Scalable Parallelization of Stencils using MODA

Nabeeh Jum'ah[1] and Julian Kunkel[2]

[1] Universität Hamburg–Jumah@informatik.uni-hamburg.de
[2] University of Reading–j.m.kunkel@reading.ac.uk

Abstract. The natural and the design limitations of the evolution of processors, e.g., frequency scaling and memory bandwidth bottlenecks, push towards scaling applications on multiple-node configurations besides to exploiting the power of each single node. This introduced new challenges to porting applications to the new infrastructure, especially with the heterogeneous environments. Domain decomposition and handling the resulting necessary communication is not a trivial task. Parallelizing code automatically cannot be decided by tools in general as a result of the semantics of the general-purpose languages.
To allow scientists to avoid such problems, we introduce the *Memory-Oblivious Data Access (MODA)* technique, and use it to scale code to configurations ranging from a single node to multiple nodes, supporting different architectures, without requiring changes in the source code of the application. We present a technique to automatically identify necessary communication based on higher-level semantics. The extracted information enables tools to generate code that handles the communication. A prototype is developed to implement the techniques and used to evaluate the approach. The results show the effectiveness of using the techniques to scale code on multi-core processors and on GPU based machines. Comparing the ratios of the achieved GFLOPS to the number of nodes in each run, and repeating that on different numbers of nodes shows that the achieved scaling efficiency is around 100%. This was repeated with up to 100 nodes. An exception to this is the single-node configuration using a GPU, in which no communication is needed, and hence, no data movement between GPU and host memory is needed, which yields higher GFLOPS.

Keywords: HPC; Scalability; Parallel Programming; Stencils

1 Introduction

In modern computing technology, the processing speed on a single processor core reached its limit. Therefore, parallelism of multiple cores within a node and inter-node communication via high-speed networks is required to satisfy performance demanding applications. For example, developers of earth system modeling software demand higher-resolution grid as they provide more accurate results from a scientific perspective.

The task of rewriting software to scale on multiple nodes is challenging for scientists. It requires distributing the data (domain decomposition [10]) and balancing of the computational load between the nodes and handling the communication. Other considerations and particularly portability should be taken into account, e.g. communicating data residing on device memory when running kernels on GPUs differs from data existing on host memory.

Stencils include access to fields at spatially-neighboring points. This leads to accessing memory multiple times to get field data to load stencil points. Normally, memory access with general-purpose languages indices defines explicitly where the data resides in memory. Such characteristic stems from the design of the general-purpose languages, which carries the semantics of access to local memory.

An important aspect of the move from the local memory to the distributed memories on multi-node machines is locating and accessing data. Generally, with general-purpose languages and explicit memory indices, developers need to keep in mind the domain decomposition and to keep track of the mapped partition of the problem domain to the local memory, and use the indices to access the right data elements. So, mapping the global position of a data element (with respect to the global problem domain) to the right indices in the local memory should be tracked by the developers. Furthermore, developers need to identify the necessary communication between the nodes, and hence write code to prepare the necessary data and handle the exchange to make the data that resides on a remote memory accessible through the local memory.

To avoid the necessity of architecture-specific code inside applications, and to ease the development of new applications, the parallelization should be handled semi-automatically by tools and libraries. Unfortunately, the semantics of the general-purpose languages do not provide compilers with the necessary information to make such decisions.

The **main contribution** of this work is the introduction of the Memory-Oblivious Data Access (MODA) technique to replace local-memory-bound explicit data access; it consists of a technique to extract the necessary semantics from the source code to automatically generate code to handle parallelization of stencil computations on multiple nodes, in addition to shared-memory parallelization.

MODA allows using the same source code on different run configurations including shared and distributed memory, and different architectures. It requires application source code to be written with higher-level semantics. As a prototype we utilize the GGDML [9] language extensions, which allow mixing general-purpose code with additional higher-level semantics to describe stencil computations. GGDML extends the grammar of a programming language with higher semantics that bypasses the architectural differences and provides performance portability [7]. This allows our solution to support performance portability and fit different architectures.

This article is structured as follows: a review of related work is done in section 2, the technique and the methodology are described in section 3, an evaluation of the technique is discussed in section 4, and we conclude the text with section 5.

2 Related Work

The natural limits of the single processors necessitate to seek for methods, strategies, and tools to support performance demanding applications and simplifying the parallelization.

