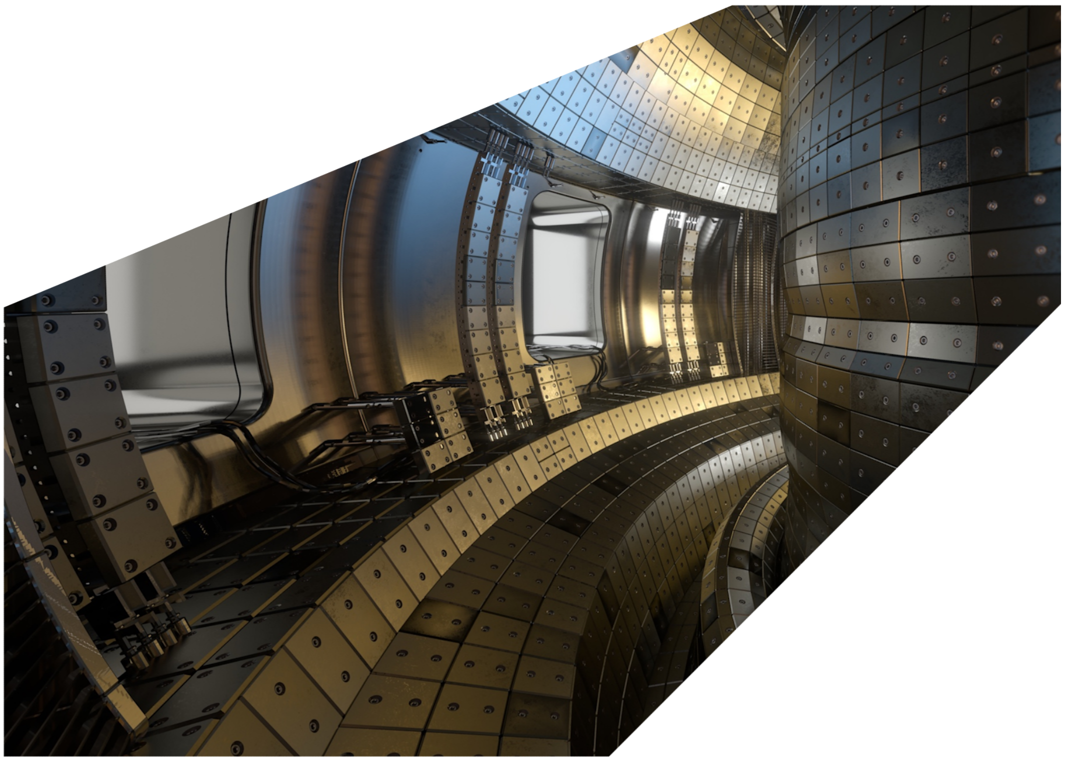
UK Atomic Energy Authority

ExCALIBUR

Survey of Domain Specific Languages

M3.2.2

Abstract
The report describes work for ExCALIBUR project NEPTUNE at Milestone 3.2.2. The work surveys usage of DSL in plasma and neutral simulation, indicating the advantages and costs for users and developers. This was achieved by holding two workshops on DSLs, both organised by UKAEA, held online on 8 April 2021 and 23 July 2021 respectively, attended by members of the NEPTUNE community and in July, also STFC experts. The conclusions presented at the second workshop were, to recognise the need for at least two DSLs, the higher level (1) for plasma physics experts, and the lower level (2) for developers with HPC skills. Two options were identified for each level, namely Python (provisionally preferred) and Julia for (1), and SYCL (provisionally preferred) and Kokkos for (2). The planned advance of NEPTUNE, via a sequence of proxyapps, allows for exploration of the options in separate proxyapp developments, in order to allow clear preferences to emerge. It was also agreed as a general principle that at any time, the NEPTUNE development should allow two options for any critical software, to provide redundancy and flexibility.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0041
	Issue:	1.00
	Date:	July 30,2021
Project Name: ExCALIBUR Fusion Modelling System		

	Name and Department	Signature	Date
Prepared By:	Ed Threlfall Wayne Arter Joseph Parker Will Saunders BD	N/A N/A N/A N/A	July 30,2021 July 30,2021 July 30,2021 July 30,2021
Reviewed By:	Rob Akers Advanced Computing Dept. Manager		July 30,2021
Approved By:	Rob Akers Advanced Computing Dept. Manager		July 30,2021

1 Introduction

This report summarises the outcome of two workshops on Domain-specific languages (DSLs) organised by UKAEA, held online on 8 April 2021 and 23 July 2021. The aim of the workshops was twofold, namely to improve understanding of what DSLs have to offer, and to identify the best way DSL(s) could be employed by NEPTUNE. In the latter case, it is important for DSL(s) not only to provide the ExCALIBUR pillar of 'Separation of Concerns', but also to furnish attractive interfaces to users to promote growth of a community around the software.

