Making Fortran Legacy Code More Functional

Using the BGS* Geomagnetic Field Modelling System as an Example

Hans-Nikolai Vießmann
Heriot-Watt University
hv15@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University
s.scholz@hw.ac.uk

Artjoms Šinkarovs
Heriot-Watt University
a.sinkarovs@hw.ac.uk

Brian Bainbridge
British Geological Survey
bba@bgs.ac.uk

Brian Hamilton
British Geological Survey
bham@bgs.ac.uk

Simon Flower
British Geological Survey
smf@bgs.ac.uk

ABSTRACT

This paper presents an application case study of the British Geological Survey's (BGS) Geomagnetic Field Modelling System code. The program consists of roughly 20 000 lines of highly-tuned FORTRAN MPI code that has a runtime of about 12 hours for a signal execution cycle on a cluster utilising approximately 100 CPU cores. The program contains a sequential bottleneck that executes on a single node of the cluster and takes up to 50% of the overall runtime. We describe an experiment in which we rewrote the bottleneck FORTRAN code in SAC, to make use of auto-parallelisation provided by the SAC compiler. The paper also presents an implementation of a foreign-function interface, to link the SAC kernel with the FORTRAN application. Our initial performance measurements compare the SAC kernel performance with the FORTRAN bottleneck code; we also present results using an OPENMP Fortran implementation. Our figures show that the SAC-based implementation achieves roughly a 12.5% runtime improvement, and outperforms the OPENMP implementation.

CCS Concepts

•**Applied computing** → *Environmental sciences;* Mathematics and statistics; •**Computing methodologies** → *Parallel programming languages;* •**Software and its engineering** → *General programming languages; Compilers;* •**Computer systems organization** → Single instruction, multiple data; Multicore architectures;

Keywords

Fortran; Single-Assignment C; High-Performance Computing; Functional Programming; Eigensystem; Foreign Function Interface;

*British Geological Survey — is the world's oldest geological survey. A member of the United Kingdom's public funding body, the Natural Environment Research Council, the BGS is the custodian of much of the country's geoscientific information as well as a source of geoscience expertise. More information can be found at www.bgs.ac.uk.

IFL '15, September 14-16, 2015, Koblenz, Germany

 ISBN 978-1-4503-4273-5/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2897336.2897348

1. INTRODUCTION

The lambda calculus' Church-Rosser property suggests that functional programming languages should offer opportunities for efficient parallel execution. Research over the last four decades has produced excellent results that demonstrate how this conceptual advantage can be turned into real speedups on parallel hardware [15, 18, 26, 27, 32, 35]. Despite these advances, functional programming has not yet made it into mainstream high-performance computing (HPC).

There are many reasons for this, but the key issue is that the HPC industry has invested heavily into imperative programming languages, such as legacy FORTRAN codes. In particular, FORTRAN has many features that make it the language of choice for performance-hungry applications. It has long been available, and its design is amenable to compiler optimisation. Most programs consist of procedures containing nested loop constructs that iterate over arrays. The use of FORTRAN for HPC applications and the design of the language, consequently, has driven research into advanced compiler optimisations which, in turn, has further improved the performance of generated code. Alongside the continuous improvement of FORTRAN compilers, hand-tuning of these imperative applications has given them highly competitive levels of performance. Given this scenario, it is not very surprising that other programming languages, including those from the functional domain, struggle to take hold in the HPC domain.

FORTRAN's limitations with respect to auto-parallelisation inspired the development of HPF (High-Performance FORTRAN), were alleviated by the manual adaptation of codes with explicit parallelism, using MPI, OPENMP, or both.

Several prominent examples show that compiler-generated codes generated from highly specialised, typically functional, domain-specific languages (DSLs), can outperform legacy codes, in both parallel and sequential settings. Examples of this approach are the Spiral project [30] for signal processing and the Diderot project [21] for image processing. While this is an indicator for the validity of the initial claim, i.e. the conceptual advantages of the functional setting for parallel executions, it only covers a few rather specific application areas. Furthermore, a DSL-based approach typically requires re-writing large parts of legacy code, if not entire applications.

In this paper, we investigate a different approach towards improving legacy FORTRAN HPC applications. Starting from a parallel, commercial FORTRAN code base we identify its sequential bottleneck, re-implement this bottleneck code in SAC and call the compiled SAC code from the FORTRAN context. The most important aspect of this approach is that it is applicable to a wide range of existing FORTRAN applications, without requiring either costly re-

writes of entire applications or the even greater costly development of a DSL. Key to the success of such a mixed-language approach is a versatile, easy to program, and low-overhead, foreign-function interface (FFI). Our paper sketches the various design choices we have made in the context of SAC to produce such an FFI. We demonstrate our approach by means of a case study, the BGS Geomagnetic Field Modelling System (BGMS), a 20 000 line long FORTRAN application with MPI code that is commercially run on the BGS in-house 448-core cluster.

The main contributions of this paper are:

- a description of the SAC FFI, describing the interface itself, as well as delineating its design space.
- an extension of the FFI for SAC that allows data from the FORTRAN world to be safely passed to the functional world and back without adding any overhead due to superfluous copying of data.
- multi-threading support.
- a description of the use case detailing our rewrite of the FORTRAN bottleneck in SAC and what the effect the rewrite has in comparison to the original code, and
- some performance measurements comparing runtimes and speedup between the original FORTRAN code and our rewritten implementation. Additionally, we compare the SAC version with an OPENMP implementation in FORTRAN.

The paper is organised as follows: Section 2 looks more closely at the BGMS and how it works, and identifies the sequential bottleneck. In Section 3, we give a brief overview of SAC and in Section 4, we present our work in creating a FORTRAN interface on top of SAC's FFI, including discussion about design considerations for our FFI. In Section 5, we detail the steps we take to re-implement the sequential bottleneck in SAC. In Section 6, we present our performance results and provide some analysis. In Section 7, we present some related work and in Section 8, we give our conclusions.

2. THE BGS GEOMAGNETIC FIELD MODELLING SYSTEM

The BGS Geomagnetic Field Modelling Systems (BGMSs) computes mathematical models of Earth's magnetic field, constructed from geomagnetic observatory data around the globe and from satellites in orbit, such as through the ESA Swarm project [7]. From the computed models, in turn, new models are derived. For example, the International Geomagnetic Reference Field (IGRF) model provides a standard mathematical description of the Earth's main magnetic field [33]. It is used widely in studies of the Earth's deep interior, its crust and its ionosphere and magnetosphere.

Another model derived from the BGMS is the World Magnetic Model (WMM); it aims to provide a reliable estimate of the magnetic field's strength and direction on or near the Earth's surface; it is used by various navigation systems [6].

Finally, the BGS Global Geomagnetic Model (BGGM), also derived from the BGMS, is widely used in the oil industry for directional drilling [5]. These models use millions of input data elements to estimate thousands of coefficients that describe the Earth's magnetic field.

As can be seen from these examples, the significance of the BGMS output is quite high; therefore, it is very important for the BGMS model to be both precise and accurate while not taking an unreasonable amount of time to compute given the large volume

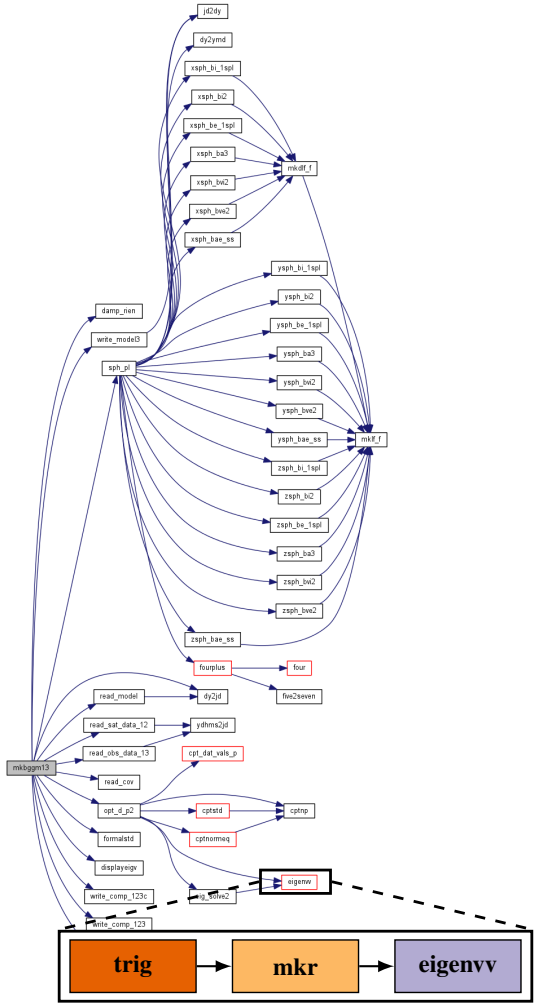

Figure 1: Diagram displays a basic call-graph of the BGMS application. The foreground shows the part of the BGMS that computes the eigensystem, making calls to the subroutines `trig`, `mkr`, and `eigenvv`.

of data that needs to be processed. This can become more difficult as the data sets grow or if the data itself increases computational difficulty; both cases can increase the runtime dramatically. That is why the BGMS application is run on a many-node cluster.