Manual parallelization With this strategy, the developer adjusts the code to integrate parallelization strategies explicitly in the code which means that application logic is mixed with code fragments that control the parallelization. This is an old strategy, for example, in the early times of concurrent computing, [4] applied explicit domain decomposition to run large-scale scientific software applications on concurrent computers, both on distributed and shared memory systems. Domain decomposition was decided by the developers within the code. Compilers are then used to build the code for the target architecture as provided by the developers. This explicit decomposition was successful on different machines, and allows for near-optimal performance.

In fact, in the 1980's many publications were released concerning strategies to apply domain decomposition to parallel computing for various application domains. Domain decomposition of stencil computations represented an important research direction in the evolution of parallel computing technologies. For example, [1] discussed a solution that handles domain decomposition and the necessary interactions between the resulting regions to parallelize elliptic problems. Also, [2] used a domain decomposition strategy to develop a Poisson solver using parallel machines. These are examples among other many suggested solutions at that time. A comparison of domain decomposition strategies was made in [11].

Even in the recent years, many papers are published that parallelize a specific problem manually solving different problems using the recent advances in the computing infrastructure. For example, [14] proposes a communication model to handle data exchange on reconfigurable clusters. Another example, [6] used a domain decomposition strategy to strong scale a solver of the Lattice Quantum Chromodynamics on the KNC Xeon Phi co-processor, which highly reduces the time to solution.

Data-structure libraries Exascale applications will need to access data which resides on another node. To support such applications, some efforts provide solutions at the data-structure level. DASH [5] is an ongoing work (under the Smart-DASH project) to provide data structures that account for node-level parallelism.

Code generation Besides to the evolution of the strategies to apply domain decomposition, another direction in research was taken to support parallelization, and hence simplify the developers' task regarding domain decomposition. Instead of manual coding, tools generate code to solve a problem, including domain decomposition and communication. This is possible because code generators generate code for a problem among a specific family of computations, e.g., elliptical PDE solvers. Code generators use a specification of a problem and generate code to solve that particular problem. This technique is used in many efforts including [3, 13] to generate code for stencil computations. Tools with specific goals, e.g., YASK [15] use code generation to generate optimized code for parallel computing. YASK allows to explore the performance of a stencil on Xeon and Xeon Phi processors, where optimal parameters can be identified for a specific problem (stencil). The ExaStencils [12] project also generates the necessary code based on an abstract higher-level problem specification. ExaStencils is an ongoing project to support multi-grid solutions of stencils counting for the expected exascale computing infrastructures.

In **our work**, we suggest a technique in which higher semantics are extracted from the source code, and used to transform the code to enable domain decomposition and data communication between nodes. Tools identify the necessary communication based on user-defined extensions, which are integrated into a general-purpose language. Those extensions provide Memory-Oblivious Data Access which resolves targeting the actual data location in memory, whether local or remote.

Using our approach, we simplify scaling code to support modern multi-node configurations using the same source code that is used for a single node. In comparison to previous efforts, scientists do not need to manually parallelize their modeling code. Nor do they need to care about calling any libraries to handle domain decomposition or communication or keeping track of such details. Tools infer all needed details from the language extensions. Compared to code generation techniques, developers can still define their indices (which serve and fit the needs of their application) instead of using explicit memory and array semantics or using a predefined set of problem-family-specific constructs, e.g. expressions to solve PDEs assuming a rectangular grid.

3 Methodology

In our approach, the computations are written using a general-purpose language (GPL) extended by language constructs that blend into the GPL. We use GGDML (General Grid Definition and Manipulation Language [9]) language extensions for this purpose. GGDML provides an adaptable set of language extensions to support application needs. The DSL syntax and behavior can be adjusted through configuration files [7] that guide the high-level code transformation procedures. This is prepared based on the needs of the specific application or domain.

With this approach, a big chunk of the code can be kept, except for some small replacements: Loop control code is written with GGDML iterator. The body of the loop is modified by replacing the indices with user-defined extensions and removing the loop structures. The semantics of the GGDML language extensions allow the tools to identify the necessary communication.

