A Framework for Auto-Parallelization and Code Generation: An Integrative Case Study with Legacy FORTRAN Codes

Konstantinos Krommydas
Intel Corporation
Hillsboro, Oregon
konstantinos.krommydas@intel.com

Paul Sathre
Dept. of Computer Science, Virginia Tech
Blacksburg, Virginia
sath6220@cs.vt.edu

Ruchira Sasanka
Intel Corporation
Hillsboro, Oregon
ruchira.sasanka@intel.com

Wu-chun Feng
Dept. of Computer Science, Virginia Tech
Blacksburg, Virginia
feng@cs.vt.edu

ABSTRACT

GLAF, short for Grid-based Language and Auto-parallelization Framework, is a programming framework that seeks to democratize parallel programming by facilitating better productivity in parallel computing via an intuitive graphical programming interface (GPI) that automatically parallelizes and generates code in many languages. Originally, GLAF addressed program development from scratch via the GPI; but this unduly restricted GLAF's utility to creating new codes only. Thus, this paper extends GLAF by enabling program development from pre-existing kernels of interest, which can then be easily and transparently integrated into existing legacy codes. Specifically, we address the theoretical and practical limitations of integration and interoperability of auto-generated parallel code within existing FORTRAN codes; enhance GLAF to overcome these limitations; and present an integrative case study and evaluation of the enhanced GLAF via the implementation of important kernels in two NASA codes: (1) the *Synoptic Surface & Atmospheric Radiation Budget (SARB)*, part of the *Clouds and the Earth's Radiant Energy System (CERES)*, and (2) the *Fully Unstructured Navier-Stokes (FUN3D)* suite for computational fluid dynamics.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Integrated and visual development environments**; *Software notations and tools*;

ACM Reference Format:
Konstantinos Krommydas, Paul Sathre, Ruchira Sasanka, and Wu-chun Feng. 2018. A Framework for Auto-Parallelization and Code Generation: An Integrative Case Study with Legacy FORTRAN Codes. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225143

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225143

1 INTRODUCTION

With the ubiquity of parallel computing architectures and their increasing heterogeneity — from traditional multi-core CPU architectures to accelerator/co-processor architectures, e.g., graphics processing units (GPU), Intel Many Integrated Core (MIC), and field-programmable gate arrays (FPGAs) — application programmers face the daunting task of trying to realize a significant fraction of their theoretical performance. Unfortunately, this daunting task is further excerbated by the variety of programmming models and interfaces (e.g., CUDA/C, CUDA/Fortran, OpenACC/C, OpenMP/C, and so on) to the aforementioned architectures.

To ameliorate the above task, and in turn, democratize parallel programming, many approaches (including domain-specific languages, auto-tuning frameworks, programming languages, library-based frameworks) have been proposed in the past [3, 5, 6, 8–10]. Different approaches each come with their own benefits and drawbacks [7, 17], which we discuss in greater detail in §5 in the context of this work. Along these lines, we created the Grid-based Language and Auto-parallelization Framework (GLAF) [14, 15], a programming framework that seeks to bridge the gaps between performance, programmability and portability on parallel computing architectures. While GLAF addressed many of the shortcomings of prior alternative approaches, GLAF itself has been limited in that it has focused exclusively on developing a program from scratch via the GLAF graphical programming interface (GPI). Although this approach enabled the application programmer to take advantage of the framework's auto-parallelization, it failed to address an important use case: auto-parallelization and optimization of *existing* programs, where a *few* critical functions often consume the bulk of the total execution time. In fact, this is a common case with multi-core CPU architectures and accelerator/co-processor architectures, where compute-intensive code segments are offloaded to multiple threads on a multi- or many-core CPU or accelerator/co-processor (e.g., GPU or Intel MIC) while the rest of the code executes serially on the CPU. In such cases, performance parallelization and optimization techniques would be applied to these aforementioned functions, i.e., *kernels*, rather than the entire legacy program, which can be on the order of hundreds of thousands to millions of source lines of code (SLOC). Thus, developing an existing program from

scratch within a programming framework like GLAF, only to parallelize and optimize a (relatively) small fraction of the entire legacy code, is inefficient and impractical.

To address this, we adapt and extend our previous work [15], which dealt with writing programs from scratch, to auto-parallelize and auto-generate kernel code that can be seamlessly integrated with two legacy FORTRAN codes from NASA: (1) the *Synoptic Surface & Atmospheric Radiation Budget (SARB)* (or "Synoptic SARB"), part of the *Clouds and the Earth's Radiant Energy System (CERES)*, and (2) the *Fully Unstructured Navier-Stokes (FUN3D)* suite of computational fluid dynamics simulation and design tools. Our contributions encompass the following:

- Characterization of issues and solutions for the integration and interoperatiliby of auto-generated code within existing FORTRAN codes.
- Extension of the GLAF user interface on the front-end and code generation on the back-end to facilitate the aforementioned integration and interoperability.
- Analysis and evaluation of the enhanced GLAF via two real-world applications — Synoptic SARB and FUN3D — as a case study.

The rest of the paper is organized as follows: §2 presents an overview of GLAF and the two NASA codes: *Synoptic SARB* and *FUN3D*. §3 describes the needed extensions in GLAF for code generation and for facilitating integration of GLAF auto-generated code with existing legacy FORTRAN code. §4 presents the details of realizing the kernels within Synoptic SARB and FUN3D using GLAF as well as an evaluation of functional correctness, performance insights, and limitations. Lastly, §6 concludes the paper.

2 BACKGROUND

We provide a brief overview of GLAF, short for Grid-based Language and Auto-parallelization Framework, which is our originally proposed visual parallel programming framework [15]. We leverage and extend the original GLAF realization as a means to showcase the problem at hand, i.e., the fact that programming frameworks generally provide minimal provisions for code integration[1] within large pre-existing legacy codes. Then, we present the two real-world codes from NASA, in particular, specific kernels that we use to showcase our approach.