The BGS in-house cluster comprises 28 homogeneous compute nodes and one head node. Each compute node contains two Intel Xeon CPUs E5-2650v2 processors, running at 2.6 GHz, with hyper-threading disabled, giving 16 physical cores, each with 1 thread, per compute node. The total amount of RAM is 128 GB per compute node.

The BGMS application is written in FORTRAN-95, making use of MPI for inter-node communication. Figure 1 shows a rough approximation of the call-graph of the BGMS. As part of its execution cycle, BGMS generates a matrix of normal equations made up of about 7000×7000 elements. This matrix is passed to the subroutine `eigenvv`, which after calling the `mkr` and `trig` subroutines (see Figure 1), computes the eigenvalues and eigenvectors of the matrix. The computed eigensystem is then used in later stages of the BGMS. For clarity, when we talk about the `eigenvv` application, we refer collectively to these three subroutines `trig`, `mkr`,

and `eigenvv` as a single entity; otherwise we mean specifically the `eigenvv` subroutine.

The BGMS has a runtime of about 12 hours for a single execution cycle, during which it typically runs more than once; close to 50% of that time is spent in `eigenvv`, called multiple times throughout the execution cycle. It executes as a single-threaded process on a *single* node of the cluster. This subroutine provides an ideal candidate for our approach, as it is clearly a performance bottleneck and is relatively small — about 200 lines of code. As it is being executed with a single thread, there is potential for using the auto-parallelisation capability of the SAC compiler in order to make better use of the available CPU cores.

Although BGS has tried hand-parallelising the `eigenvv` subroutine (including `mkr` and `trig`), they were unable to achieve significant performance gains. Also, the effort to implement the parallel code without harming the existing robustness of the BGMS was considered too high. Furthermore, implementing a different, more parallelisable, algorithm to compute the eigensystem would require a large amount of effort and time to ensure that the robustness of the BGMS is not comprised. Since the model generated by the BGMS is considered critical for its end users, any changes to the code base that might compromise the accuracy of the generated model is unacceptable. That is why rewriting `eigenvv` in SAC to take advantage of our compiler optimisations and auto-tuning capabilities is the focus of our work.

3. SINGLE-ASSIGNMENT C

SAC is a data-parallel array programming language. One of the main goals of the SAC project is to provide a fully automated tool chain with implicit support for various parallel architectures, i.e. the syntax of the language does not support annotations or provide commands that prescribe where to run the code, or which data structures to materialise in memory at runtime. Those choices are made by the compiler. If the compiler cannot decide statically, it generates code variants and chooses the appropriate one at runtime.

The class of architectures that SAC currently supports include shared memory processors (SMPs), CUDA GPUs and distributed clusters with MPI support. Because SAC is purely functional, i.e. data is immutable and there are no side-effects, so the compiler is able to apply aggressive optimisations to the code for improved performance.

When SMP is used as a target architecture, the internal tasks are mapped directly into operating system threads — no light-weight threads are used. When executing SAC programs with multiple threads, the assumption is that the number of threads that will be exclusively available to the SAC program is known in advance and is being passed as a parameter to the executable. After the number of threads is chosen, and execution of SAC program is started, the threads are not be reclaimed until the program terminates. Although work is underway to change this, for the time being this is a given. This restriction makes the choice of the number of threads a very important decision. Another aspect of the SAC runtime system is that there is no guarantee that synchronisation of threads is done via locks. Depending on some compiler internal decisions, the SAC compiler may or may not choose to implement thread synchronisation by means of busy waiting.

Arrays are the fundamental data structure of the SAC language. Conceptually, all values in SAC are multi-dimensional arrays, in which each array carries its rank and the shape (extent) of each array axis. All array references compile into pointers to continuous chunks of memory; a row-major linearisation scheme is used for indexing.

Memory management is implicit at the language level; at runtime, non-deferred garbage collection is implemented using reference counting. This offers an opportunity to reuse arrays at runtime, in cases where no further array references exist. All allocations are made through the standard `malloc` interface. They may make use of the system `malloc`, but the SAC runtime system provides an alternate implementation of the heap manager which provides better performance, particularly in the SMP case [12].

Data parallelism is achieved by means of an explicit construct called the `with`-loop [15], which is a generalised map/reduce combinator.

SAC also provides stateful objects, enabling fully fledged I/O from within SAC. The language does so by means of uniqueness types, allowing purity to be maintained, despite an apparently imperative syntax [14]. More information about SAC and how to use it can be found in [15].

4. SAC TO FORTRAN FFI

Designing and implementing FFIs that are robust and easy to use is very challenging, especially when we are interested in bidirectional FFIs, in which a language *A* can call language *B*, and language *B* can call *A*. In our particular situation, we have to deal with generic FFI-related problems as well as with specific differences between SAC/C and FORTRAN. The overall design of SAC-FORTRAN FFI piggybacks on the existing SAC-C FFI and C-FORTRAN interface, which is a part of the most recent FORTRAN standards. Although such a scheme might seem to be over-complicated, we demonstrate in the rest of Section 6 that it turns out to be powerful, yet lightweight.

We start our discussion by pointing out generic FFI-related problems and our solutions to them, then discuss the SAC-, C-, and FORTRAN-specific aspects of the FFI. We demonstrate that the simplicity and implicitness of SAC can be used as an advantage, and that most of the problems and complex techniques used in other settings can be avoided.

4.1 FFI Challenges

Any FFI has to deal with the differences in treating memory, types, and parallelism. An additional challenge arises from the fact that one language is imperative, while the other is not, leading to the issue of dealing correctly with state.

Memory Management.

If, prior to the FFI being called, all relevant values from the environment of language *A* are copied into the environment of language *B* and, afterwards, copied back, then memory management does not cause any troubles. However, such an approach is not always feasible as sizes of the relevant data structure can be very large, leading to performance degradation. Arrays sizes within the BGMS application typify this problem.

A better approach is to avoid copying and, instead, pass a reference from language *A* to *B*, and get a reference back at the end of the FFI. In order to do this, we have to bear in mind that memory systems of the languages may operate differently. Memory management can be explicit, i.e. when a programmer has to allocate and deallocate objects manually; or implicit, when reference counting and/or garbage collection are being used. When both languages use explicit memory managers, the languages have to make sure that objects allocated by language *A* can be modified and deallocated by language *B* and vice versa. In this case, a programmer will have to ensure that allocation/deallocation is done correctly. To allow modification and deallocation of foreign objects, languages usually agree on the memory allocator they both use.

If at least one of the languages uses an automated memory man-

agement system, then there has to exist an external interface to that system. A language *A* may want to register its object with the garbage collector of language *B*, ensuring that this object can be used safely within *B*. For instance, if one passes a pointer from C to HASKELL and wants to use it as a valid object within HASKELL. Also, a language *A* might need to notify its own garbage collector that a certain object should not be deallocated because it is being passed to a language *B* via the FFI.

SAC uses reference counting and C/FORTRAN use explicit memory managers. When calling SAC from C, prior to the FFI call, one has to obtain a reference count on the pointer that will be passed as an argument to the SAC function. Because of SAC's implicit memory management, once the pointer is passed to the SAC function, it is no longer available within C — SAC's memory management system will deal with the reference counting. For SAC function return values, these are passed as pointers without any reference counting, meaning that within C they need to be explicitly freed. When calling C from SAC an object can be passed directly to C. The only guarantee required is that the object won't be deallocated explicitly, e.g. by calling `free` — deallocation will happen on the SAC side.

Type safety.

Type safety of FFI-called functions is a non-trivial problem, because type safety breaks when the type system of a language *A* provides stronger or weaker guarantees than the type system of *B*. Some of languages, such as C, might not be type safe at all. Also, when passing arguments of a certain type that exist in language *A* to language *B*, one has two options. The first one is to make sure that all the types of *A* are expressible in *B*, which can be difficult, e.g. when user-defined types may exist only in terms of the given program. This means that additional glue-code must be provided, to express the relevant types of *A* in *B*. Alternatively, and this is a very common practice, arguments are passed via the most generic type, like `object` in JAVA or `void *` in C, and type conversion is performed at the level of the embedded language. This is when type safety breaks. One solution to this problem is described in [10], which proposes a type inference system that tracks types by embedding the original type information into the type of the target language as they are transformed by the FFI. Doing so provides enough information to allow both languages to do type checking and prevent programming errors.