Listing 1.1: Example GGDML access operator definitions

```
east_neighbor():   XD=$XD+1
north_neighbor():  YD=$YD+1
west_neighbor():   XD=$XD−1
south_neighbor():  YD=$YD−1
```

3.1 MODA and User-Defined Indices

As discussed, array notation and memory access semantics in general-purpose languages define explicitly the location of data in local memory, which obligates the developers to keep track of mapping data from global domain to distributed memories and handle communication to guarantee access to the right data through local memory when needed. Here comes the role of MODA, where we use GGDML language extensions to access data. With GGDML indices, the source code does not include explicit memory locations that depend on machine semantics. On the contrary, GGDML indices reflect spatial relationships. Thus, developers do not need to know if the neighboring grid cell is in the local memory or in a remote one.

In fact the GGDML indices serve other purposes. As they hide the real location of the data in memory, they allow using different data layouts. Different memory layouts could achieve different performance on different architectures or problems. A study to show the impact was published in [8].

To cope with different application needs, e.g., collocated vs. staggered, regular vs. icosahedral grids, triangular vs. hexagonal vs. rectangular cells, GGDML allows users to define access operators to specify indices. Index adaptability to application needs allows to define halo patterns and identify the necessary communication. An example definition of a GGDML index is illustrated in Listing 1.1. The example shows definitions of access operators to refer to the four neighboring cells around a cell in a regular rectangular grid; as mentioned, the definition is provided by the user and can be adjusted to any problem.

3.2 Using GGDML Indices

To illustrate the flexibility in the definition of access operators, take as example a simple collocated rectangular grid. If we want to write a simple Laplacian kernel using GGDML, we can use the access operators shown in Listing 1.1 inside a configuration file that we use to process our application code. In the source code we can write the following kernel (Listing 1.2). In this kernel, we could access the four neighboring cells using the spatial relationships, which we define to fit our application.

Assume in another application we need to use a staggered grid to compute the divergence at the centers of the grid cells based on flux values which reside on the edges between the grid cells. In this case, we can add a new set of access operators to support this second application, e.g., *east_edge*, *north_edge*, *west_edge*, and *south_edge*. Using those access operators, the kernel can be written as shown

Listing 1.2: Example GGDML code using access operators

```
// Traverse the cells of the grid
foreach c in grid{
  f_H_new[c] = f_H[c] * W1              +
              (f_H[c.east_neighbor() ] +
               f_H[c.north_neighbor()] +
               f_H[c.west_neighbor() ] +
               f_H[c.south_neighbor()]
              ) * W2;
}
```

Listing 1.3: Example GGDML code using access operators in a staggered grid

```
// Traverse the cells of the grid
foreach c in grid{
      // Use GGDML access operators east_edge & west_edge
      //  to refer to the U edges of the cell
      float df = (f_F[c.east_edge()] -
                  f_F[c.west_edge()]) / dx;

      // Use GGDML access operators north_edge & south_edge
      //  to refer to the V edges of the cell
      float dg = (f_G[c.north_edge()] -
                  f_G[c.south_edge()]) / dy;

      f_HT[c] = df + dg;
}
```

in Listing 1.3. The new access operators define new spatial relationships that allowed access to the cell edges.

Looking at both applications, a user (or better scientific programmer) could define the necessary access operators that serve the application, where spatial relationships are used, while no information regarding where the data is located in memory are mentioned. The source code in both applications doesn't explicitly state whether the data is in the local memory or stored remotely.

3.3 Communication Identification

The developers responsibility to track data location, to communicate data between nodes, and to use the right memory indices to access data locally is shifted to the tools through the semantics of the GGDML extensions. Depending on the domain decomposition method, an access operator leads to identify the needed communication if any. For example, *north_edge* is sufficient to let a tool know that the data of the edges should be communicated when the edges of a set of cells reside on a different node when dividing the surface into sub-domains. To do this, we suggest an algorithm (Algorithm 1) to infer some information from the AST and use this information to generate the necessary code to handle the communication.

In this algorithm, we look for data access expressions and process all the access operators used to access data. This processing includes checking if the access operator corresponds to a halo pattern. Information is logged in a list about the variable, e.g., whether we need to read some halo region from a different

```
/* traverse the iterator AST */
foreach AST_node in iterator_subtree do
    /* if the node is an expression to access a field data */
    if AST_node is a field_access_expression then
        /* get field name, list of indices, and access type */
        field_name ← get_field_name(AST_node);
        access_type ← get_access_type(AST_node);    /* e.g., read */
        index_node_list ← get_index_node_list(AST_node);
        /* iterate over the access indices */
        foreach index_node in index_node_list do
            /* use indices to identify necessary communication */
            if is_GGDML_index(index_node) then
                /* build a list of access operators */
                AO_list ← fetch_access_operator_list(index_node);
                /* check all access operators if they require halo
                   exchange */
                foreach AO in AO_list do
                    if is_access_operator_a_probable_halo_exchange_reason(AO)
                     then
                        add_entry_to_needed_halo_exchange_list(AO, field_name,
                         access_type);
                    end
                end
            end
        end
    end
end
/* check redundancies and dependencies */
analyse_and_rebuild_needed_halo_exchange_list();
/* generate code to handle communication */
generate_code_halo_pattern_communication_code();
```