2.1 GLAF

GLAF is a programming framework that facilitates high-productivity parallel programming. Towards this goal, GLAF employs a graphical programming interface (GPI) on the front-end and three back-ends that are responsible for (1) auto-parallelization, (2) code optimization, and (3) automatic code generation,

The GLAF programming model revolves around the concept of a *grid*. All variables in GLAF (e.g., scalar variables, arrays, structs) are represented via the grid abstraction. A grid can represent data structures as simple as a scalar variable or multi-dimensional array or as complex as C-like structs with elements of varying data types, e.g., trees or graphs. In fact, the grid abstraction is general and scalable and can represent any discrete and finite mathematical relation. Figure 1 shows the internal representation of a grid in GLAF. Its

[1]Either in a specialized language or auto-generated code in a traditional programming language.

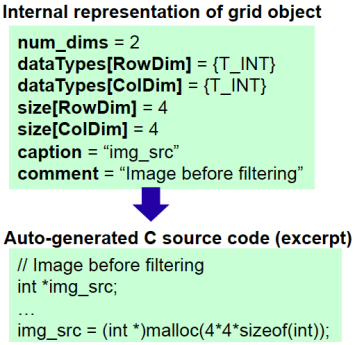

Figure 1: Grid internal representation (simplification)

structure format enables a uniform, regular internal representation that guides GLAF's back-end features, as articulated above.

Figure 2 shows the graphical programming interface (GPI) of GLAF, which serves as the interface to the grid internal representation. It utilizes HTML5 and JavaScript in the familiar web-browser abstraction for easy access to the programmer. In addition, it uses a simple point-and-click interface for most of its functionality with text entry being enabled only when necessary (e.g., naming grids). The GPI also enforces a structured way of programming in *modules* that include *functions*, which in turn, are composed of *steps*. A special module, *Global Scope*, represents a scope that is visible across the whole program. This workflow provides a structured way of programming that is ideal for novice or inexperienced programmers and greatly reduces complexity and the chances for programming errors, which may occur with free-flow, text-based programming languages. Overall, the GPI delivers more intuitive code development, enables visualization of the computation and results, and facilitates debugging.

User actions via the GPI guide the implementation of the algorithm and feed the appropriate information (in the form of the *GLAF internal representation*) to the GLAF back-ends. Specifically, GLAF functionality is handled by three back-ends:

- *Auto-parallelization* includes algorithms that parse the internal representation of the algorithm, identify dependencies, and guide code generation of parallel code (e.g., OpenMP directives for C and Fortran or appropriate kernel offload calls and kernels for OpenCL).
- *Code optimization* includes options for guiding the code generation by providing different data layout (array-of-structures vs. structure-of-arrays), loop collapsing, or loop interchange options.
- *Automatic code generation* parses the internal representation, collects the input from the auto-parallelization and code optimization back-ends, and generates human-readable, compatible code for the selected language.

Originally, GLAF sought to facilitate programming from scratch *once* and then to automatially generate parallel source code in *many* target languages. Specifically, GLAF supported C and FORTRAN auto-parallelization and code generation [15] and later extended support to OpenCL [14], thus facilitating execution on target devices that follow offload programming models, e.g., GPUs and FPGAs. In contrast, this work exploits the extensibility of the GLAF

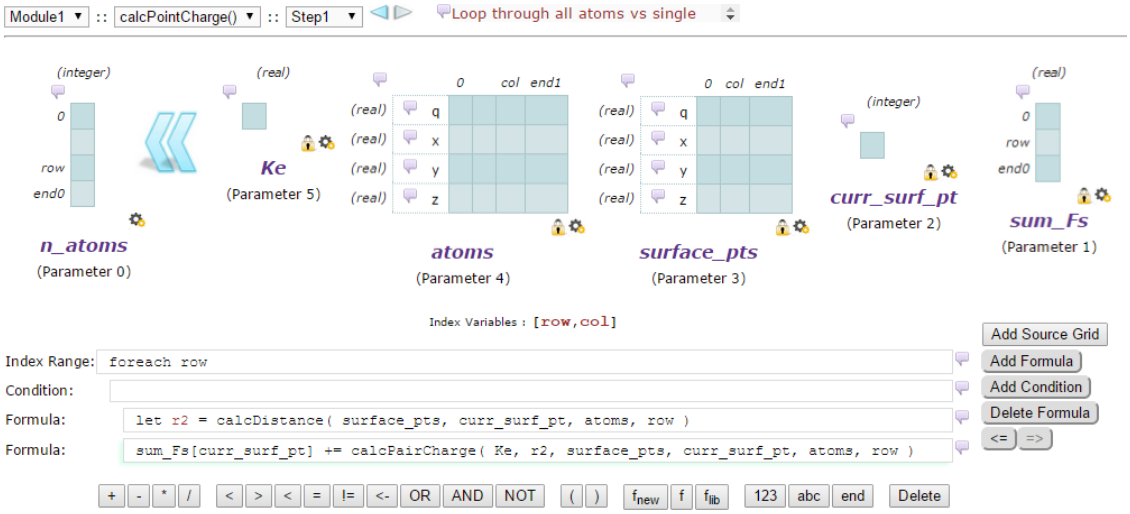

Figure 2: GLAF graphical programming interface (GPI) [15]

framework in order to accomodate the common need for parallelization of certain parts of *existing* legacy codes, two of which we discuss below.

2.2 Synoptic SARB Code

NASA's *Cloud and the Earth's Radiant Energy System* (CERES) program [13] seeks to enhance the existing knowledge about the Earth's climate system and contribute to the associated climate prediction models. Specifically, it aims to provide a better understanding of the interdependencies between clouds and the energy cycle and their effect in the global climate change.

The key software system in CERES is *SARB*, short for *Surface and Atmospheric Radiation Budget*. Each subsystem of SARB ingests certain inputs and generates outputs that may be archived "as is" or used as inputs in a subsequent subsystem. In the context of this work, we focus on *Synoptic SARB*, which computes the vertical longwave, shortwave, and window channel flux profiles that span from the surface of the earth to the top of the atmosphere. That is, it calculates the energy exchange between the sun, the earth's atmosphere, clouds and surface, and outer space.