With SAC, the C-FFI is designed to be fairly restrictive, limiting the types to those supported in both languages (e.g. integers, single- and double- floating point, etc.). This makes it possible to eliminate most of the type safety considerations, especially those relating to user-defined types. First of all, no data structures other than arrays are used. Multi-dimensional arrays in SAC are handled in exactly the same way as they are in C99. The only little extra that comes with every array in SAC is a structure that keeps reference counting information. This information is decoupled from the memory that stores an array, therefore arrays can be literally passed from SAC to C and vice versa.

The only other kind of types that are present in SAC are types for stateful objects. When those objects are passed from SAC to C, no extra care is needed due to imperative nature of C. See a more detailed discussion about stateful objects later on. A C data structure cannot be represented in SAC, so when it needs to be passed via FFI, it has to be bound to an abstract type, and accessor functions have to be provided by the C side. Unfortunately this process is not automated, so the task is left to the programmer.

Parallelism.

When both languages that are connected by an FFI provide a means to run code in parallel, the following problems have to be considered:

1. resource awareness — if both languages are using threads, sooner or later they will have to map those threads on physical cores, in which case the host and embedded language may start competing for resources which can cause significant overheads. The same might happen if both languages are mapping parallelism on devices like GPUs, FPGAs or other kind of special purpose hardware;

2. synchronisation — if an FFI call gets an object that is shared between the threads of the host and embedded language, then one needs to answer the question of how to avoid data races or deadlocks.

One of the possible approaches used for example in CONCURRENT HASKELL [28] is to consider different thread abstraction levels for host and embedded languages and to provide explicit mapping abstractions and synchronisation mechanisms to a programmer. This assumes that the language has an explicit operator, like `fork`, to spawn a thread. However, this still leaves an open question of calling several multi-threaded functions from the host language in parallel. That is, do we need to notify the environment of the embedded language that multiple groups of multiple threads will be run in parallel?

There are no explicit `fork`-like operations in SAC, and it is up to the compiler to decide to execute code in parallel. When evaluating a function, the compiler may generate several variants of it, for both serial execution and parallel execution. The SAC runtime system creates a thread pool with a fixed number of threads. How many threads are to be used is specified by a parameter passed to the executable. When evaluating a function at runtime, if enough threads have been created and a parallel-executable variant of the function exists, the runtime system distributes the computation among the threads. This is all done implicitly, with no annotations or other input from the programmer.

In the case of the FFI, the situations is not as straightforward, because SAC's runtime system assumes that it has exclusive control of all available CPU-cores. In the context of an FFI, this may not be the case, as the SAC code might be called from an already multi-threaded context or there may be several concurrent calls to SAC code [31]. To circumvent the associated issues with these scenarios, two things are done. First, the SAC compiler generates a new SAC function variant, designed to be called from an *external* context, whereby it is explicitly associated with a thread pool. Secondly, the existing SAC/C-FFI exposes the runtime system by providing functions to create and destroy thread pools. This action exposes a *hook* through which a multi-threaded SAC function can associate itself with the thread pool. With this setup, the runtime system can be selectively activated to start running just before executing SAC code, minimising overheads associated with it. An interesting feature of this design is that multiple thread pools can be generated and SAC functions executed concurrently therein, all through the SAC/C-FFI.

The problem of executing multiple multi-threaded SAC functions in parallel can be solved by passing an extra flag when compiling a module. This flag makes the generated code protect the section that updates the reference count with a lock. The problem of locking the data that is shared between the threads of a host language and the threads of the embedded language is, currently, left to the programmer.

Handling Statefulness.

Dealing with state in pure functional languages is non-trivial, but when two languages are connected via an FFI, one has to deal with potential differences in treatment of the state in both languages. The simplest approach would be to treat any foreign function as being stateful. In this case, the only mechanism required is the ability to create stateful objects in both languages. This simple, but conservative approach precludes the ability to share results of a pure function call with identical arguments. A better approach would be to introduce annotations for pure and stateful functions and propagate them, either manually or automatically.

When SAC is called from C, there is not much we can do, as standard C does not provide any mechanisms to annotate that a function is side-effect free. When C is called from SAC, the language provides a construct to annotate that a function has a side effect. If such an annotation is not provided, the function is considered pure. For stateful C functions, a programmer has to create an object of a special type and indicate, at every call, that the function has an effect on this object. All other technicalities are handled by the compiler's uniqueness type resolution phase [14].

4.2 Fortran-specific FFI issues

The SAC compiler uses C as its target language, so we have two options when building a SAC-FORTRAN FFI: we can either implement a new backend for SAC, to output FORTRAN code, or we can investigate how to call C from FORTRAN and use the existing SAC backend. To implement a new backend is a major undertaking, and there would not be much to gain by doing so as in this particular case C and FORTRAN are reasonably compatible. We rely on this compatibility to call both SAC functions from FORTRAN and FORTRAN functions from SAC.

Specifically, we show how to extend the SAC-C FFI to generate a FORTRAN module that exposes SAC functions from within the FORTRAN context. Additionally, we discuss a number of technical difficulties and design choices that we encountered.

C and Fortran Interoperability.

One of the first concerns we have is the variance of existing solutions to support the interfacing of C and FORTRAN functions. It should be noted that FORTRAN provides two types of callable units, unlike C, which makes no distinction; there is a `function`, which has one explicit return value, and a `subroutine` which, through its parameters, can return any number of values. Most commercially available compilers provide their own FORTRAN-C interface mechanism, but lack much in the way of portability between vendors [9, 29]. As such, how functions are exposed and how variables are passed and utilised in both language contexts is somewhat different for each vendor. Examples include data types, which can be differently defined in both contexts, the mechanism for passing variables by value or by reference, and how functions are associated between language contexts. These vendor-specific behaviours potentially render the SAC-FORTRAN FFI dependent on a particular target compiler and its behaviour.

Fortunately, from the FORTRAN Standard 2003 onwards, standardised support for the interoperability between FORTRAN and C programs is proposed, making the interfacing mechanism across compiler vendors portable [20]. This support is made possible through the `iso_c_binding` module, which defines C data types and structures within FORTRAN including C pointers. Additionally, a mechanism is provided that exposes C functions within FORTRAN and makes FORTRAN functions directly callable in C. Furthermore, functions and subroutines are provided to convert between FORTRAN and C data structures.

We base our SAC-FORTRAN FFI on this standard because it provides portability across compiler vendors. It has the drawback that we now require FORTRAN code to be compliant with this standard, but we consider this small compromise worth it, as we gain in type compatibility, correct handling of function parameters, and portable data structure across vendors. For future work, we intend to investigate whether it would be possible to maintain parts of the code within the older standard and use newer standards for the interface parts only.

Function Binding.

The `iso_c_binding` module makes the task of using C functions within FORTRAN consistent and robust. This is achieved by declaring FORTRAN prototypes of the C functions within an `interface` block as `function` or `subroutine` blocks depending on the number of return values, similar to declaring an external function in C. For example, the FORTRAN prototype of a C function `void foobar (int a)` look something like this:

```
interface
  subroutine foo (a) bind (c, name = 'foobar')
    integer(c_int), value :: a
  end subroutine foo
end interface
```

In every block, the parameters of the function prototypes must be defined with the correct type and may also have attributes specified. For example, use of the `value` attribute marks the parameter `a` to be passed by value instead of the by reference. For guaranteed type consistency, the parameter type must be taken from the `iso_c_binding` module; for instance `integer(c_int)` is equivalent to the C `int`. FORTRAN function prototypes are linked to their C counterparts using a `iso_c_binding`, providing a `bind` function attribute indicating to the compiler that the current function prototype *binds* to a specific C function by name.

The `iso_c_binding` module also allows us to call FORTRAN functions from C. Similar to the previous example, this is achieved by using the `bind` attribute as part of a FORTRAN function declaration. The difference here is that the attribute does not bind to a C function, but exposes the FORTRAN function with the given name. This allows the FORTRAN function to be called directly from C with the given name. For instance, `integer increment (a)` can be called from C by setting its `bind` name to *inc*, which is exposed as the C function `int inc (int a)`. In practice, this is what it would look like:

```
function increment (a) bind(c, name = 'inc')
  integer(c_int), value :: a
  integer(c_int) increment ! declare return type
  increment = a + 1
end function increment
```

Our FORTRAN function `increment` takes an integer, increments its value by one, and returns the result. As before, we pass the integer by value by specifying the `value` attribute. With this declaration, we can call the FORTRAN function by declaring it `extern` within our C application:

```
extern int inc(int a);

int main(void) {
  int b = 1;
  b = inc(b);
  return b; // b = 2
}
```

Pointers.