Algorithm 1: Necessary communication detection algorithm

node. This list is further processed to analyze dependencies and redundancies to optimize communication. Finally, code is generated to handle the communication. The generated code includes the necessary data preparations and calls to communication library routines, e.g. MPI_Isend or MPI_Irecv.

To demonstrate the work of the algorithm and the techniques, lets take a look at the example code shown in Listing 1.3. For this code, we apply a domain decomposition of the Y dimension, where a set of consecutive X-rows is stored on a node and processed on it. Based on this domain decomposition and the relationships between the cells and their edges, the expression $f\_G[c.north\_edge()]$ means an X-row of edges (the halo/south-most row) should be communicated from the node that is responsible for the north neighborhood. Our implementation generated the MPI code in Listing 1.4 to handle the needed communication.

Some data access expressions imply the need to access halo data which resides on the same node, which does not need MPI communication. In this case a normal data copy can be done. For example, the access operator *east_edge* in

Listing 1.4: Generated communication sections from example code in Listing 1.3

```
if (mpi_world_size > 1) {
    comm_tag++;
    int pp = mpi_rank != 0 ? mpi_rank - 1 : mpi_world_size - 1;
    int np = mpi_rank != mpi_world_size - 1 ? mpi_rank + 1 : 0;

    MPI_Isend(f_G[0], GRIDX + 1, MPI_FLOAT, pp, comm_tag,
              MPI_COMM_WORLD, &mpi_requests[0]);

    MPI_Irecv(f_G[local_Y_Eregion], GRIDX + 1, MPI_FLOAT,
              np, comm_tag, MPI_COMM_WORLD, &mpi_requests[1]);

    MPI_Waitall(2, mpi_requests, MPI_STATUSES_IGNORE);
}
```

Listing 1.5: Generated data copy from example code in Listing 1.3

```
for (int j = 0; j < local_Y_Eregion; j++) {
    f_F[j][GRIDX] = f_F[j][0];
}
```

the expression $f\_F[c.east\_edge()]$ and the mentioned domain decomposition case means the cells at the rightmost column needs to access their right edges. In this application we use periodic boundaries, in which the rightmost edge of a row is itself the leftmost one. This means, copying those edges allows the rightmost cells to access edges using the same computational kernel. Again our implemented tool generates the following code (Listing 1.5) to copy the data of those halo edges.

After the necessary data is ready in memory on the processing node to execute the computation, the compute kernel can be run. To improve this in lengthy communication cases, the communication code time can be overlapped with the computation time, given that inner regions do not depend on the data that should be communicated. In this case, the computation of the outer region (which depends on halo data) should start after the communication is finished. The computation kernel that is generated from the example code in Listing 1.3 is shown in Listing 1.6.

4 Evaluation

In this section, we show some results achieved using the discussed techniques. Experiments were done on single nodes and multiple nodes, multi-core processors and GPUs were involved in the experiments.

4.1 Test Application

The test application is a solver of the shallow water equations on a two-dimensional regular grid with cyclic boundary conditions[3]. The application applies the finite

[3] The code is available at
`https://github.com/aimes-project/ShallowWaterEquations/`.

Listing 1.6: Generated computing code from example code in Listing 1.3

```
for (size_t blk_start = (0); blk_start < (GRIDX); blk_start += 20000) {
    size_t blk_end = GRIDX;
    if ((blk_end - blk_start) > 20000) blk_end = blk_start + 20000;
    #pragma omp parallel for
    for (size_t YD_index = (0); YD_index < local_Y_Cregion; YD_index++) {
        #pragma omp simd
        for (size_t XD_index= blk_start; XD_index < blk_end; XD_index++){
            {
                float df = (f_F[YD_index][XD_index + 1] -
                            f_F[YD_index][XD_index]) / dx;
                float dg = (f_G[YD_index + 1][XD_index] -
                            f_G[YD_index][XD_index]) / dy;
                f_HT[YD_index][XD_index] = df + dg;
            }
        }
    }
}
```

difference method with an explicit time stepping scheme. Eight kernels are included in which flux, velocities, surface level are computed besides to tendencies in each time step.

