt/cray/pe/mpt/7.7.2/gn
↪ i/mpich-intel/16.0/lib/pkgconfig:/opt/cray/pe/hd
↪ f5-parallel/1.10.2.0/INTEL/16.0/lib/pkgconfig:/o
↪ pt/cray/pe/hdf5/1.10.2.0/INTEL/16.0/lib/pkgconfi
↪ g:/opt/cray/pe/ga/5.3.0.8/INTEL/18.0/lib/pkgconf
↪ ig
PE_PAPI_DEFAULT_ACCEL_LIBS=
PE_PETSC_DEFAULT_GENCOMPILERS_GNU_interlagos=7.1 5.3
↪ 4.9
PE_PETSC_DEFAULT_GENCOMPILERS_INTEL_haswell=16.0
PE_SMA_DEFAULT_DIR_PGI_DEFAULT64=64
PE_TPSL_64_DEFAULT_GENCOMPILERS_INTEL_x86_skylake=16
↪ .0
EBVERSIONIPYTHON=5.7.0
EBROOTZLIB=/apps/daint/UES/jenkins/6.0.UP07/gpu/easy
↪ build/software/zlib/1.2.11-CrayGNU-18.08
COLORTERM=1
JAVA_ROOT=/usr/lib64/jvm/java
PE_MPICH_DEFAULT_DIR_CRAY_DEFAULT64=64
PE_PETSC_DEFAULT_GENCOMPS_CRAY_haswell=86
PE_PETSC_DEFAULT_GENCOMPS_GNU_x86_64=71 53 49
PE_PETSC_DEFAULT_GENCOMPS_INTEL_x86_64=160
BASH_FUNC_module%%=() {  eval
↪ `/opt/cray/pe/modules/3.2.10.6/bin/modulecmd
↪ bash $*`
}
LSB Version:        n/a
Distributor ID:        SUSE
Description:        SUSE Linux Enterprise Server 12
↪ SP3
Release:        12.3
Codename:        n/a
Linux nid04277 4.4.103-6.38_4.0.153-cray_ari_c #1 SMP
↪ Thu Nov 1 16:05:05 UTC 2018 (6ef8fef) x86_64
↪ x86_64 x86_64 GNU/Linux
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2690 v3
↪ @ 2.60GHz
Stepping:              2
CPU MHz:               2601.000
CPU max MHz:           2601.0000
CPU min MHz:           1200.0000
BogoMIPS:              5199.88
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-23
Flags:                 fpu vme de pse tsc msr pae mce
↪ cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪ dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
↪ pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
↪ bts rep_good nopl xtopology nonstop_tsc
↪ aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
↪ ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
↪ pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
↪ tsc_deadline_timer aes xsave avx f16c rdrand
↪ lahf_lm abm ida arat epb invpcid_single pln pts
↪ dtherm spec_ctrl kaiser tpr_shadow vnmi
↪ flexpriority ept vpid fsgsbase tsc_adjust bmi1
↪ avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc
↪ cqm_occup_llc
MemTotal:       65844884 kB
```

```
MemFree:        63387800 kB
MemAvailable:   63047684 kB
Buffers:           18148 kB
Cached:           778352 kB
SwapCached:            0 kB
Active:           187020 kB
Inactive:         719572 kB
Active(anon):     139996 kB
Inactive(anon):   609860 kB
Active(file):      47024 kB
Inactive(file):   109712 kB
Unevictable:        8184 kB
Mlocked:            8184 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:        118512 kB
Mapped:            81732 kB
Shmem:            639048 kB
Slab:             146216 kB
SReclaimable:      22204 kB
SUnreclaim:       124012 kB
KernelStack:        7008 kB
PageTables:         5152 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    32922440 kB
Committed_AS:     953416 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:     1661940 kB
DirectMap2M:     4495360 kB
DirectMap1G:    62914560 kB
NAME  MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
loop0   7:0    0 18.5M  0 loop
↪  /var/opt/cray/imps-distribution/squash/mounts/p0
loop1   7:1    0  128K  0 loop /var/opt/cray/imps-dis┘
↪  tribution/squash/mounts/global
loop2   7:2    0  2.5G  1 loop /.rootfs_lower_ro
loop3   7:3    0  1.6G  1 loop
↪  /var/opt/cray/imps-image-binding/diags/squash_mo┘
↪  unts/squashfs_3cX6Ph_mount_point
loop4   7:4    0        1 loop
loop5   7:5    0        1 loop
loop6   7:6    0        1 loop
loop7   7:7    0        1 loop
loop8   7:8    0        1 loop
```

```
Wed Apr 10 10:58:29 2019
+-----------------------------------------------------┐
↪  --------------------------+
| NVIDIA-SMI 396.44                 Driver Version:
↪  396.44                    |
|-------------------------------+------------------┐
↪  ---+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A
↪  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage
↪  | GPU-Util  Compute M. |
|===============================+==================┐
↪  ===+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:02:00.0 Off
↪  |                    0 |
| N/A   31C    P0    29W / 250W |      0MiB / 16280MiB
↪  |      0%   E. Process |
+-------------------------------+------------------┐
↪  ---+----------------------+

+-----------------------------------------------------┐
↪  --------------------------+
| Processes:
↪             GPU Memory |
|  GPU       PID   Type   Process name
↪             Usage      |
|=============================================================┐
↪  ==========================|
|  No running processes found
↪                          |
+-----------------------------------------------------┐
↪  --------------------------+
```

ARTIFACT EVALUATION

Verification and validation studies: For parameterization P1 (as described in the paper) simulation results were validated against experimental results.

Accuracy and precision of timings: Scaling experiments have been run multiple times. We ensured that every single timing measurement to compute MLUP/s is averaged over at least 10 seconds. For each node count at least 20 measurements have been taken.