Both C and FORTRAN since Standard 90/95 support a notion of pointers. In C, a pointer is a variable that represents the address in memory of another variable which shares the same type. This

is different from FORTRAN, where pointers are descriptors that act like an *alias* to a variable. Specifically, a FORTRAN pointer links to the value of another variable, not its address as in C. FORTRAN pointers cannot be arbitrarily associated with, as is the case in C. Only variables that are declared with the `target` attribute can have pointers associated to them. In the case of arrays, the pointer can be associated to the array variable, or to a sub-array of the array variable. Because the FORTRAN pointer carries with it the shape of the associated array, it can be used to iterate across array or sub-array.

Though C is strongly typed, C supports variable casting, meaning a variable can be *cast* to another type without copying its value. This is also permitted for C pointers. Such a facility exists in FORTRAN as well, but in the case of pointers there is no *in-place* type conversion possible. Instead the data is copied and associated with a new variable. Because of these distinctions, the conversion from one pointer type to another is not straightforward. Some compiler vendors have implemented a mechanism to convert between FORTRAN and C pointers at FORTRAN subroutine calls or through the use of compiler directives, specifying how the pointer should be treated. This is not consistent across vendors, and can lead to issues in accessing variables across language boundaries.

With `iso_c_binding`, C pointers are treated as a variable type in FORTRAN that can be used to declare variables that either represent literal C pointers or that provide a C-compatible pointer to a FORTRAN variable. To make use of these pointer formulations within FORTRAN, one has to convert between C pointers and FORTRAN pointers. `iso_c_binding` provides two functions, `cloc` and `c_f_pointer` through which a C pointer to a FORTRAN variable can be created, and where a C pointer can be converted to a FORTRAN pointer, respectively. For example, given a C function `void foobar (int *a) { *a += 1; }`, the FORTRAN application looks like this when using `iso_c_binding`:

```
program example
  use iso_c_binding

  interface
    subroutine foo (a) bind (c, name = 'foobaz')
      import
      type(c_ptr), value :: a
    end subroutine foo
  end interface

  integer(c_int), target :: b = 10
  integer(c_int), pointer :: c => b

  call foo (c_loc (c))

  write (*,*) c ! c is now 11
end program example
```

As mentioned previously, when using `iso_c_binding`, we need to create a FORTRAN function prototype which links to our C function. With the above example, we declare `foo`, which binds to function `foobaz`, and specify that its only parameter be a C pointer type (`c_ptr`) which is to be passed by value. As for the FORTRAN programs variables, we declare as before the same variables with the same attributes, but specify that they are of type `c_int` instead. As the parameter of `foo` is a C pointer, we call `c_loc` to return the C address of our variable `c`. Our variable `c` is incremented by one yielding a final value of 11.

Arrays.

Multi-dimensional arrays manifest one of the strongest incompatibilities between C and FORTRAN. Although both languages support multi-dimensional arrays as first-class citizens, they differ in that C arrays are stored in row-major order, whereas FORTRAN arrays are stored in column-major order. For the purposes of interfacing with another language, this poses a problem, as one needs to ensure that accessing elements is done correctly. One cannot blindly pass a C multi-dimensional array to a FORTRAN function or vice versa.

One possible solution to this problem might be to prohibit passing arrays of dimensions greater than one and ensuring identical linearisation in both language contexts. One potential issue with this solution is that it is incompatible with existing code-bases or algorithms that rely on arrays with dimensions greater than one, requiring extensive refactoring to make the algorithm use one-dimensional arrays.

Another solution is to implicitly convert arrays before and after the call a call to a FFI function. This solution, however, entails a large amount of overhead, which may have a negative impact on performance.

The last solution is to pass a flag as part of the parameters of an FFI function indicating whether the inputs are column- or row-major, as is done in [3]. Here the function reads the flag, and accordingly changes how elements within the array are accessed throughout the algorithm. One issue with this solution is that it requires that the programmer implements this "switching" mechanism. Another issue is that array accesses may have an increased overhead associated with it, as the access won't be aligned to the layout of the array in memory. In the SAC/C-FFI we have opted to use the last solution, whereby the programmer switches array accesses.

4.3 The SaC FFIs

Now we present a small example of how the SAC-C/FORTRAN FFI works. We first introduce `sac4c`, the tool provided by the SAC compiler to generate the interface, and describe how it works.

Generating the C FFI.

The SAC compiler tool-chain `sac4c` tool generates a C-FFI, allowing C programs to make use of SAC functions [13, 25, 31]. `sac4c` achieves this by generating C libraries from SAC modules and a C-interface to facilitate the passing of variables to and from the functional implicit-memory context of SAC. This is facilitated by a dedicated C data structure, called `SACarg`, which encapsulates a variable as well as its type and shape, making it accessible within the SAC context [13]. This allows C applications to interface with SAC generated functions which have been optimised and specialised. An underlying feature of the existing FFI is that there are no overheads associated with interfacing variables from one language context to another. This is made possible by using a C pointer to a variable, avoiding the need to copy the variable in memory. Functions are provided to convert C variables to, and from, `SACarg` variables, and to interact with the SAC runtime system (as described in Section 4.1).

In Figure 2, we show some of the basic compilation steps from a SAC module source code to the C library and interface. Assume we are given a SAC module, *Example*, containing a function `baz`. Calling the main compiler tool `sac2c` generates two shared object files: the first is `lib`*Example*`Mod.so`, which provides the generic and specialised formulations of the functions declared in the module source code, the other is `lib`*Example*`Tree.so`, which provides the source code's AST, allowing for inline dependent optimisations to be applied at a later time — this library is only used by the `sac2c` tool when compiling SAC applications. At this stage, the SAC module can be linked to a SAC program, making use of the functions from `lib`*Example*`Tree.so`.

To link to a C program, we call the `sac4c` tool, which generates a C library with the function prototypes from `lib`*Example*`Mod.so`. A header file is also generated providing both the C declarations of the functions available in the SAC module and functions to interact with

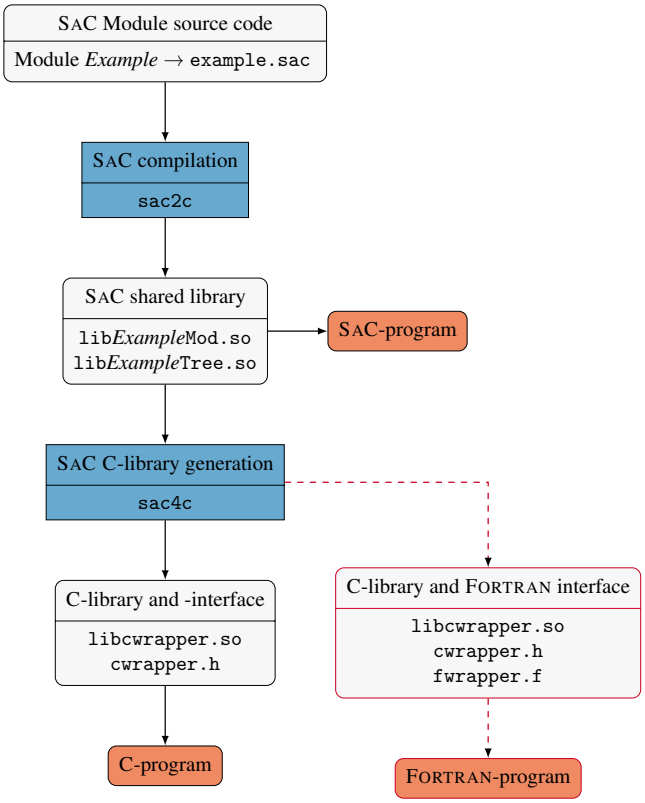

Figure 2: Steps taken to link a SAC module to either a C or FORTRAN program. The generation of the FORTRAN interface is activated through the `sac4c` flag `-fortran`.

the SAC runtime system. At this point, a C program can link to and call a SAC module. Excerpts of the content of the auto-generated `cwrapper.h` are:

```
1  typedef struct SAC_SACARG SACarg;
   typedef struct SAC_SAChive SAChive;
3
   extern void SAC_InitRuntimeSystem (void);
5  extern SAChive *SAC_AllocHive (unsigned int num_bees,
                 int num_schedulers, const int *places,
7                void *thdata);
   extern void SAC_AttachHive (SAChive *hive);
9  extern void SAC_ReleaseHive (SAChive *hive);
   extern SAChive *SAC_DetachHive (void);
11 extern void SAC_ReleaseQueen (void);
   extern void SAC_FreeRuntimeSystem (void);
13

15 extern SACarg *SACARGconvertFromIntPointerVect
                 (int *data, int dim, int *shape);
17
   extern void Example__baz1 (SACarg **ret1, SACarg *arg1);
```

Line 1 defines the C type `SACarg` used by SAC to represent C types and variables. Line 2 defines the type `SAChive` representing a single thread pool. Several functions to interact with the SAC runtime system are given in lines 4–12, including runtime system initialisation, creating a single thread pool, specifying how many threads should be created, attaching the current process to a thread pool, detaching from a thread pool, releasing the master thread of the thread pool, and terminating the runtime system. Line 15 defines a function to convert a C `int` array to the SAC data structure `SACarg`, in which the pointer to the data, as well as its dimensionality and shape, are given as parameters. It is through this, and similar functions, that data can be passed from the C imperative context to SAC's functional context and vice versa. The last line defines the function `baz` from our *Example* SAC module. It is here that we pass in our converted `SACarg` data to the function, which emits the first parameter as the result of the computation.