It is worth beginning with a certain amount of historical detail concerning DSLs. The term DSL is not in fact recognised by the standard reference for software engineering [1], and Hewitt [2] notes only 'domain' as a set-theoretic term primarily used to denote an API to a library, where 'domain language' implies a naming convention for the interface subroutines. However, the paper by Johanson et al [3] in the collection of Carver et al cites a book "Domain-specific languages" [4] published in 2010 by Fowler. Fowler's introductory description [4] allows for a DSL to be little more than a thin wrapper, although a weightier construct conceptually closer to a framework is typically implied, especially in more recent DSL works.

Project NEPTUNE is fortunate in an absence of historical constraints that mean the user interface may be a primary consideration. At both workshops it was asked for DSL(s) that make the software attractive to three broad classes of user:

1. Engineer or physicist using the code as a 'black box', eg. a design engineer or a plasma physics experimentalist.
2. High-level programmer using eg. Python or Julia.
3. Writer of new problem-specific code in eg. C++.

There was the reminder that NEPTUNE code needs to be modifiable and extensible to stand the test of 30 years' future use, which implies either the DSL be supported over such a timescale, or that it generate code capable of modification and extension in an enduring language such as C++. The complexity of the governing equations, particularly as gyro-averaged kinetic models were envisaged, was also highlighted.

The division of users into classes resonates with the layering concept introduced by ref [3], which is recorded as being based upon extensive user consultation. Users [3] also requested DSL learning/training materials, although these were not considered in either workshop.

Edited discussions from the two workshops form Annex Section A and Annex Section B below. The presentation of the minutes below in abbreviated form without detailed references preserves much of the excitement of the debate (in which nearly all the participants actively engaged) without unduly detracting from understanding. Whereas the first workshop was confined to members of the NEPTUNE community, the second workshop benefitted greatly from the contributions of a team from STFC. Although much of the discussion was very interesting, the conclusions are the main item of significance as as far as the NEPTUNE project is concerned, and these appear in the next Section 2.

2 Conclusions

The workshops imply that there are *two* separate DSLs required, for the separate *domains* of (1) tokamak (edge) plasma physics and (2) HPC at the Exascale. The first DSL is primarily a user interface and the second more for developers. (An older pairing with which many would be familiar is the employment of shell scripts for (1) and CRAY FORTRAN for (2), with the HPC aspect met by use of special directives for the CRAY architecture inserted into the FORTRAN, whereas a more recena, more machine-indpendent pairing is Python and C++ combined with Kokkos [5].) The layer represented by the 'black box' user could be met by further layering in domain (1), hence need not be an immediate concern, although the need for such a layer should be borne in mind. UKAEA design engineers presntly prefer software capable of integration into ANSYS workbench, whereas experimentalists might want to use initial conditions taken from a database, specified by a given shot number at a specified time.

Owing to continuing developments in HPC architectures and the high level languages for their exploitation, it is concluded that it would be wise to keep options open for both (1) and (2). There is anyway a general principle to observe in any project, namely that there should always be more than one option, in order to provide redundancy and in the current context also prevent stagnation, should a radically new, more attractive package emerge unexpectedly.

The preferred combination of options, particularly as far as finite elements are concerned is Python plus C++/SYCL [6] for (1) and (2) respectively. The value of Python at level (1) is demonstrated by both Unified Form Language, UFL [7, § 2.2.1] and the work of Dedalus Project [8, 9].

Since Python is object-oriented, its use in a 'black box' should be straightforward, and its ability to coordinate use of other packages at a high level, eg. to couple or 'glue' codes together, will also be invaluable. UFL allows for use of 'escape hatches' that will be essential given the complexity of plasma modelling envisaged. It should be possible, following Dedalus Project to implement a thin software layer over UFL that allows for user specification of the strong form of an equation, both directly and also where appropriate in Lagrangian or Hamiltonian form. The ultimate interface could be specified as a subset of LaTeX, widely known because of its use in the production of scientific papers, giving attractive options for implementing models directly from the literature, with LaTeXcompatibility an important cross-check on the model equations and indeed boundary conditions, perhaps also aiding the automatic production of documentation. The DSL could also be extended with safety features that eg. monitor conservation properties and/or permit use of Method of Manufactured Solutions (MMS) [10], plus special features for Uncertainty Quantification (UQ), such as the automatic generation of adjoint systems.