For Synoptic SARB, the earth is split into multiple zones that run parallel to the equator. Computation for each zone can occur independently (and hence in parallel), serial processing occurs *within* each zone, according to the *synoptic hour* for which the data being processed was acquired via the related measurement collection instruments. However, *within* each synoptic hour, many computations can run in parallel in a finer-grained resolution. The execution of each zone takes time that is proportional to its size (i.e., zones closer to the equator are naturally larger than zones near the poles). Prior to our introduction to the code, Synpotic SARB only used (coarse-grained) inter-zone parallelism via MPI. Opportunities for intra-zone parallelism were ignored. Hence, any opportunities for the parallel execution of loops via multithreading or vectorization within a zone were left unexploited, leaving computing resources underutilized. As one of the two real-world applications in our case study, we use GLAF to implement subroutines (or functions) of interest to the Synoptic SARB program, and evaluate the implemented subroutines for functional correctness and for performance.

2.3 FUN3D Code

FUN3D, short for Fully Unstructured Navier-Stokes, is a suite of applications developed by NASA to support research into unstructured grid computational fluid dynamics (CFD) simulations. The suite supports incompressible and compressible transonic flows and has evolved over the years to include many additional capabilities. The suite supports a number of input grid formats, boundary conditions, grid motion and adaptation, and design optimization. For additional information, see the expansive FUN3D website details about the scientific community's research contributions, including related research publications and presentations [1, 2, 4].

We focus on the *Jacobian matrix reconstruction* portion of the FUN3D solver. This phase of FUN3D had not been previously parallelized. The matrix reconstruction consists of about 10 subroutines that build pieces of the matrix for linear solving. This linear solver is realized by using a Green-Gauss formulation of primitive gradients at nodes in order to try and resolve an inconsistency in heating and shear observed with the former Gram-Schmidt approach [12]. For this portion of reconstruction, the computation must account for the flux across all the cells in the local MPI process' domain. Within each cell, this process must further loop over all nodes, faces, and edges within a cell. Thus, many opportunities exist to exploit intra-node parallelism at multiple scales. Currently, the original matrix reconstruction is implemented as a single function with several levels of loop nesting. GLAF (via its GPI) realizes the entire function, including all interior nested loops, such that opportunities for parallelization at each loop nesting level may be investigated, both in isolation and in concert.

3 ENABLING TRANSPARENT CODE INTEGRATION WITH ENHANCED GLAF

Due to the inability of existing programming frameworks (including GLAF until now) to easily and transparently integrate auto-generated kernel code into an existing legacy code, we articulate below how we extend GLAF to enable such transparent code integration via two real-world NASA programs, both of which happen to be written in FORTRAN. Specifically, we provide an overview of the extensions, enhancements, or changes required of the graphical programming interface (GPI) on the front-end and the FORTRAN code generation on the back-end. While the focus of this paper is intentionally on FORTRAN, similar requirements are presented in the case of other languages and many of the solutions presented here can also be applied to code generation for other languages.

3.1 Enabling the Use of Existing Variables from Imported Modules

Many code scenarios include variables that may be declared in externally included files (*modules* in FORTRAN terminology). To access such variables in GLAF and use them in various GLAF steps, we need to create the corresponding grids in the GLAF *Global Scope* and mark them as belonging to an *"existing module."* For grids that belong to an existing FORTRAN module, the user needs to provide the name of the module the grid belongs to. This information is then used in code generation. Figure 3 shows an example screenshot of the GLAF GPI, where the user can specify an existing variable (grid) and indicate it belongs to an existing MODULE.

Specifically, variables that belong to the above category do not need to be re-declared in the body of the function where they are used (so they are excluded from the variable declaration set). However, the code generation back-end needs to generate code for *using* (i.e., importing) the appropriate FORTRAN module (via the appropriate "*USE* < *var_name* >" code).

3.2 Enabling the Use of Variables in COMMON Blocks

The *COMMON block* is a FORTRAN 77 language construct that defines a block of memory that can be shared among different program units (e.g., functions or subroutines). Variables that belong to a COMMON block can be used in any program unit where the COMMON block is referenced. While this eliminates the need for passing variables that belong in COMMON blocks as arguments in functions and subroutines, it is considered a bad programming practice that creates more complex and less maintainable code. Despite the above, COMMON blocks are present in a lot of production-level codes. Therefore, maintaining backward compatibility and enabling integration of GLAF auto-generated FORTRAN code with an existing legacy code requires that COMMON blocks be supported.

The way variables that belong to COMMON blocks are declared in GLAF is similar to variables that belong to existing imported modules. Specifically, the grid that corresponds to such a variable needs to be created in the GLAF *Global Scope* and be marked as belonging to a COMMON block (Figure 3). The user then indicates the name of that COMMON block. In code generation, the appropriate language structure reflects the COMMON block; all the variables in a given program unit that are identified by the code generation back-end as belonging to the same COMMON block are automatically grouped and declared (i.e., grid type and grid name), and the appropriate *"COMMON /< name >/ var1, var2, ..."* code is subsequently generated.

3.3 Enabling the Use of Module-Scope Variables

Module-scope variables in FORTRAN (similar to global-scope variables in C/C++) provide a means for multiple functions from the same module to share data without relying on complex pointer passing schemes. Within GLAF, these variables are treated similarly to those imported from other modules or COMMON blocks. However, as they are provided by the GLAF-generated module, rather than some external component, GLAF must explicitly *declare* and *initialize* them within the module's global scope.

Module-scope variables may be present in existing code that users wish to auto-parallelize via GLAF or may become necessary due to GLAF's enforced rigid program structure. A typical use case would be for multiple subprograms that compute portions of a shared data structure. Within GLAF, an additional use appears when interior loops must return complex data to an outer scope. GLAF requires that interior nested loops be modeled as a separate function call, and thus, complex data may be passed into interior loops, but it is difficult or impossible to pass out without a data structure that is visible to both the interior and exterior function.

3.4 Enabling the Use of Subroutines

The original GLAF version [14, 15], in which the user was limited in developing a program from scratch, model all FORTRAN subprograms as functions and generate the corresponding code. FORTRAN supports *functions* (where a value is returned to the caller) and *subroutines* (where no value is returned). In practice, a subroutine's functionality can be achieved using a function. However, to enable interoperability and integration of auto-generated code with existing FORTRAN code, we need to explicitly support generation of a subprogram as a subroutine (i.e., subprograms that do not return a value, as opposed to functions).