To use the SAC/C-FFI, a programmer needs to include the `cwrapper.h` header file in the C application. The runtime system initialisation and termination calls can be added at the beginning and end of the application, but are only necessary for the `baz` function — as such they can be placed directly around it. In the case of a single-threaded execution, no further SAC runtime function calls are necessary; only the relevant `SACarg` conversion functions need be called. In the case of multi-threaded execution, after the SAC runtime has been initialised, a thread pool needs to be created and the number of threads specified. The SAC FFI facility supports creating and managing more than one thread pool at a time, facilitated through a single master thread. This allows for multiple SAC functions to be executed in their own thread pool. Creating the thread pool does not however make it available to the process or any SAC function calls. As such, before we can call the `baz` function, we need to attach the thread pool to the current process. After the SAC function has finished executing, and there are no further SAC function calls, the thread pool can be detached from the process. This does not free the thread pool though, meaning we can attach it to it later. To free it, we call a function which releases the thread pool resources. The master thread needs to be freed separately as well. After all SAC related functions have been executed and there are no other resources that need to be released, we can free the SAC runtime system.

Generating the Fortran FFI.

As we are using the underlying support of the existing C-FORTRAN FFI, we need only specify the `-fortran` flag for the `sac4c` tool to generate, in addition to the C library and interface, the FORTRAN interface as well. The SAC-FORTRAN FFI is exposed as a FORTRAN module, which includes the function and subroutine prototypes to bind with the C functions given in the SAC-C FFI. As part of the FORTRAN specification, function and subroutine prototypes are declared within an `interface`-block.

Excerpts of the module file generated by the SAC-FORTRAN FFI are:

```
1  module fwrapper
     use, intrinsic :: iso_c_binding
3    implicit none

5    interface
       subroutine SAC_InitRuntimeSystem ()
7 &        bind (c, name = 'SAC_InitRuntimeSystem')
         import
9        implicit none
       end subroutine SAC_InitRuntimeSystem
11     ...
       type(c_ptr) function SACARGconvertFromIntPointerVect
13 &        (matrix, dims, shape)
  &        bind(c, name = 'SACARGconvertFromIntPointerVect')
15       import
         implicit none
17       type(c_ptr), value, intent(in) :: matrix
         type(c_ptr), value, intent(in) :: shape
19       integer(c_int), value, intent(in) :: dims
       end function SACARGconvertFromIntPointerVect
21     ...
       subroutine sac_baz (num, matrix)
23 &        bind (c, name = 'Example__baz1')
         import
25       implicit none
         type(c_ptr), intent (inout) :: num
27       type(c_ptr), value, intent (in) :: matrix
       end subroutine sac_baz
```

FORTRAN Code

```
integer j, l, lj, li      ! Counters.
integer nn                ! Length of one side of matrix.
real(8) matrix (nn*nn)    ! The one-dimensional matrix.
real(8) alpha, beta, gama ! Temp variables reassigned at
                          ! each iteration.

do j=1,nn-1
  do l=1,nn
    lj = l + (j-1) * nn
    li = lj + nn
    gama = alpha * matrix(lj) + beta * matrix(li)
    matrix(li) = beta * matrix(lj) - alpha * matrix(li)
    matrix(lj) = gama
  enddo
enddo
```

SAC Code

```
int nn;                   // Length of one side of matrix.
int j;                    // Counter.
double matrix[nn,nn];     // The two-dimensional matrix.
double alpha, beta, gama; // Temp variables reassigned at
                          // each iteration.

for (j = 0; j < nn - 1; j++) {
  gama = alpha * matrix[j] + beta * matrix[j+1];
  matrix[j+1] = beta * matrix[j] - alpha * matrix[j+1];
  matrix[j] = gama;
}
```

Figure 3: Here is shown a side-by-side comparison between two pieces of code from the `eigenvv` subroutine that compute over the rows of a matrix. SAC supports row operations, so we need only select the row index in our 2-dimensional `matrix`.

```
  end interface
end module fwrapper
```

Within the FORTRAN interface-block, there are two subroutines and one function declared which bind to the matching C functions given above by the SAC-C FFI — as before, more exist but these have been omitted for brevity. Within each prototype is the `import` statement which "copies" the environment of the `fwrapper` module — meaning that we import the `iso_c_binding` module to within the scope of the prototype. Each prototype then has the `bind` function attribute specified (lines 7, 14, and 23) binding it to the named function. Parameters and return values are matched to the C function signature and named types, such as `c_ptr`. The `fwrapper` module contains no further declarations.

The first subroutine on line 6 has no parameters, so no further declarations are necessary. Line 12 declares a function that returns a `c_ptr` and has one parameter. Attributes of the parameter are declared defining its type, how it is to be passed to the bound C function, and specifying that it is only an input. Line 22 declares a subroutine that links to our example SAC module *Example*, binding to the function `baz`. The parameter attributes are declared in the same way as in the previous case, matching the C function signature. The following code gives a more detailed example of how one would use the SAC-FORTRAN-FFI, in contrast to the one given for the SAC/C-FFI:

```
program example
  use, intrinsic :: iso_c_binding
  use fwrapper

  integer(c_int) f_num
  integer(c_int), parameter :: f_threads = 4
  real(c_double), target :: f_matrix (10, 10)
  type(c_ptr) ::  sac_num, sac_matrix, sac_hive
  !More variable declarations follow
  ...
  !Initialise the SaC runtime system
  call SAC_InitRuntimeSystem ()

  !Convert Fortran/C data types to SaC data structure
  sac_num = SACARGconvertFromIntScalar (f_num)
  sac_matrix = SACARGconvertFromDoublePointer
&               (c_loc(f_matrix), 2, 10, 10)

  !Initialise SaC multi-threaded backend
  SAChive = SAC_AllocHive
&           (f_threads, 0, c_null_ptr, c_null_ptr)
  call SAC_AttachHive (sac_hive)

  !Call SaC module Example function baz ()
  call sac_baz (sac_num, sac_matrix)

  !Shutdown SaC multi-threaded backend
  call SAC_ReleaseQueen ()
  sac_hive = SAC_DetachHive ()

  !Convert back to Fortran/C data types
  f_num = SACARGconvertToScalarArray (sac_num)
  ...
  !Release SaC runtime system resources
  call SAC_FreeRuntimeSystem ()
  ...
end program example
```

The FORTRAN program `example` begins by specifying which modules it uses, in this case the `iso_c_binding` module and our SAC-FORTRAN-FFI module `fwrapper`. From line 5 we declare our variables using C types. We specify that `f_matrix` is an array of size 10×10 and give it the attribute `target`, making it available to be associated with a pointer. Unlike a previous example (in Section 4.2), we do not associate a FORTRAN pointer with `f_matrix`, instead deriving its C pointer directly, using `c_loc`. Critically, data types and structures in FORTRAN are converted to their SAC equivalent; references to these are then passed to the subroutine `sac_baz`. After the subroutine completes, any parameters which are outputs are converted back to FORTRAN data types and manipulated further. We call SAC runtimes functions (lines 12, 20, 22) to set up the environment for the SAC function, which we call on line 25. Thereafter, we convert the SAC data types into FORTRAN types and terminated the runtime system on line 35.

5. IMPLEMENTATION IN SAC

Having implemented the SAC-FORTRAN FFI, we move on to implementing the `eigenvv` SAC application, by rewriting the FORTRAN application source code into SAC code, and ensuring that the both computations produce the same results. Then, we refactor 40 lines out of the 200 lines of the code to take advantage of SAC's capabilities. Rewriting and refactoring the code has two benefits: the refactored SAC code, unlike the original FORTRAN code, directly reflects the two-dimensional nature of the algorithm, and exposes regions of the code that can be executed in parallel, potentially improving application performance.