C++/SYCL could be used within the escape hatches, see Figure 1. SYCL has been developed for performance portability across heterogeneous HPC architectures. It is opensource and conforms to a recent C++ standard (C++17) standard, both properties which should be good for long term maintainability. The range of applications for which it has been shown to produce performance-portable code is increasing [11]. Features of oneAPI [12] which is ultimately based on SYCL, allow an approach to the goal that the programmer need only write code for a single cpu with all issues to due with parallel execution to be addressed by the DSL, as in the case of `PSyclone` [13].

There are the drawbacks that SYCL is not as fully developed or as widely used as Kokkos however, and there might be questions about its long term viability if Intel were to withdraw the corporate

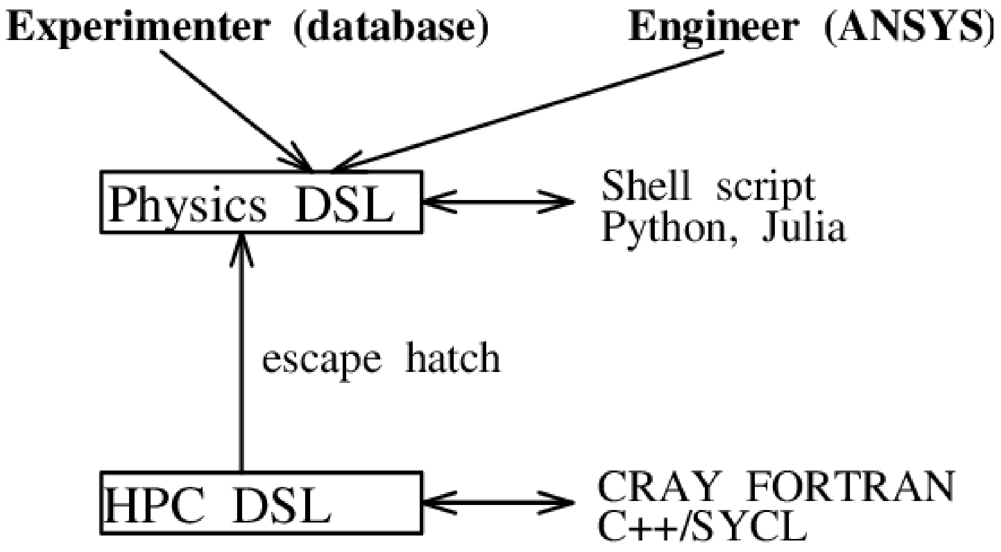


Figure 1: The two DSLs and their inter-relationships with each other and potential users.

support demonstrated by DPC++ (note the freely available book [14]) and OneAPI. At the high level (1), the Julia language is gaining in popularity for scientific work, for which it was designed, eg. Jupyter notebooks now apparently widely used for teaching mathematics, now also support a Julia interface [15] in addition to Python, thus it seems unwise to rule out future use of Julia for NEPTUNE.

To continue to keep options for both (1) and (2) open for a while longer, the structure of the NEPTUNE project which proceeds by the development of a series of proxyapps should be exploited. Presently, there appear to be few if any DSLs for particle codes, and UFL itself is focussed on finite elements, thus it would seem possible, even desirable to produce at least one proxyappfor particle work using Julia at (1). Although SYCL should be preferred at (2), the use of Kokkos probably indirectly as the underpinning of numerical libraries, should be allowed, indeed may be essential, for the time being.

Acknowledgement

The support of the UK Meteorological Office and Strategic Priorities Fund is acknowledged.

References

[1] I. Sommerville. *Software Engineering. 5th Edition (10th Edition, 2017)*. Addison-Wesley, 1997.

[2] E. Hewitt. *Semantic Software Design: A New Theory and Practical Guide for Modern Architects*. O'Reilly Media, 2019.

[3] A.N. Johanson, W. Hasselbring, A. Oschlies, and B. Worm. Evaluating hierarchical domain-specific languages for computational science: Applying the Sprat approach to a marine ecosystem model. In J.C. Carver, N.P. Chue Hong, and G.K. Thiruvathukal, editors, *Software Engineering for Science*, pages 175–199. 2017.

[4] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[5] Kokkos website. `https://github.com/kokkos`, 2020. Accessed: November 2020.

[6] Khronos Group SYCL website. `https://www.khronos.org/sycl/`, 2020. Accessed: November 2020.

[7] E. Threlfall and W. Arter. Survey of code generators and their suitability for NEPTUNE. Technical Report CD/EXCALIBUR-FMS/0039-M3.2.1, UKAEA, 2021.

[8] K.J. Burns, G.M. Vasil, J.S. Oishi, D. Lecoanet, and B.P. Brown. Dedalus: A flexible framework for numerical simulations with spectral methods. *arXiv preprint arXiv:1905.10388*, 2019.

[9] A flexible framework for spectrally solving differential equations. `https://dedalus-project.org`, 2021. Accessed: July 2021.