For every subprogram structure, the GLAF GPI presents the programmer with a choice for a return value type in the header step. To indicate preference for a subroutine structure, the user simply needs to specify the *void* data type (i.e., no return value) in the subprogram header screen (Figure 4). The code generation back-end subsequently constructs the subprogram as a subroutine (in the *"SUBROUTINE < subroutine_name >(...arguments...)"* form), and for all the caller sites a *"CALL < subroutine_name >"* subroutine call code is also generated.

3.5 Enabling the Use of Existing Elements of TYPE Variables

TYPE structures in FORTRAN are analogous to *structs* in C and, accordingly, TYPE elements correspond to C struct elements. As with the case of simple variables from existing MODULES, variables that are elements of TYPE structures can be already defined in an existing MODULE.

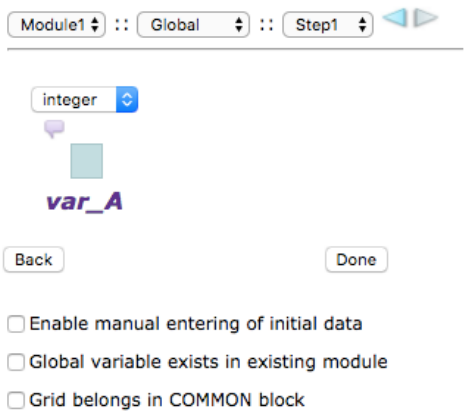

Figure 3: Global scope grid configuration screen

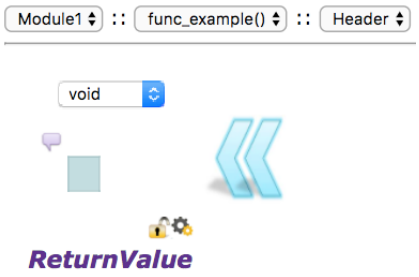

Figure 4: Subprogram header screen

Support for such elements of existing TYPE structures is a subcase of using existing variables from imported modules. The user needs to declare the grid's type (e.g., INTEGER) as normal in the grid definition GPI screen within the *Global Scope*, mark it as belonging to an existing module (Figure 3) and provide the name of the MODULE the TYPE element belongs to (not shown in Figure 3). In addition to simple existing variables from imported modules, the user needs to specify that the grid is part of an existing TYPE structure, in which case he is prompted for the TYPE variable name. On the code generation side, any code for use of the above element of a TYPE structure is appropriately generated with the TYPE variable name prefix (e.g., an element *charge* that belongs to a *TYPE atom* variable named *atom1* would be generated as *atom1%charge*).

3.6 Extending GLAF Auto-Generated Libraries

GLAF already supports a number of common libraries and associated library functions. Such functions correspond to frequently used operations (e.g., GLAF supports a large number of the C/FORTRAN math library functions). Using a library function (via the GPI) leads to appropriate code generation for all supported languages. Libraries are an extensible part of GLAF, which can be adapted as needed to domain-specific needs. In the case of this work, we extended support for the ABS(), ALOG(), SUM(), and other functions, used in FORTRAN that were missing in the previous versions of GLAF.

4 RESULTS

In §3 we discuss the requirements for generating code that can be transparently integrated with existing FORTRAN codes and illustrate how we extend the GLAF programming framework to accomodate such code integration. In this section, we test the enhanced GLAF version by implementing specific kernels (subprograms) of interest from two NASA-related codes: Synoptic SARB and FUN3D.

4.1 SARB: Synoptic SARB

Synoptic SARB is contained within a large and complex code-base. A large part of this code includes the *fulib* library. This library provides an implementation of the *Fu-Liou Radiative Transfer Model* [11], which is the model used in Synoptic SARB to model the energy transfer (in the form of electromagnetic radiation) between and across the earth and the top of the atmosphere. The Fu-Liou model takes into account absorption, emission and scattering of the radiation. Energy can be lost due to absorption, gained by emission, while redistribution can occur by scattering.

4.1.1 ***Subroutines of interest and evaluation of functional correctness.*** The algorithmic implementation of Fu-Liou in Synoptic SARB addresses two sub-cases of radiation: in the longwave and shortwave spectrum (as can be inferred from the names of the corresponding subroutines in Table 1). The subroutines of interest for implementation via GLAF, as identified by NASA scientists, originally span about 700 source lines of code (a per-subroutine break-down is presented in Table 1). This number does not account for lines of code that correspond to data types and variables from imported modules (mainly related to the Fu-Liou radiative transfer model's input, output variables, and custom data types whose part some of the variables are). Due to the nature of the computations, and the fact that these subroutines are part of a much larger code-base with lots of dependencies, we are able to exercise multiple aspects of GLAF auto-parallelization and code generation, including the aspects of code integration this work focuses on, which were not tested or supported before.

For evaluating the functional correctness of the code, we create a wrapper function that calls the GLAF auto-generated subroutines and provides sample values for the required inputs. The imported FORTRAN modules, from which the auto-generated code uses existing variables and custom data types, are used "as is". We then conduct a step-by-step unit testing of the code, and a code-wide side-by-side comparison of the results from the execution using the GLAF auto-generated subroutines, against the results from executing the original code. We repeat this process for both the serial and parallel versions of the auto-generated code and verify that the auto-generated code is functionally equivalent to the original. For the parallel version of the auto-generated code, as an additional inspection step, we manually verify the correctness of the OpenMP directives and associated clauses used. After the unit testing process via the wrapper, we substitute the original subroutines in the Synoptic SARB code with the ones that were implemented and whose code was automatically generated by GLAF. We subsequently run the provided Synoptic SARB test suite and corroborate the correctness of the results in real-world test scenarios.

Table 1: Subroutines implemented using GLAF

Subroutine name	SLOC
lw_spectral_integration	75
longwave_entropy_model	422
sw_spectral_integration	50
shortwave_entropy_model	13
entropy_interface	46
adjust2	38

4.1.2 ***Performance evaluation.*** In this section we present and discuss the performance of the parallel code for Synoptic SARB, as automatically generated by GLAF in FORTRAN (the original code-base that the GLAF-generated code needs to be integrated with is in FORTRAN, too). The code was compiled with gfortran (v4.9.2) at the -O3 optimization level and was run on a Linux-based machine (Debian Linux 8.6, kernel v3.16) with an Intel Core i5-2400 CPU (four cores clocked at 3.10 GHz). Figure 5 shows the results of performance evaluation across different implementations of Synoptic SARB kernels. Specifically, it shows the speed-up of the GLAF serial implementation (*GLAF serial*) and incrementally optimized GLAF parallel (four threads) implementations (*GLAF-parallel v0-v3*) versus the original serial Synoptic SARB implementation (*original serial*). The details of each implementation are given in Table 2.