Regarding the first aspect, the FORTRAN application treats the input matrix as a one-dimensional vector and uses offsets to access the matrix in a two-dimensional fashion. In SAC we rewrite the arrays as two dimensional, which simplifies the code. This leads to the second aspect, whereby the reformulation aids the compiler in identifying data parallelism present in the algorithm. We also expose data parallelism explicitly to the compiler by rewriting several loops as `with`-loops. Recall from our discussion on SAC that the

FORTRAN Code

```
integer j, k, jj, k1, i1   ! Counters.
integer nn                 ! Length of one side of matrix.
real(8) matrix (nn*nn)     ! The one-dimensional matrix.
real(8) alpha, beta, gama, ! Temp variables reassigned at
        delta              ! each iteration.

do k=1,nn-2
  k1 = 1 + k + (k-1)*nn
  do i=k+2,nn
    i1 = i + (k-1)*nn
    do j=1,nn-k
      jj = j * nn
      gama = alpha * matrix (k1+jj) + beta * matrix (i1+jj)
      delta = beta * matrix (k1+jj) - alpha * matrix (i1+jj)
      matrix (i1+jj) = delta
      matrix (k1+jj) = gama
    enddo
  enddo
enddo
```

SAC Code

```
int nn;                 // Length of one side of matrix.
int k, i, iv            // Counters.
double matrix[nn,nn];   // The two-dimensional matrix.
double fmatrix[nn*nn];  // The one-dimensional matrix
double zeta[.,.];       // Two-dimensional temp variable.
double alpha, beta, gama // Temp variables.

for (k = 0; k < nn - 2; k++) {
  for (i = k + 2; i < nn; i++) {
    fmatrix = reshape ([nn*nn], matrix);

    zeta = with { (. <= [iv] <= .) {
           gama = alpha * fmatrix[(k+iv+1)*nn + (k+1)]
                  + beta * fmatrix[(k+iv+1)*nn + i];
           delta = beta * fmatrix[(k+iv+1)*nn + (k+1)]
                  - alpha * fmatrix[(k+iv+1)*nn + i];
           } : [gama,  delta];
     } : genarray ([nn-k-1], [0d, 0d]);

     fmatrix = with {
        ([(k+1) * nn+i] <= [iv] <= [(nn-1)*nn + i]
          step [nn]) : zeta[(iv/nn)-k-1, 1];
     } : modarray (fmatrix);

     fmatrix = with {
        ([(k+1)*nn+(k+1)] <= [iv] <= [(nn-1)*nn+(k+1)]
          step [nn]) : zeta[(iv/nn)-k-1, 0];
     } : modarray (fmatrix);

     matrix = reshape ([nn, nn], fmatrix);
  }
}
```

Figure 4: A side-by-side comparison between two pieces of code from the `trig` subroutine that updates the rows of the matrix in a sweeping motion from left-to-right and top-to-bottom, working over the upper diagonal part of the matrix.

`with`-loop represents a data-parallel operation. Furthermore, some features of SAC, such as array operations, make use of `with`-loops, allowing array operation to be done in parallel. With these two changes to the SAC implementation of the `eigenvv` application, the compiler is able to generate code that executes in parallel.

An example of this simplification can be seen in Figure 3, where the FORTRAN code updates two adjacent columns of the matrix with previously computed variables `alpha` and `beta`. Our SAC rewrite simplifies the two `do`-loop nesting into a single `for`-loop with a sequence of array operation on the array `matrix` that updates the adjacent rows using implicit `with`-loops. Although this rewrite simplifies the code and expresses some data parallelism to the compiler, it is not always the case.

For example, in Figure 4 we see that the SAC rewrite adds syntactic complexity to the code, in order to expose data parallelism to the compiler. The code comes from the `trig` subroutines, which generates a tridiagonal matrix. In particular, the FORTRAN code updates two rows (specified by indexes `k1` and `i1`) in two adjacent columns which move across the array from left to right. The `k` index value is used on each iteration of the second `do`-loop to move the second column by two columns to the right. The same is done on the outermost `do`-loop, but it does not move the first column until the second column has reached the right side of the matrix. This creates a staggered sweeping motion from left to right, bounded by the first column. Each iteration of the inner most `do`-loop takes the previously computed element of the matrix and updates it with the fixed variables `alpha` and `beta`. The second `do`-loop is responsible for the second row down (specified by index `i1`), in essence creating a sweeping motion from the top to the bottom of the matrix. Given this, the outermost `do`-loop moves the first row down, and as its index is used to define the index of the second row, it creates a upper bound for the second `do`-loop. This sweeping movement from left to right, coupled with the sweeping movement from top to bottom, updates the values above the diagonal of the matrix.

Data dependencies disguise the data parallelism for the code in Figure 4. Our initial rewrite, as seen in Figure 3, refactored the innermost `do`-loop into a sequence of array operations on our two-dimensional matrix. This exposed some data parallelism as part of the array operations, but the decreasing iteration space, bounded by the two `for`-loops, reduced the granularity of the data dependencies enough to incur a performance penalty. Our solution to this was to realise that the variables `alpha` and `beta` are computed only as part of the second `for`-loop: they have no data dependence on the matrix, as the columns (in the FORTRAN code) move by two steps to the right. In essence, this means we can expose more data parallelism by precomputing the entire range of `alpha` and `beta`, and updating complete columns (in the SAC code) of the matrix. This increases the granularity of the computation, potentially improving performance, and opens up an opportunity for the compiler to reuse memory.

Expressing this change in a concise way is difficult because our SAC code uses row-major array ordering, meaning that our computation updates columns instead of rows, as is the case with the FORTRAN code. We have to use a different indexing space in order to make efficient jumps from column to column. To do this, we alter the shape of our matrix from two-dimensions to one-dimension using the `reshape` function from the SAC standard library. Within this *reshape* region, the first `with`-loop generates a $2 \times (nn - k - 1)$ element array `zeta` and populates the rows with values computed for the variables `gama` and `delta`. The second and third `with`-loops copy in the values from the first and second rows respectively of the `zeta` array. After performing our computation, we call `reshape` again to the matrix back to two dimensions. The SAC compiler will not generate any additional code for the reshapes, because our

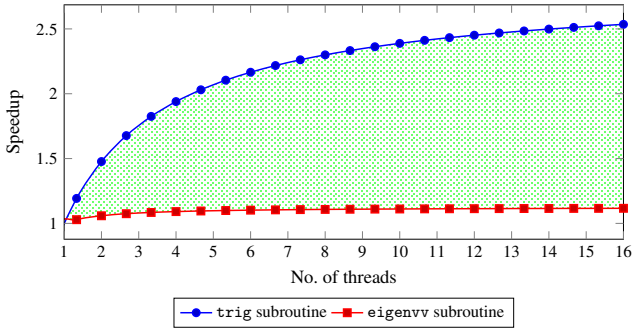

Figure 5: The graph shows the Amdahl Model for the parallelised parts within the `trig` and `eigenvv` subroutines using an input size of 7000×7000. The shaded in region indicates the bounded range for real speedup.

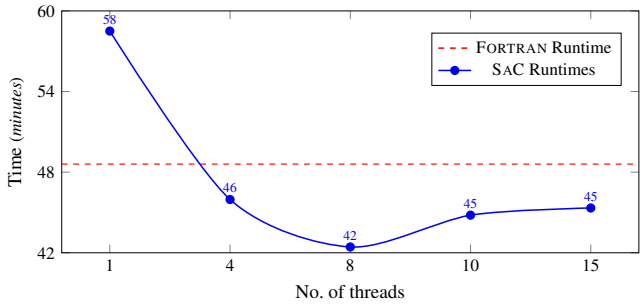

Figure 6: The graph shows the runtime of two versions of the bottleneck subroutine computing over a matrix of about 7000×7000 elements. The first version is the original FORTRAN implementation and the second is the SAC implementation.

one-dimensional array `fmatrix` is only referenced between the two reshapes, and our two-dimensional array `matrix` is not referenced inside of this region. This means that the compiler can avoid creating copies of the data — reference counting in action.

6. EVALUATION

In this section, we present a preliminary set of experiments to observe whether performance of the refactored application is at least as good as the original one. More extensive sets of experiments, as well as investigation of further optimisations, is left for future work.

Setup.

Our experiments included running the FORTRAN and SAC implementations of `eigenvv` on the BGMS data made up of 7000×7000 elements, using a single node of the BGS cluster. Recall that the FORTRAN implementation is single-thread, executing on only one node. The node contains two Intel Xeon CPUs E5-2650v2 2.6 GHz (16 cores in total) with 128 GB of RAM.