[10] B.D. Dudson, J. Madsen, J. Omotani, P. Hill, L. Easy, and M. Løiten. Verification of BOUT++ by the method of manufactured solutions. *Physics of Plasmas*, 23(6):062303, 2016.

[11] T. Deakin and S. McIntosh-Smith. Evaluating the performance of HPC-style SYCL applications. In *Proceedings of the International Workshop on OpenCL*, pages 1–11, 2020.

[12] Optimize Application Performance for the Latest Intel Hardware. `https://software.intel.com/content/www/us/en/develop/tools/oneapi.html`, 2021. Accessed: July 2021.

[13] S.V. Adams, R.W. Ford, M. Hambley, J.M. Hobson, I. Kavčič, C.M. Maynard, T. Melvin, E.H. Müller, S. Mullerworth, A.R. Porter, M. Rezny, B.J. Shipway, and R. Wong. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *Journal of Parallel and Distributed Computing*, 132:383–396, 2019.

[14] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[15] IJulia is a Julia-language backend combined with Jupyter. `https://github.com/JuliaLang/IJulia.jl`, 2021. Accessed: July 2021.

A Edited Minutes of DSL Workshop: 8 April 2021

A.1 Bridging the complexity gap in exascale simulation software development through high-level DSLs, Gihan Mudalige (Warwick)

This was an overview of many years' work by GM. A motivation for the work was given: the need for parallelizable code capable of working on a diverse hardware landscape: a large 'zoo' of accelerator types and many different programming methods (OpenMP, SIMD, CUDA, ROCm / HIP, MPI, PGAS ...). Open standards attempting to keep up with the range of hardware and complicated by a lack of overall consensus and companies' vested interests in optimizing code for their own hardware; also the issue of large legacy codes (Fortran was mentioned).

Raising the level of abstraction to solve the problem of a varied hardware landscape: GM explained that a classical compiler has two basic themes: analysis (syntax, semantics, ... , polyhedra) and synthesis (parallelization, tiling, vectorization) (WA asked for clarification of 'polyhedra' in this context: it means things like cache-blocking and tiling). The problem is really that for a general program the compiler cannot easily optimize in terms of layouts and memory spaces (eg. if running a computation on an unstructured mesh). The proposed solution is to raise the level of abstraction from that of a general programming language to eg. a communication skeleton (`OP2` / `OPS`) and further to a specific numerical method eg. FIREDRAKE. With a narrower problem ambit (eg. just FEM), there is scope to re-use a known set of optimizations (or sets, given that there are multiple target platforms). This would work as a domain-specific API (the 'contract' with the user) being embedded in eg. Python or C++. A quote from Mike Giles was cited: `OP2` / `OPS` 'straitjacket' the user and prevent them from writing bad code. So one has a scheme in which a problem is declared by the user and a set of automated routines produce an optimized implementation. Examples of handling unstructured mesh code (which looked to be finite-difference based) were shown; the problem here was sorting out data races if multiple edges tried to update the same node.

A nice application structure diagram was shown and it was made clear that eg. `OP2` acts as a parser: it is a source-to-source translator and it outputs human-readable code, which then goes into a general compiler and can be used with general debugging / profiling tools (g++, gdb, Allinea MAP). The DSL layer here handles automatic parallelization, load-balancing, checkpointing, and runtime (JIT) compilation.

The talk necessarily accelerated here due to time constraints; some `SYCL` examples were shown (a useful citation is a 2021 publication by GM, Jarvis, Powell, Owenson, Reguly `https://warwick.ac.uk/fac/sci/dcs/people/gihan_mudalige/op2-mgcfd.pdf`).

Mention was given to the ASIMOV project: re-engineering existing codes with `OP2`; eg. one which is 50k lines of Fortran with over 300 parallel loops.

It was emphasized that getting the correct abstraction (ie. higher-level description) is the main thing and that this has more mileage in it than the particular implementation using the technology of the moment (this paraphrases a quote from Alfred Aho and Jeffrey Ullman).

WA asked how to minimize the workload of a future re-engineering exercise such as in ASIMOV; GM replied that NEPTUNE is in a good position as it starts code from scratch - just get the abstractions correct.

Patrick Farrell asked whether the DSL acts at compile or runtime. GM replied that historically it has been done at compile time due to eg. not having the compiler available on the HPC compute nodes, but some stuff can be done at runtime eg. FIREDRAKE does loop nesting and tiling.

A.2 What makes a good DSL, Will Saunders (UKAEA)

WS advised he would provide a more high-level discussion, starting with some (ubiquitous!) examples of DSLs eg. LaTeX, SQL, HTML, APIs. Desirable properties are:

- Ease of use.
- Communication: allows accurate problem description and allows enforcing of explicit standards from third parties eg. a publisher's LaTeXconventions.
- Abstractions to give separation of concerns (eg. LaTeXusers are agnostic as to *how* exactly their PDF is generated).