First, we observe that the GLAF generated code in its serial implementation (*GLAF serial*) performs slightly worse than the original serial implementation (*original serial*). In our prior experience with GLAF we have both observed cases like this, as well as cases where the GLAF auto-generated serial implementation outperforms the original serial. As noted in §3.3, the GLAF GPI programming enforces an implicit structure in a program, wherein any loops within a step except at the outermost level (e.g., within an *if* statement) need to be implemented as a new GLAF function. If the functions are not inlined by the compiler, there is a certain calling overhead that depending on the algorithm may negatively affect its execution time. Also, certain compiler optimizations may be difficult/impossible when code spans multiple functions. In the opposite case, smaller functions can be automatically inlined by the compiler, and the implicitly enforced structure can even help with certain function-level compiler optimizations. In any case, the potential for such (small) performance deterioration is normally outweighted by the parallelism benefits of GLAF auto-generated (parallel) code.

GLAF-parallel v0 in Figure 5 corresponds to the implementation that contains the parallel code generated by GLAF. As we describe in Table 2, this includes OpenMP directives that surround all loops that the parallelism detection back-end has identified as parallelizable. This implementation performs about 50% slower than the original serial implementation, highlighting the disadvantage of a "one-size fits all" approach when it comes to applying OpenMP directives to eligible loops. Currently, GLAF does not contain a means of evaluating whether a loop is better off without OpenMP directives. As future work, we suggest the incorporation of a performance prediction/modeling back-end that will guide the automatic code generation in a more intelligent way (e.g., selecting SIMD directives, instead of OpenMP, or neither). In the cases *GLAF-parallel v0* to *GLAF-parallel v3* we incrementally remove OpenMP directives from *three distinct cases* of loops and provide insights on performance, thereby highlighting the potential impact of intelligent automation of such removal in future work.

GLAF-parallel v1 is based on *GLAF-parallel v0* with the difference that we have manually removed OpenMP parallelization directives from *two* types of loops: a) initialization of arrays (grids) to zero value, and, b) initialization of arrays with a single value loaded from another array. These two are typical cases where the compiler can apply optimizations that outperform thread-level parallelism (and its associated overheads). For instance, initializing an array to zero can be done via *memset* emitted by the compiler, as an optimization, for eligible loops. Alternatively, SIMD operations can be used for loading and assigning values from an input array. This can be observed in the performance results: speed-up increases from 0.48-fold (*GLAF-parallel v0*) to 0.66-fold (*GLAF-parallel v1*).

In *GLAF-parallel v2* we proceed with removal of further OpenMP directives from otherwise parallelizable parts of the code. Specifically, we remove OpenMP directives from all remaining single loops of the code. This contains loops with one-line assignments that contain mathematical operations, few lines (two to four) of similar assignments, as well as loops that contain reductions (and that have been identified as such by GLAF auto-parallelization back-end). In these cases, we identified (as above, via inspection of the compiler optimization reports and/or generated assembly code) that the compiler emits SIMD instructions or proceeds with loop unrolling (when the number of loop iterations is low). As with the previous cases, the overhead of thread-level parallelism is not justified over data-level parallelism (SIMD) or instruction-level parallelism (ILP). Hence, removing the OpenMP directives from the corresponding loops and allowing the compiler to apply its own optimizations increases the speed-up over the original serial implementation to 1.11-fold.

Last, in *GLAF-parallel v3* we remove OpenMP directives from double-nested loops that contain one or a few statements without including any control structure (if/else statements). Again, observation of the compiler optimization reports reveals that the compiler can identify the loops as parallel and applies SIMD optimizations or loop unrolling. Effectively, this leaves the GLAF auto-generated parallel code with OpenMP directives in two large loops in the *longwave_entropy_model* subroutine. The compiler fails to identify these loops as parallel, hence the performance of the GLAF code that includes the appropriate OpenMP directives (*"OMP PARALLEL FOR"* with the necessary *"PRIVATE"* clause) outperforms the original serial implementation by 1.41-fold.

As we mention at the start of §4.1.2, the parallel implementations shown correspond to execution with four threads. Experimentation with varying number of threads (up to the maximum of 8 threads for our test CPU) showed that four threads provides the optimal performance. Figure 6 shows the performance of the fastest implementation (*GLAF-parallel v3*) as the speed-up over the *GLAF-serial* implementation. The parallel version that uses one thread presents a minor slow-down (0.92-fold over *GLAF-serial*) due to the OpenMP run-time associated overhead (which is present, despite using a single thread). Two and four threads yield speed-ups of 1.24- and 1.59-fold, respectively, while adding more threads (e.g., 8) yields diminishing returns (0.7-fold). For the latter, one should consider the

Table 2: Synoptic SARB implementations

Implementation	Description
original serial	Original serial implementation
GLAF serial	Serial implementation generated by GLAF
GLAF-parallel v0	Parallel implementation generated by GLAF with OMP directives in all applicable loops
GLAF-parallel v1	GLAF-parallel v0 with removed OMP directives from initializations to zero or with single value assignments (loads)
GLAF-parallel v2	GLAF-parallel v1 with removed OMP directives from simple single loops
GLAF-parallel v3	GLAF-parallel v2 with removed OMP directives from simple double loops