For compiling the SAC implementation, we used the SAC compiler `sac2c` version 1.2.beta-BlackForest-82-49e20 and used the following flags: `-O3 -enforceIEEE -target mt_pth -doPRA -Xc '-mcmodel=medium'`. The generated C code is then compiled with GCC compiler `gcc` version 4.8.4. The `-enforceIEEE` flag that we passed to `sac2c` disables "unsafe" optimisations on floating-point arguments (such as those that assume that floating-point addition is associative). This is needed because the convergence criteria of `eigenvv` is sensitive to such unsafe optimisations. The `-target` flag instructs the compiler to generate code for a multicore machine by using POSIX threads to run the code in parallel. Finally, the `-doPRA` flag turns on Polyhedral Reuse Analysis (PRA). This optimisation avoids unnecessary array copies by using a polyhedral model to analyse references within the application of `with`-loops and their results to determine if, at runtime, a result of a `with`-loop can be stored in the memory allocated for one of the existing arrays [16]. In order to do this, one has to prove that a `with`-loop will never read an element of the reused array after it has been written.

To compile the FORTRAN implementation, we used the PGI compiler `pgfortran` version 14.10-0 with flags `-fast -mcmodel=medium`. The `-fast` flag turns on the highest level of optimisations, including vectorisation [29], and `-mcmodel=medium` makes sure that large arrays can be allocated.

Amdahl Model.

To better understand how achievable our goal is in improving the performance of `eigenvv`, we took some initial runtime measurements and used these to generate an Amdahl Model. We did this by measuring the wall-clock time for each region within the FORTRAN implementation that has been rewritten in SAC. There are four such regions, with two in the `trig` subroutine with Figure 4 being one of them and the other two in the `eigenvv` subroutine with Figure 3 being one of those. Since our analysis of the relative percentage speedup between the FORTRAN implementation and the SAC implementation running sequentially showed no significant differences, we felt that our FORTRAN runtime measurements are representative of both implementations. In Figure 5, we have plotted the Amdahl Model of each subroutine individually. We have done this to illustrate that there is a significant difference in the potential parallelism that is possible in each subroutine. With 16 threads, `trig` is able to achieve a theoretical speedup factor of over 2.5, whereas the `eigenvv` subroutine barely achieves a speedup factor of 1.1. What this indicates to us is that, for the current data, our ideal speedup is primarily dependent upon how well the `trig` subroutine can realistically scale with increasing number of threads. This then means that our ideal speedup is the area between the speedup of `trig` and `eigenvv`, shown in Figure 5 as the shaded in region.

SaC vs. Fortran Runtime Experiment.

Using the setup described previously, we made several measurements of the FORTRAN and SAC implementations as they operated on the BGMS data of 7000×7000 elements. All data points are taken from the median of five runs each. In the case of the SAC implementation, we executed it with 1, 4, 8, 10, and finally 15 threads. For the FORTRAN implementation, we ran it as a single thread, as the code is sequential.

A graph of our runtime results can be seen in Figure 6. As the FORTRAN runtime does not vary with the number of threads, we have drawn a straight line for its runtime. For the SAC runtime numbers, it is clear that at one thread, we have a performance degradation which drastically improves as we add more threads. This is a well-known behaviour of multi-threaded SAC programs, and is explained by scheduling overheads within the runtime system. As we increase the number of threads, we begin to see some speedup, with the best runtime occurring with eight threads. Thereafter the performance degrades somewhat and plateaus with additional threads. It is not clear to us why this is and requires further investigation, but it is possible that this is due to the inherent data dependencies within the application. In general, the SAC implementation achieves a modest speedup of approximately 12.5% when running the code

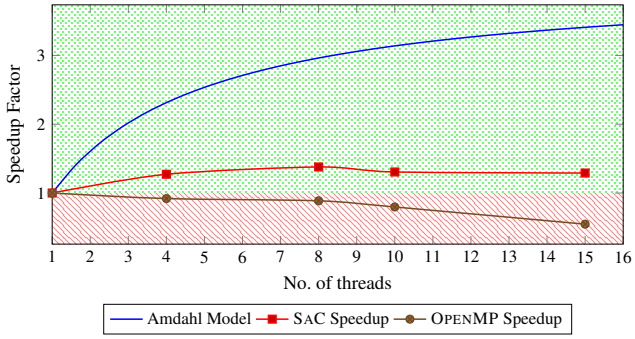

Figure 7: The graph shows the Amdahl Model of the parallelised regions within `trig` and `eigenvv` subroutines, compared to the actual speedup of both the SAC application and the FORTRAN implementation using OPENMP directives.

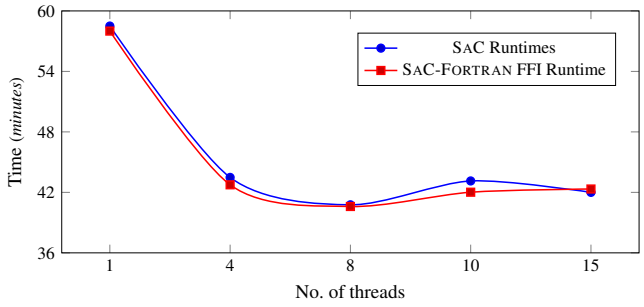

Figure 8: The graph demonstrates that there is a minimal amount of difference in runtime between the SAC implementation and the SAC/FORTRAN-FFI implementation, meaning that there are no measurable overheads associated with using the FFI.

on eight cores.

Comparative Experiment with OpenMP.

In order to see how effective the SAC auto-parallelisation is, we decided to do a comparative experiment with another auto-parallelising framework: OPENMP. To do this, we took the FORTRAN implementation and added OPENMP `PARALLEL DO` directives around the same regions of code that SAC is able to generate parallel code for. We added further directives, such as `FIRSTPRIVATE` and `SHARED`, to indicate which variables are private and which are shared, trying to give the compiler as much information as possible. Furthermore, in order to have the OPENMP runtime system behave similarly to SAC, we used the `GUIDED` scheduler which at each invocation of the `do`-loops distributes the computation as equally as possible among the threads. Additionally, we added a `IF`-clause directive, to prevent parallel execution for chunk sizes less then 250 — this is the same limit set for SAC parallel executing regions. As before, we used the same setup as described previously and took our measurements using 1, 4, 8, 10, and 15 threads.

The runtime results are given as speedup values and are shown in Figure 7. The graph has three plots on it, the uppermost is the complete Amdahl Model for the application (`trig` and `eigenvv` ideal speedup together), the next is the real speedup of the SAC implementation, and the bottom most plot is the real speedup of the OPENMP implementation. As can be seen, the SAC implementation achieves a modest speedup that is nonetheless still far off from the ideal speedup given by the Amdahl Model. In contrast, the OPENMP implementation achieves *no actual speedup* at all, instead we see a continuous degradation in performance as the number of threads is increased. A possible reason for this might be the overheads associated with the OPENMP fork/join model. This experiment shows then that the auto-parallelisation within SAC is more effective then OPENMP.

SaC/Fortran-FFI Overhead Experiment.

For this experiment, we wanted to demonstrate that the SAC/-FORTRAN-FFI does not incur any overheads in comparison to the SAC implementation. To do this, we repeated our SAC runtime experiment from above for both the SAC implementation and an implementation that called the SAC codes from the FORTRAN context via our FFI. As before, we took our measurements by computing over the BGMS using 1, 4, 8, 10 and 15 threads. The runtime results are shown in Figure 8. As can be seen, there is little difference in the overall runtime between both the non-FFI and FFI implementation. Furthermore, they both display the same performance characteristics, with the best speedup achieved with eight threads.

7. RELATED WORK

Most programming languages provide an FFI in one form or another. Typically, this is an FFI to C, as C is available on a large number of platforms, has many advanced libraries, and usually offers a straightforward interface. When a structure within the language is similar to one in C, then interfacing is very simple. For example, consider how mixed-language programming works between C and C++. As part of the C++ standard, one can interface with and expose functions between C and C++ by declaring C++ functions with `extern "C"` operator.

When semantics and language concepts are more distant, FFIs are more involved and require challenging design decisions [8, 19, 24, 28], similarly to what we discussed in Section 4.1.

A number of works propose systems that allow tighter symbiosis between the host and embedded languages. For example, in [4], the authors propose a system to encode any C type in ML and provide functions to inspect and manipulate data structures of C directly from ML. This approach is possible because the type system of ML is not weaker than the type system of C.

A system to identify errors that are caused by using PYTHON/C FFIs and that relate to reference counting errors is proposed by [23]. The authors propose a tool called Pungi — a system that statically checks whether a PYTHON objects' reference count is adjusted correctly in PYTHON/C interface code.

In [34] the authors propose a system to reason about safety and compatibility of effects in different languages connected via FFI. Specifically, they are interested in the case when one language has a notion of effects that are not directly representable in other languages. For example, if an ML function throws an exception, it is not clear how to handle this in C.