Two types of DSLs were considered: 1) External, implemented by a specific interpreter or compiler (eg. LaTeX, SQL, `Make`), which are flexible but involve the hard work of writing a compiler; 2) Embedded, which extend an existing host language (ie. an API) (eg. `SYCL`, `UFL` - Unified Form Language). The latter allows the use of the host ecosystem, but restricts the DSL to use the lexicon of the host language (eg. Python does not support overloading the assignment operator).

WS presented three characteristics of a 'good' DSL:

- Provides the correct abstraction to describe domain tasks (concurring with GM's preceding talk).
- Ease of use ie. intuitive for domain user.
- Composable, as DSLs are rarely used in isolation.

WS finished by discussing the question of what we want from a DSL:

- This is really an open question for all levels of our (prospective) user community.
- Separation of concerns - a hierarchy.
- Performance portability over likely HPC targets.
- Offering interoperability between components ie. acting as a gluing language.

A.3 Discussion of NEKTAR++ from a DSL standpoint (my title), Spencer Sherwin

SS explained that NEKTAR++ did not initially use DSLs. He gave then an overview of the structure of the code in the latest version (refined in the light of knowledge gained writing earlier spectral / hp codes). Currently the developers are pushing a top-level Python interface (also, I'd add that the xml session file is a DSL of sorts).

In a discussion between SS and WA it became clear that the fluids community had relatively little need for a complex DSL because fluid equations are fairly standardized things (in contrast to the wide range of models used in fusion). WA emphasized that the new terms added to fusion models (eg. sources) had the potential to cause a wide range of numerical issues, and so it was clear that these might need a relatively deep integration with the source code.

A.4 `UFL` / FIREDRAKE (my title), Patrick Farrell

The question of what happens if the DSL is too restrictive to add a certain piece of new physics was raised by SS; PF explained that FIREDRAKE circumvents this issue by allowing pieces of code from other languages to be included (via 'escape hatches'), so there can be C++ or Python 'bolt-ons'. PF showed then some slides taken from a FEM theory course he teaches showing the use of `UFL` in FIREDRAKE (one very nice feature is a simple API to generate regular meshes - not present in NEKTAR++). This showed how easy it is to specify weak-form PDEs eg. `G = inner(grad(u), grad(v))*dx - inner(f,v)*dx` then `Solve(G==0,u,bc)`. He then showed a more complex linear elasticity example and explained that these toy problems could nevertheless be scaled to billions of degrees of freedom of ARCHER. WA asked to see an example of some bolt-on code (PF prepared some - see A.6 below).

A.5 DSLs in BOUT++ (my title), Ben Dudson

BD's experience with coming to existing code and finding discrepancies between the code and the documentation, as well as fusion physics' lack of completely-specified models, led to the aim to make it easy to add new physics to BOUT++, and easy to read what equations are being solved. He explained the code structure ie. method-of-lines time integration with all the physics in a module that computes the time-derivatives, and explained that there are two DSLs used in BOUT++:

- The physics equations are written in C++ eg. `ddt(n) = -vE_Grad(n,phi)+Div_par(Jpar)+2*DDZ(n) / R_c`.
- An input configuration file format. This evolved from a simple configuration file (INI format), and now it can eg. parse complex mathematical formulae, and is Turing-complete. This evolution was driven by the need for increasingly complex configurations, in particular testing with MMS. In hindsight it might have been better to adopt a standardised interpreted input language, rather than evolve a unique one. WA asked whether the next bit of code shown was C++ but BD replied it is an external DSL that is interpreted at runtime by an interpreter inside BOUT++.

BD explained how the code had been made performant and cited work by Joseph Parker (2018) vectorizing the kernel inner loops (subsequently, the bottleneck became the elliptic inversion). Another problem overcome was that of too many small loops not parallelizing efficiently on GPUs (need more work per unit of loaded data to get the efficiency). Showed example of merging loops to form a single outer loop. This became hard to debug when using techniques such as C++ templates and code generation, and adding debugging framework code wrecked the performance. To address this, an idea was borrowed from `SYCL` - unsafe but lightweight wrappers and doing the runtime checks outside the loops. This enables efficient code which retains readability, and can be more easily debugged. There was a brief discussion about the need for code tuning and manual vectorization, as compilers cannot always do this well, and interaction with threading only complicates things.