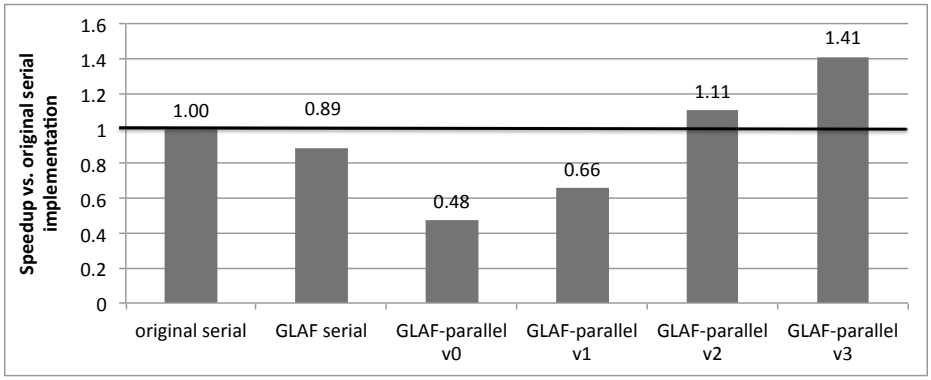

Figure 5: Performance results: Speed-up of GLAF-generated versions versus the original serial implementation of Synoptic SARB kernels of interest

fact that our test CPU has a maximum of four physical cores (up to 8 logical cores with hyper-threading). Also, the double-nested loops that are annotated with OpenMP directives by GLAF consist of a total of *2 x 60 = 120* iterations (since GLAF generates a *"COLLAPSE(2)"* clause). This is a small number (also considering the complexity of the loop code), hence more threads entail overhead (OpenMP run-time, memory coherence, etc.) that cannot be amortized. This adds another dimension to consider in our future efforts for further automation of the GLAF optimization process.

Overall, we find that GLAF auto-generated parallel code (with appropriate selection of which parallelized loops to keep) performs 1.41 times faster than the original serial implementation. While this may appear underwhelming for a multi-threaded execution with four threads, it does not reflect on the capabilities of GLAF itself. Rather, performance owing to parallelization is limited by the workload itself (in the original algorithm) as described above (i.e., small number of iterations in parallel loops). Also, serial parts of the algorithm, between the parallel section can limit the maximum parallelism (*Amdahl's Law*). As far as other opportunities for parallelism are concerned, most loops prove to be amenable to automatic optimizations by the compiler (mainly in the form of SIMD or loop unrolling optimizations). The amount of optimizations the compiler can perform depend on the algorithm itself (in previous works [14, 15] we have seen instances of GLAF parallel code to far outperform the compiler's optimizations). In any case, GLAF does not seek to compete against the compiler, but rather to complement it in an attempt to provide the best achievable performance for domain scientists in automated ways.

4.2 FUN3D: Jacobian Matrix Reconstruction

The GLAF implementation of the FUN3D Jacobian Matrix Reconstruction mini-app decomposes the original function into five sub-functions. The first (*EdgeJP*) represents the outermost scope, which initializes critical module-wide constants and loops over cells of the simulation. The second (*cell_loop*) represents the computation required within a cell and includes interior loops over nodes, faces, and edges within the cell. The node and face loops are parallelized within cell_loop, as they do not include any further nesting, whereas the loop over edges calls out to the third sub-function, edge_loop. The final two sub-functions (*angle_check*) and (*ioff_search*) are both called by cell_loop and respectively represent a check for a cell-face angle in excess of some threshold (which results in skipping the rest of the cell's contribution), and a search for the offset at which a node's contribution should be recorded in the final output data structure.

4.2.1 ***Functional Correctness and Adaptations.*** To evaluate the correctness of the GLAF-generated version, the produced code is integrated with the rest of the program's code, and output at various stages is compared to that produced by the original on a

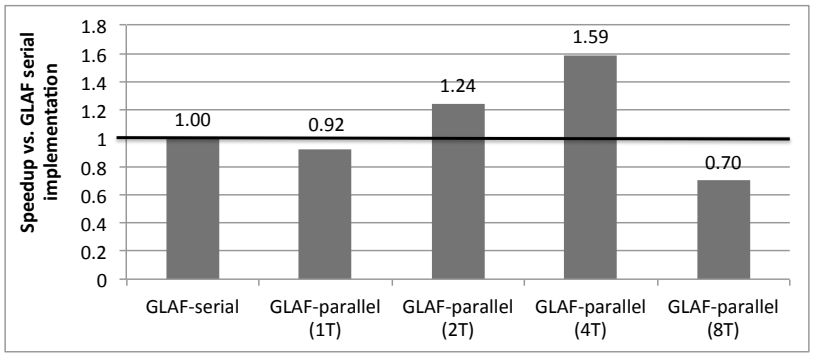

Figure 6: Parallel scalability: Speed-up of fastest GLAF-generated version (*GLAF-parallel v3*) with varying number of threads (*T*) versus GLAF serial implementation of Synoptic SARB kernels of interest

representative data set provided by NASA. The dataset consists of approximately one million cells and ten million edges. Additionally, the dataset includes a reference root mean square of the output arrays that is automatically checked at a 10^{-6} (absolute) tolerance after all cells have been processed to ensure against any major floating point errors in the computation - critical when performing parallel summation, as required by the kernel.

The implementation of Jacobian matrix reconstruction via GLAF uncovered the need for various enhancements in the code-generation and auto-parallelization back-ends. While we were able to automate most, there was an array of manual code tweaks performed on the GLAF generated code to satisfy final correctness of the integrated parallel implementation of the program. These are relatively simple engineering tasks and we plan to address in subsequent updates but we report for the sake of completeness. Such manual tweaks included the below:

- Function-scope arrays from inner functions are applied the "save" attribute, when in a parallelized region, to reduce excess dynamic reallocation.
- Module-scope (and some function-scope) arrays are explicitly declared as *private* or *thread-private* as appropriate based on the selected parallelization level.
- Some module-scope arrays are replaced with pointers and *copyprivate* clauses when supporting nested parallelism (to share to multiple threads in an inner scope without sharing to threads at an outer scope).
- Reduction clauses are updated to specify multiple reduction variables when a loop has effectively more than one output (common when computing multiple values from all nodes in a given cell).
- Atomic update clauses are added to parallel updates to module-scope arrays generated by GLAF or imported from other modules.
- An OpenMP critical clause is added to the early-return section of ioff_search, to ensure the correct offset is returned.