4.2 Test System

The multi-core processor experiments are run on dual socket Broadwell nodes on the machine Mistral at the German Climate Computing Center (DKRZ). The processors are Intel(R) Xeon(R) CPU E5-2695 v4 with 2.10GHz. We used the Intel (18.0.2) C compiler and the IntelMPI (2018.1.163) library.

The GPU experiments are run on the nodes on the machine 'Piz Daint' at the Swiss National Supercomputing Center (CSCS). The GPUs are Tesla P100 with 16 GB memory and PCIe interconnect to the host. We used the PGI (17.7.0) C compiler and the MPICH (7.6.0) library.

4.3 Experiments

We used GGDML with the C language to write our code. Configuration files were prepared to guide the code translation into C with OpenMP for multi-core processors, and C with OpenACC for GPUs. Optimization procedures were applied during the translation process, e.g., blocking, to exploit the features, e.g., caching, of the processing units. Parallelization on the node resources, i.e., the cores of the multi-core processors and the threads and SMs on GPUs, was applied using OpenMP and OpenACC.

Translating the source code for the Broadwell and running it on a single node shows near optimal use of the processor. The application (and the kernels) runs with around 80% of the processor's memory bandwidth (measurement with the 'stream_sp_mem_avx' benchmark from the 'Likwid' tools measured 67 GBytes/s). This code uses caches optimally, where minimal data movement between memory and processor is needed. Minimizing the movement of the data in a memory-bound code means the code runs with about an optimal performance.

Using the defined access operators to generate communication code, allows us to run the same source code on multiple nodes. Using our implementation of the technique, we generated the necessary MPI code to handle halo exchange. Running the code on different numbers of nodes we could scale the code to more resources. We use multiples of ten, up to hundred nodes. The results are shown in Figure 1.

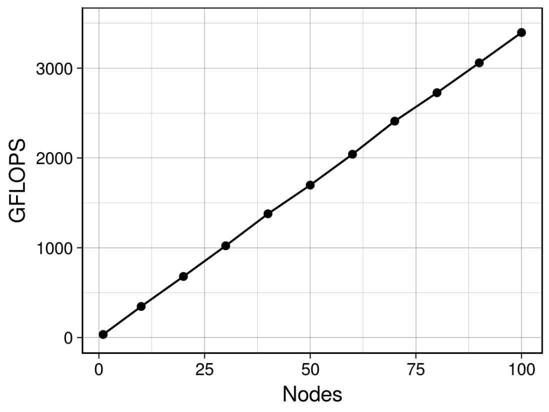

Fig. 1: Scaling on multiple Broadwell nodes

Translating the source code for the P100 GPU and running it on a single node shows near optimal use of the GPU. The application (and the kernels) runs with around 80% of the GPU's memory bandwidth (measurement with a CUDA STREAM benchmark yielded about 498 GB/s). This code uses caches and warps optimally, where minimal data movement between the device memory and the executing GPU threads is done. This means the code runs with about an optimal performance.

Using the access operators again we generated the application code that includes the necessary communication code, which allowed to run the same source code on multiple nodes with GPUs. We generated the necessary MPI code to handle halo exchange, besides to the OpenACC code. The application scaled to multiple nodes with GPUs. Again we use multiples of ten, up to hundred nodes. The results are shown in Figure 2.

To distribute the work between the running resources, both on multi-core processors and on GPUs, the problem domain is decomposed into local domains that reside on each node. Contiguous lines of the grid are given to each local domain. While the domain decomposition strategy maximizes load balance between nodes, other on-node considerations are taken into account. Data reuse, and distribution over cores/threads were maximized with blocking and on-node parallelization.