A.6 General discussion

PF showed FIREDRAKE with bolt-on code snippets eg. example of converting non-periodic mesh to periodic in loopy syntax (https://documen.tician.de/loopy/) and the other as a normal C function to exploit tensor product structure. The kernels (either plain C or loopy) are inputs along with sets, maps and access descriptors to PyOP2. PyOP2 is a large Python code integral to FIREDRAKE which generates the C source code for a shared library which is written to disk and then compiled with a C compiler of choice (usually GCC).

WA raised a couple of points:

- From BD talk: it was clear that not all plasma physicists will know about FEM.
- BOUT++ uses method of lines, and also elliptic solvers: what if we needed to solve coupled elliptic problems? BD explained that such things could be transformed into something that can be solved.

SS raised issue of whether a DSL could encourage good practice - clearly with a DSL the added flexibility means more scope to get nonsense results out (though, presumably, VVUQ techniques would flag up bad calculations). WA agreed that is a big question ... SS mentioned that in finite difference, the only adjustable parameter is the global refinement (whereas, in finite element, one can locally refine the mesh, or do p-refinement; WA mentioned it had to be done dynamically as shocks can form during simulations).

SS asked whether the user could be warned if they had introduced a term that was likely to cause the simulation to fail; WA responded that the equations for fusion tended not to be that bad in this regard (second derivatives, collisions and transport) and that the difficulty really came from the sheer number of different terms and possible species. PF asked what a warning (or straitjacketing) system would mean in practice given that it might reduce overall freedom (eg. clearly LaTeXallows the user to write mathematically incorrect equations). SS replied that his intent was to help the user understand how simulation results are affected by the various solver options. PF put it that it should be made easy for the user to do things they 'should' be doing (ie. use the interface to 'nudge' users in the right direction). BD agreed, saying that this was the thinking behind BOUT++. PF and SS agreed that error checking by means of the method of manufactured solutions was useful. PF then said his concern with `UFL` was that it is designed for FEM, not plasmas: he plans to talk to BD about what equations are needed for NEPTUNE and also the wider cross-cutting ExCALIBUR themes. Specifically, BD asked whether `UFL` can handle 5-D and 6-D phase spaces (ie. including velocity space), to which PF replied that `UFL` can handle this but the solvers (FIREDRAKE) currently cannot.

WA said it seemed many people were happy with `UFL` but many in the plasma community are unfamiliar with FEM; he thought some in the community will have an equation they wish to solve, so good if they can use the DSL to implement it on their own (separation of concerns between the equation and the FEM used to solve it). WA asked whether the NEPTUNE community should put effort behind `UFL`, given that most `UFL` users are not in the fusion field. BD said `UFL` was promising. PF added that `UFL` was becoming the language of choice in the FEM community and is good for comparing FEM runtimes.

BD asked about methods for transforming higher-level mathematics into the weak form used in `UFL`. PF cautioned that there are many possible variational forms for the same mathematical equations, different Sobolev spaces etc., so automating this is probably impossible. WA questioned whether something like this existed - biased to providing a ‘robust’ solver in all cases - PF unsure, but added that a least-square coercive approach would be robust but conditioning and performance would always be suboptimal (WA agreed and there was an astrophysical application by Wiegelmann which might be admitted to be at least 100 times slower). PF opined that the approach should be to try to ‘automate out’ the computer programmer and not the mathematician or the physicist.

WA raised the issue that the fusion equations may become very complicated and prone to errors / typos in implementation; PF agreed that it might be much quicker to implement the equations in *Firedrake* that for the mathematician to try to debug the equation system by inspection.

WA said one goal was to educate the user community about aspects of FEM, therefore expose some of the options (eg. element order, basis type). PF agreed and said we should try to give enough education that users can avoid common pitfalls. SS mentioned some of the options in FEM and added that people like the strong form and the nodal basis (as ‘easier to think about’). PF mentioned that not all possible discretizations are stable, citing compatibility conditions. WA mentioned that workers on the European Boundary Code project are getting good results with Discontinuous Galerkin, as this is typically more stable than classical Galerkin - SS added that there is a large literature on DG stability, Riemann fluxes etc.

WA wrapped up the session, stating that this meeting was more about discussion than reaching firm conclusions (and added that we have yet to define a DSL). GM made the point that there is a higher-level maths / physics interface as well as a lower-level hardware abstraction layer dealing with loops as its input (GM’s work concerns the latter of these two). PF mentioned that `Firedrake` takes a `UFL` input and produces computer code output so acts to separate concerns. WA added that there is a great deal of scope for additional physics / more species in our equations - is there a need for more software in consequence? WA asked whether new functionality can be added to `OP2` if we need (eg. PIC codes); GM said he had yet to try PIC codes and that these need their own abstractions; it was mentioned that Steven Wright has worked on PIC via Kokkos, so a discussion to be had there.