4.2.2 ***Performance Evaluation.*** Performance evaluation of the above modified GLAF-generated code was performed on a Dell Edge server node. The code was compiled with ifort v2017.1.132 at the -O3 optimization level and with support for the AVX2 instruction-set. [2] Our Dell Edge node runs Debian Linux 8.9 (kernel 3.16.0-4-amd64), contains two Intel Xeon E5-2637 v4 CPUs (4 cores/8 threads, each) clocked at 3.50 GHz, and 8x32GB of DDR4-2400 ECC RAM. The original serial implementation was used as the baseline and was compared to the GLAF-generated code in all combinations of parallelization and non-reallocation options. Finally, the original serial version was manually parallelized at the same level as the best-performing GLAF implementation to provide for an additional comparison case.

Figure 7 gives an overview of the speed-up achieved versus the original serial version (results from parallelizing the cell-face angle check are omitted as it had negligible performance impact). Of note we find that the decomposition of the multiply-nested loop into separate GLAF functions can have a negative performance effect due to the sheer number of arrays that are reallocated within interior contexts. For example the innermost edge loop has 50 dynamically allocated temporary arrays and is called an average of 10 times per cell in the provided test case. Once this dynamic reallocation was eliminated via FORTRAN SAVE attributes and manual pointer storage, parallelization began to yield a performance benefit. As future work, an option to GLAF could be added to limit such excessive reallocation automatically. The best performance is achieved when parallelized at the coarsest granularity, that is, the outermost loop over all the simulation cells. Within each cell the maximum number of nodes and edges is limited and thus the opportunity for parallelism is inherently limited too. Consequently, the cost of repeated calls to the more complex parallel loop structures cannot be efficiently amortized. To estimate the trade-off between programmability and performance when using GLAF, we manually parallelized the original serial code at the same outermost scope as GLAF. In this regard GLAF's generation of all possible levels of parallelization drastically eased the search of the optimization space, as well as identifying the 219 variables that needed to be declared as OpenMP private. This manual version ends up outperforming the best GLAF version by almost 2.3-fold. It is once more worth noting that manually creating this implementation from scratch in FORTRAN would be difficult if not impossible for someone without

[2]The entire set of compiler options is (-O3 -ip -align array64byte -fno-alias -g -traceback -qopt-report=5 -std03 -axCORE-AVX2 -convert big_endian -openmp -qopenmp).

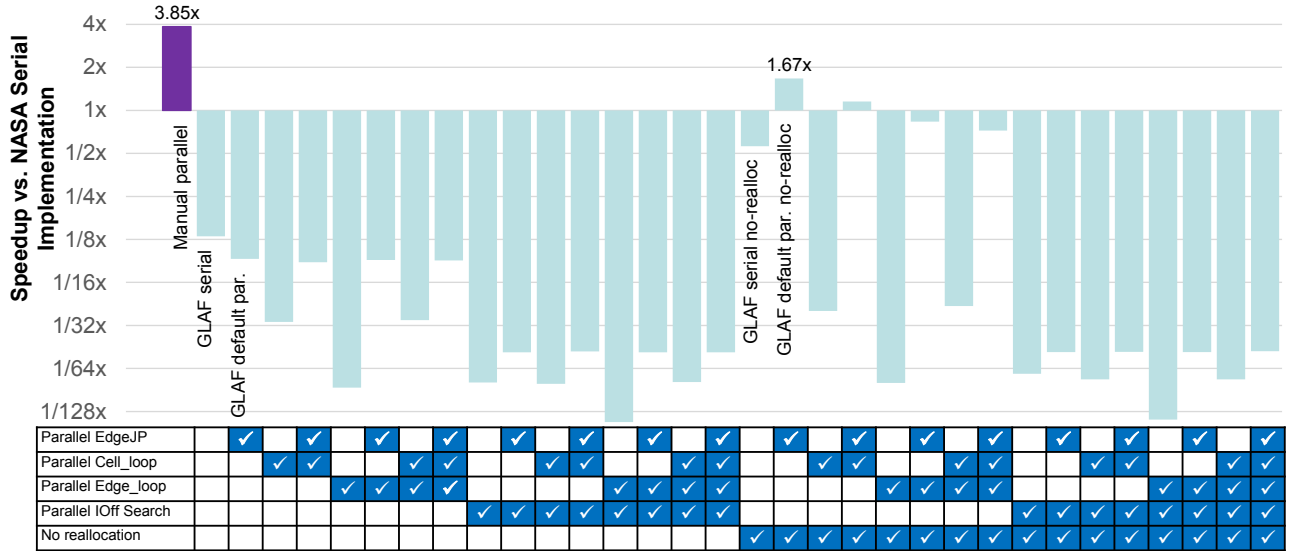

Figure 7: Performance results: 16-thread parallel speed-up of GLAF-generated matrix reconstruction of a 1M cell test case with all combinations of parallelization and no-reallocation options. Manual parallel version (based on best-performing GLAF options), provided for comparison. A colored/checked box indicates the indicated option is "on" in the column above, and a blank box indicates it is "off" (i.e., running the function in serial or with default reallocation behavior).

good parallel programming knowledge, such as domain scientists. Ultimately, GLAF provides a trade-off between programming effort (programmability) and performance.

5 RELATED WORK

Parallel computing is of high importance across scientific domains with many applications in engineering, physics, finance, among others. It has enabled scientists to perform complex computations faster, on larger amounts of data, and/or do so with much higher accuracy than in the past. Parallel programming can prove challenging for many programmers, and even more so for non-expert programmers like domain scientists. In order to facilitate optimized parallel programming for such audiences many approaches have been explored. These include domain-specific languages (DSLs), problem-solving environments (PSEs), auto-tuning frameworks, and specialized programming languages. Examples of DSLs, PSEs, and auto-tuning frameworks include works like ATLAS [9], FFTW [10], PATUS [8], FAST [16] - to name a few. On the other hand, Chapel [5], X10 [6], Julia [3], are some representative programming languages specifically targeting programmability and productivity.

While all the above approaches have their indisputable merits for the specific use case they were conceived, they also come with certain limitations. For instance, as their name denotes, DSLs are specific to a domain (or an even more specific subdomain). For example, ATLAS [9] addresses dense linear algebra code, and FFTW [10] relates to FFTs. This allows them to utilize domain-specific knowledge to achieve the best performance, but lack of generalization poses limitations to their practical utility.