In [1], the authors use the notion of universes in dependently typed languages to represent types of programming language *A* within programming language *B*.

Another interesting approach to avoid FFI-related problems is to compile both languages to a common representation, such as Microsoft CLR [17]. This approach solves the FFI problem by requiring a language to be able to compile down to a CLR.

An alternative approach, used by SWIG [2], makes it possible to generate automatically most of the glue code that is required for two languages connected via a FFI.

Another approach is proposed in [22], which instead of creating

an FFI to link legacy code with specialised code, seeks to modify the compiled binary by auto-parallelising affine loops. The authors solution uses a "binary rewriter" that takes a x86 binary as input and decomposes into the Low-Level Virtual Machine (LLVM) compiler intermediary representation (IR). The process of decomposing is done completely without any symbolic knowledge of the application, and involves identifying loops and nestings. Once identified, the binary rewriter generates appropriate code to parallelise the loops, and outputs the code as LLVM IR. It is from here that the new binary can be compiled, using LLVM compiler. The authors show that they are able to achieve speedups across a range of benchmarks from both the Polyhedral Benchmark suite and *Stream* from the HPCC Benchmark suite.

8. CONCLUSION AND FUTURE WORK

This paper studies a use case of a functional programming language that resolves a performance bottleneck in a commercial, high-performance, legacy FORTRAN application; we achieve a modest improvement in the performance of the bottleneck code. The study demonstrates a possible solution to resolving the open problem of bringing functional programming into the high-performance domain, showing that:

1. a functional/imperative symbiosis in the context of high-performance computing is possible, and is an effective way of utilising the strengths of the functional setting to bring about performance improvements; and

2. that the key to success is a clear migration path for integrating functional languages into legacy code.

A key enabler for this demonstration is the ability to mix functional and imperative languages together through a foreign-function interface (FFI) that is lightweight and straight-forward to use. The functional programming language Single-Assignment C (SAC) provides such an FFI to expose SAC code to C applications. We implement an extension to the SAC FFI to allow the integration of SAC code within FORTRAN applications. This is made possible by using the FORTRAN/C interoperability module `iso_c_binding`, which provides a robust framework for interfacing C and FORTRAN code together.

The legacy FORTRAN application at the centre of this study is the BGS Geomagnetic Field Modelling System (BGMS), which takes typically 12 hours to execute a single cycle on a cluster using approximately 100 CPU cores. The BGMS has a performance bottleneck which takes approximately 50% of the overall runtime as it executes as a single thread on a single node. It is this bottleneck code which is at the core of our study. We rewrite it in SAC to take advantage of compiler technologies to auto-parallelise the code.

We demonstrate through several experiments that we can achieve a speedup of 12.5% for the bottleneck code. We compare this to an implementation that uses OPENMP directives and show that the OPENMP implementation is unable to achieve any speedup at all. Furthermore, we show through a further experiment that the SAC/FORTRAN-FFI has no measurable overheads associated with it.

Though we only achieve a relative speedup of 12.5% with the bottleneck code, which translate to an overall speedup for the BGMS application of approximately 6%, this is nonetheless a good result. With legacy applications like the BGMS, which have been around for decades, a considerable amount of time and effort has been invested into improving their performance while maintaining semantic robustness. Thus achieving any further speedup, especially when the associated effort is comparatively low, is of significant value.

Much of this study relies on using FFIs and we have tried to answer and resolve some of the issues associated with creating an FFI, but there are still a number of questions open to further investigation. For example, we could investigate how to provide more guarantees regarding type safety and deadlock/data-race freedom. Furthermore, we could investigate how stable an FFI is when a language *A* includes a function in language *B* that calls a function in language *A*, which could be arbitrarily nested. And finally, we could try to identify properties that a language should have in order to become FFI-compatible with another language. If those properties could be clearly defined, then we can envision systems with multi-language FFIs. Also, we can consider allowing mixing various languages in the same source file, similarly to the way it is done in GCC by using `extern "<language>"` directive, where `<language>` can be one of the languages supported by GCC, like for example "C" or "C++".

Given that we have not hit the theoretical limit, further work could include analysing the generated code to see whether the compilation technology itself can be improved or if we have missed some of the optimisation opportunities. Additionally we could look at variants of the algorithm used within the bottleneck code that are more suitable for parallel execution such as those proposed in [11].

We believe that this paper demonstrates that the proposed approach of rewriting performance bottlenecks in functional languages and integrating them back into the legacy code via an FFI is feasible and can be successfully applied outside the SAC/C/FORTRAN world.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive feedback. Additionally, we would like to thank Robert Bernecky of Snake Island Research Inc. for his input and support. This work was supported in part by grant EP/L00058X/1 from the UK Engineering and Physical Sciences Research Council (EPSRC) and by the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems through grant EP/L016834/1. Further support was provided by the British Geological Survey in association with the Natural Environment Research Council (NERC).

10. REFERENCES

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1*, pages 1–20. Kluwer, B.V., 2003.

[2] D. M. Beazley et al. SWig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[3] L. S. Blackford et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[4] M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *Electr. Notes Theor. Comput. Sci.*, 59(1):36–52, 2001.

[5] British Geological Survey (BGS). BGS Global Geomagnetic Model, 2015.

[6] A. Chulliat et al. The us/uk world magnetic model for 2015-2020. Technical Report 10.7289/V5TB14V7, National Geophysical Data Center, NOAA, Mar. 2015.

[7] ESA Swarm mission: An overview, 2015.

[8] S. Finne et al. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN*, ICFP '99, pages 114–125, New York, NY, USA, 1999. ACM.

[9] Free Software Foundation, Inc. *The GNU Fortran Compiler Manual*, 2015.

[10] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN*, PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM.

[11] G. H. Golub and C. F. Van Loan. *Matrix Computations (4rd Ed.)*. The Johns Hopkins University Press, Baltimore, MD, USA, 2013.

[12] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Language SAC*. PhD thesis, University of Kiel, Germany, 2001.

[13] C. Grelck. *CEFP 2011*, chapter Single Assignment C (SAC) High Productivity Meets High Performance, pages 207–278. Springer, 2012.

[14] C. Grelck and S.-B. Scholz. Classes and objects as basis for I/O in sac. In *Proceedings of the 7th International Workshop on the Implementation of Functional Languages*, volume 95 of *IFL '95*, pages 30–44, 1995.

[15] C. Grelck and S.-B. Scholz. SAC – A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[16] J. Guo, R. Bernecky, J. Thiyagalingam, and S.-B. Scholz. Polyhedral methods for improving parallel update-in-place. In S. Rajopadhye and S. Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, Jan. 2014.

[17] J. Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, Feb. 2003.

[18] K. Hammond and G. Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 1999.

[19] L. Huelsbergen. A Portable C Interface for Standard ML of New Jersey. http://www.smlnj.org//doc/SMLNJ-C/smlnj-c.ps, 1996.

[20] ISO. Information technology — Further interoperability of Fortran with C. ISO ISO/IEC TS 29113:2012, International Organization for Standardization, Geneva, Switzerland, 2012.

[21] G. Kindlmann et al. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics*, Oct. 2015.

[22] A. Kotha et al. Automatic parallelization in a binary rewriter. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 547–557, Dec. 2010.

[23] S. Li and G. Tan. Finding reference-counting errors in python/c programs with affine analysis. In R. Jones, editor, *ECOOP 2014*, volume 8586 of *Lecture Notes in Computer Science*, pages 80–104. Springer, 2014.

[24] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[25] N. Marcussen-Wulff and S.-B. Scholz. On interfacing sac modules with c programs. In *Proceedings of the 12th International Workshop on the Implementation of Functional Languages (IFL 2000)*, pages 381–386, 2000.

[26] S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly Media, 2013.

[27] S. Marlow et al. Seq no more: Better strategies for parallel Haskell. In *Haskell '10: Proceedings of the Third ACM SIGPLAN Symposium on Haskell*. ACM, 2010.

[28] S. Marlow, S. P. Jones, and W. Thaller. Extending the haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 22–32, New York, NY, USA, 2004. ACM.

[29] NVIDIA Corporation. *PGI Fortran User's Guide*, 2015.

[30] M. Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[31] P. Rauzy. Implicit parallelization of code called from an external and already parallelized environment: from design to implementation. Technical report, Uni. of Amsterdam, 2011.

[32] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, 2015.

[33] E. Thébault et al. International geomagnetic reference field: the 12th generation. *Earth, Planets and Space*, 67(1), 2015.

[34] V. Trifonov and Z. Shao. Safe and principled language interoperation. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 128–146. 1999.

[35] P. W. Trinder et al. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, Jan. 1998.