B Edited Minutes of DSL Workshop 2: 23 July 2021

List of attendees (some were unable to be present for the entire meeting):

- David Moxey, Exeter
- Chris Cantwell, Imperial
- Bin Liu, Imperial
- Spencer Sherwin, Imperial
- Aidan Chalk, STFC
- Rupert Ford, STFC
- Xaiohu Guo, STFC
- Sue Thorne, STFC
- Wayne Arter, UKAEA
- James Cook, UKAEA
- John Omotani, UKAEA
- Joseph Parker, UKAEA
- Will Saunders, UKAEA
- Ed Threlfall, UKAEA
- Gihan Mudalige, Warwick
- Steven Wright, York

B.1 A whirlwind tour of `PSyclone`, Rupert Ford (STFC Hartree Centre)

`PSyclone` is a collaboration between the Hartree Centre, the Met Office, and the Australian Government Bureau of Meteorology. Aidan Chalk works on the Hartree Particle DSL. There is a potential collaboration between Hartree and UKAEA to examine PIC codes for NEPTUNE (to be ratified).

Motivation is the usual *three Ps*: performance, portability, productivity. A single-source science code is desirable (for maintenance reasons) but the complexity means there is unlikely to be a single optimal solution (RF mentioned Kokkos is usually a good compromise). Separation of concerns between code specification and optimization is desirable. Programming approaches divide into hardware specific (eg. CUDA, HIP), functionally portable (though probably not widely performant), and potentially portable (with performance approaching hand-typed across platforms).

Possible approaches to the latter include MPI, MPI+Kokkos/RAJA, MPI+task-based (HPX, Legion) - the speaker thinks this is the most forward-looking, DSLs, libraries. Various other tricks are possible eg. cache-blocking, macros ...

Outline of `PSyclone` 1: it supports Fortran due to complex millions-of-lines Fortran legacy code and many, many components. Schattler quote says that Fortran is still the main development language. `PSyclone` does not preclude other languages, however.

Outline of `PSyclone` 2: configurable - a domain-specific (API-specific) compiler. RF presented the major APIs currently in use: LFRic is mixed finite elements; NEMO ocean model is finite volume / difference, where existing parallel code is transformed not generated - Fortran coders did not want to re-structure their code so the DSL does the transformation; also GOcean API, which is a 'playground' for new approaches.

Outline of `PSyclone` 3: tool for HPC experts. RF noted that it was hard to beat a(n expert) human; optimizations are a Python 'recipe' which could be automated; profiling options are included. Example of LFRic parallelization in which DSL identifies and parallelizes loops.

RF showed the structure of `PSyclone`. PSyIR is intermediate language-independent realization - largely generic but small parts specific to each API eg. LFRicIR - from this, backends generate parallel code. Also SIR - explained later.

A slide on DSL abstraction showed trade-off between levels of specificity; note 'performance' here really means 'performance-portable'. RF showed where LFRic and NEMO sit on this scale.

LFRic example - MPI / OpenMP part is entirely hidden from developers (in fact, there is a large emphasis on letting existing Fortran codes continue working in the way they always have done ie. DSL abstracts away parallelization); one weakness is that GPU support is still in development. RF showed matvec benchmark of GPU performance of a small part of the code - two times speed-up. LFRic was parallelized only in 2016 and then shown to produce identical results to serial code; first weather / climate simulations done May 2021 and appear plausible.

NEMO example - Orca SI3 models ocean and sea ice. `PSyclone` script inserted 3,315 OpenAcc kernel directives. MEDUSA is bio- / geo-chemistry code, now working on GPU but not optimized. NEMO VA code runs on GPU but implemented by hand (not `PSyclone`). NEMO with sea ice V100 shows two times speed up - very good.

Beyond Fortran - there is OpenCL backend with `PSyclone` OpenCL nearly as fast as manual OpenAcc. Developers chose OpenCL as it can target FPGA (for which they had funding at the time).

Kokkos / SYCL new C++ frameworks. RF showed GOcean API manual Kokkos and SYCL implementations performance comparison inc. eg. Kokkos views vs. Kokkos raw pointers. Nvidia can't use OpenMP so used OpenCL. Kokkos seems to have widest spread of successful target devices and acceptable performance across all, but 'best' performance option varies by device, making performance portability choice ambiguous.

SIR backend (ie. translated PSyIR to SIR) intended to facilitate DSL interoperability (EU ESiWACE(2)). SIR-Dawn generates optimized CUDA code. Also there is OP2/OPS backend proposal with Gihan Mudalige. Also MLIR (Multi-Level Intermediate Representation from LLVM). As example, DSL can translate Fortran $\rightarrow$ CUDA, so really a sort of Rosetta stone for programming languages. Apparently also other functionality eg. code transformation, between tangent-linear and adjoint models. However the emphasis on interoperability here stems partly from the need to sustain a large Fortran code base, a relatively minor issue for NEPTUNE.