On the other hand certain tools that belong in the categories mentioned above may generate code in a specific language, other than Fortran (e.g., C, CUDA, or OpenCL). Therefore, integrating the generated code within FORTRAN legacy scientific codes is rendered problematic. For example, R-Stream [18] takes C as input and generates parallel C code with OpenMP directives (also supports code generation for the Cell BE processor and CUDA for GPU execution). Our own previous work, GLAF [15], does generate FORTRAN parallel code (among other languages), however it did not provide ways for easy integration and interoperability of auto-generated parallel code with existing FORTRAN code (beyond trivial cases).

All the above notwithstanding, the major problem of the above approaches in relation to the motivating issue of this work lies in the lack of consideration for the integration of auto-parallelized code with *existing* FORTRAN code. Many tools *do* address integration of *their* code (i.e., in a specialized programming language) with code in "traditional" programming languages (like C or FORTRAN). For example, Chapel supports C code integration (e.g., pointers and external C types, variables and constants). Julia, similarly, provides a C interface (e.g., for calling C functions via *ccall()*). To the best of our knowledge, however, such works as the ones described above typically lack provisions for the inverse process, i.e., integration of *their* code within languages like C or FORTRAN. The exception is Julia that supports passing Julia functions as callbacks to C functions (assuming one includes the appropriate Julia library file and links against *libjulia*). Julia2C (j2c) [19] is also able to translate basic Julia types and expressions to C. In our work, we seek to address this gap that exists between software approaches for programmability for parallel programming and the ability to integrate specific kernels within large, existing scientific codes that are typically written in FORTRAN a language used heavily in scientific codes.

6 CONCLUSIONS

In this work, we discussed an important requirement of programming frameworks and tools that seek to enhance programmability and productivity for parallel computing. Specifically, we focused on the need for modular program development, integration and interoperability of a program's kernels developed in such tools with the encompassing program's code. We explored the issues related to the above in the context of our GLAF programming framework and provided a reference implementation of the proposed solutions. Last, we evaluated the new functionalities within kernels of two real-world applications of interest to NASA (Synoptic SARB and FUN3D).

In experimenting with these two large scale, real-world applications, we were able to further exercise all GLAF front-ends, back-ends and overall functionality in concert. This included previously developed capabilities (such as the graphical programming interface, code generation, and auto-parallelization) and the novel functionality of auto-generated code integration with existing encompassing code. Our experiments allowed us to draw useful conclusions for enhancing the automatic code generation of parallel code in this version, as well as current limitations that serve as learnings for further improving GLAF in our future work.

Overall, our experiments with two real-world NASA applications and code integration supplement our prior work that had focused on stand-alone microbenchmarks. We showed that the grid abstraction upon which GLAF is built is generic enough and can represent real-world application scenarios. Last, we illustrated that GLAF itself is now more robust and can generate correct serial and parallel FORTRAN code for program kernels that can in turn be transparently integrated with the respective existing encompassing programs.

ACKNOWLEDGMENTS

This work was supported in part by NSF I/UCRC IIP-1266245 and NASA via the NSF Center for High-Performance Reconfigurable Computing and the Institute for Critical Technology and Applied Science (ICTAS). The authors would like to acknowledge Dana P. Hammond, Louis Nguyen, and Erik Nielsen for their guidance with regards to the case-study applications, as well as their overall feedback on the GLAF project.

REFERENCES

[1] W Kyle Anderson and Daryl L Bonhaus. 1994. An Implicit Upwind Algorithm for Computing Turbulent Flows on Unstructured Grids. *Computers & Fluids* 23, 1 (1994), 1–21.

[2] W Kyle Anderson, Russ D Rausch, and Daryl L Bonhaus. 1996. Implicit/Multigrid Algorithms for Incompressible Turbulent Flows on Unstructured Grids. *J. Comput. Phys.* 128, 2 (1996), 391–408.

[3] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *arXiv preprint arXiv:1209.5145* (2012).

[4] Jan-Renee Carlson. 2018. FUN3D Fully Unstructured Navier Stokes. (March 2018). https://fun3d.larc.nasa.gov/

[5] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 20. https://doi.org/10.1145/1094811.1094852

[7] Iris Christadler, Giovanni Erbacci, and Alan D. Simpson. 2012. *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Performance and Productivity of New Programming Languages, 24–35. https://doi.org/10.1007/978-3-642-30397-5_3

[8] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. https://doi.org/10.1109/IPDPS.2011.70

[9] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Comput.* 27, 1 (2001), 3–35.

[10] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301

[11] Qiang Fu and K. N. Liou. 1993. Parameterization of the Radiative Properties of Cirrus Clouds. *Journal of the Atmospheric Sciences* 50, 13 (1993), 2008–2025. https://doi.org/10.1175/1520-0469(1993)050<2008:POTRPO>2.0.CO;2

[12] Peter Gnoffo. [n. d.]. Updates to Multi-Dimensional Flux Reconstruction for Hypersonic Simulations on Tetrahedral Grids. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. 1271.

[13] Edward Kizer and Norman Loeb. 2018. NASA CERES: Clouds and the Earth's Radiant Energy System Information and Data. https://ceres.larc.nasa.gov. (April 2018).

[14] K. Krommydas, R. Sasanka, and W. Feng. 2016. Bridging the FPGA programmability-portability Gap via automatic OpenCL code generation and tuning. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 00. 213–218. https://doi.org/10.1109/ASAP.2016.7760796

[15] K. Krommydas, R. Sasanka, and W. C. Feng. 2015. GLAF: A Visual Programming and Auto-tuning Framework for Parallel Computing. In *2015 44th International Conference on Parallel Processing*. 859–868. https://doi.org/10.1109/ICPP.2015.95

[16] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-Solution Space Model. In *ACM International Conference on Supercomputing (ICS)*. 10. https://doi.org/10.1145/2751205.2751214

[17] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. 2016. An Evaluation of Emerging Many-Core Parallel Programming Models. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 10. https://doi.org/10.1145/2883404.2883420

[18] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-stream compiler. In *Encyclopedia of Parallel Computing*. Springer, 1756–1765.

[19] Hongbo Rong, Todd A Anderson, and Hai Liu. 2015. Julia2C Source-to-Source Translator. https://github.com/IntelLabs/julia/tree/j2c/j2c. (January 2015).