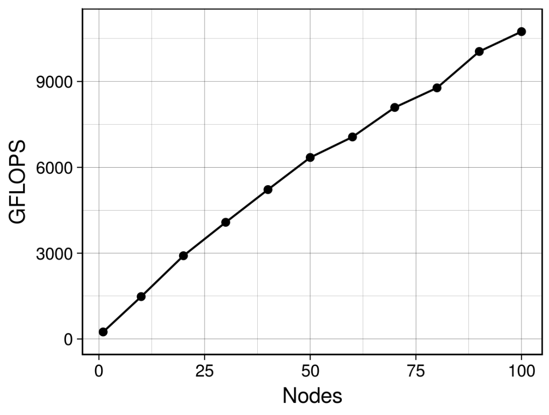

Fig. 2: Scaling on multiple nodes with P100 GPUs

5 Summary

In this paper, we introduced the MODA technique to allow access to data while having no information about where the data is located or whether it is on the local memory or on a remote one. We used the GGDML set of language extensions to access data based on spatial relationships rather than memory location. The GGDML extensions allowed to describe an application in a single (unified) source code, this code could be translated to different target different architectures. We used a technique to extract information through the higher-level semantics to identify the necessary communication to exchange data between the nodes, or even copy data on the local memory. We demonstrated the use of MODA and the GGDML language extensions to write the kernels in two kinds of grids; collocated and staggered. That was possible as a result of the design of GGDML, in which users define access operators. User-defined access operators enable the adaptability of the GGDML extensions to support application-specific needs. We described an algorithm to extract the necessary information from the source code and generate the necessary communication code and hence scale the code, which is not aware of sequential or parallel execution, to use resources on multiple nodes.

We showed the generated inter-node communication code and on-node data copy code which were generated from the example staggered grid kernel. We also showed the results of experiments executed on a test application, which was written with GGDML and C language. The results show that the discussed techniques scale the same source code that can be used for a single node (or even sequential code) to run on multiple nodes. Two architectures were included in our experiments, multi-core processors, and GPUs.

We have a list of todos for future work. We have already implemented some improvements to prepare data to optimize communication, e.g. packing data residing on GPU's device memory, but there are still some work to be done to study and implement interfering communication with computing in more complex computations. Also, improvements towards more flexible and improved preparation of communication code fragments will be published soon, including the flexibility to switch communication libraries, e.g. use GASPI is an alternative to MPI.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing SPPEXA (GZ: LU 1353/11-1). We also thank the Swiss National Supercomputing Center (CSCS), who provided access to their machines to run the experiments. We also thank Prof. John Thuburn – University of Exeter, for his help to develop the code of the shallow water equations.

References

1. Petter E Bjørstad and Olof B Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM Journal on Numerical Analysis*, 23(6):1097–1120, 1986.
2. Tony F Chan and Diana C Resasco. A domain-decomposed fast poisson solver on a rectangle. *SIAM journal on scientific and statistical computing*, 8(1):s14–s26, 1987.
3. Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 676–687. IEEE, 2011.
4. Geoffrey C Fox. Domain decomposition in distributed and shared memory environments. In *International Conference on Supercomputing*, pages 1042–1073. Springer, 1987.
5. Karl Fürlinger, Colin Glass, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. Dash: Data structures and algorithms with support for hierarchical locality. In *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.
6. Simon Heybrock, Bálint Joó, Dhiraj D Kalamkar, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Tilo Wettig, and Pradeep Dubey. Lattice qcd with domain decomposition on intel® xeon phi^{TM} co-processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 69–80. IEEE Press, 2014.
7. Nabeeh Jum'ah and Julian Kunkel. Performance portability of earth system models with user-controlled ggdml code translation. In Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, pages 693–710, Cham, 2018. Springer International Publishing.
8. Nabeeh Jumah and Julian Kunkel. Automatic vectorization of stencil codes with the ggdml language extensions. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'19, pages 2:1–2:7, New York, NY, USA, 2019. ACM.

9. Nabeeh Jumah, Julian M Kunkel, Günther Zängl, Hisashi Yashiro, Thomas Dubos, and Thomas Meurdesoif. Ggdml: icosahedral models language extensions. *Journal of Computer Science Technology Updates*, 4(1):1–10, 2017.
10. David E Keyes. Domain decomposition: a bridge between nature and parallel computers. Technical report, INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA, 1992.
11. David E Keyes and William D Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s166–s202, 1987.
12. Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.
13. Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2011.
14. Xinyu Niu, Jose GF Coutinho, and Wayne Luk. A scalable design approach for stencil computation on reconfigurable clusters. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.
15. Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2016 Sixth International Workshop on*, pages 30–39. IEEE, 2016.