Questions / discussion:

Will Saunders asked to see an example of how the API was typically used, and whether the user was expected to provide an entire program or just a kernel.

WA asked, further to this, whether there were any constraints on the input code; FR replied that `PSyclone` can parse all Fortran loops, but SIR needs standard triply-nested loops; there is a translator between array notation and standard loops. WA asked specifically whether 'do while' was handled and FR admitted that some code is not supported eg. 'do while' and 'write' statements, though 'code blocks' can be left in native form and passed through the toolchain *(presumably restricting the output to be Fortran)*.

Steven Wright asked about the difficulties of debugging in a framework involving multiple layers of DSL translation; RF said this was a much-asked question (and also referred to Gihan's presentation of the last workshop). The output Fortran can clearly be debugged using standard techniques, although the ease here depends how 'nice' the generated code is, (but as the framework is just loop-based parallelism this probably isn't much of an issue). Alternatively, just debug the input code. RF acknowledged the possibility of bugs in the DSL but mentioned that these only need fixing once (then the fix applies to any code using the DSL). Apparently the Met Office find no problem just debugging the output code; note also there are runtime checks and tools in `PSyclone`. SW mentioned Kokkos also has debugging tools eg. data naming, bounds checking. Gihan indicated that debugging templated C++ is complex and in that case the best option is to work backward from the generated code. RF reinforced the point that Fortran is simpler than C++.

RF showed slide expositing separation of concerns aspects: algorithm, `Psyclone` layer (parallelizes loops - 'iteration space'), kernels (work); (structure called `PsyKAl`). Code translation occurs in the kernel layer. Resembles `Op2` with OpPar having equivalent method here, called Invoke. There are global operators for fields defined all over the Earth's globe. User contract is with DSL; logically global algorithm provides metadata for kernels (via arguments). WA question that system seems set up for structured data - asked about unstructured case. Answer is higher-level DSLs or extensions for more flexible layouts. Also RF emphasizes hard-coded bits are only interior to kernels. WA stated that the restricted data structure is reason why NEPTUNE not considering using `PSyclone`. RF suggested handling unstructured case via backdoor. WA asked Gihan Mudalige whether unstructured case can be handled in OP2 - GM said `PSyclone` is fairly close to OP2 esp. in terms of interoperability ideas - RF confirmed OP2 influenced the Gung-Ho project that spawned `PSyclone`. WA asked how much effort would be needed to handled unstructured grid and particles; RF replied that there is currently a project trying different approaches to adding particles, but that the front end will require considerable work, and that different abstractions are necessary in the particle case. Overall, it seems he is not pushing `PSyclone` as a framework for particles - and not pushing it for NEPTUNE because he is aware that we do not intend to use Fortran extensively (if at all). RF confirmed that the handling of Fortran is really the USP of `PSyclone` and that there are no plans to deviate from this ethos.

B.2 Summary of previous workshop, Will Saunders (UKAEA)

WS explained that we anticipate a wide range of users falling into three broad classes: engineer / physicist where the code is a ‘black box’; researcher / developer; low-level developer. We anticipate a structure with a low-level language that consumes a high-level language, possibly via an intermediate representation. The target DSL is intended to be uniform across domains eg. FEM and particle representations. The code is intended to be a ‘living form’ ie. modifiable; also actionable ie. verifiable and providing information on model error. From previous discussions, a LaTeXparser is one option for the high-level DSL. Generally, selecting the correct abstractions is vital. WS cited the talk by Gihan Mudalige at the previous workshop re abstractions enabling separation of concerns while allowing ‘escape hatches’ to handle unforeseen cases / allow ultimate flexibility.

High-level DSL could be implemented via Python (either embedding the DSL or by letting Python consume external DSL). Python becoming HPC-acceptable, using compiled code for performance eg. Numba / Cython, although there were still issues re parallelization with Python. An alternative path is to use Julia (uses multiple dispatch ie. overloading instead of object-orientation. Julia is an emerging force in HPC; there are now MPI bindings. Note it uses JIT compilation with a LLVM (can handle SIMD, threading, and GPU).

Low-level DSL options include SYCL (does not mandate ownership of memory, may require per-architecture implementations); Kokkos / RAJA (both are C++ frameworks for loop-based parallelism-note that in RAJA, managing host and device memory is the responsibility of the user); Julia, even. One general issue is limited vector reduction support.

B.3 DSLs for NEPTUNE UFL, Patrick Farrell (slide presented by WA)

WA presented a slide supplied by Patrick Farrell of Oxford University (who was unable to attend in person). This concerned the ‘high level’ aspect, explaining that UFL is a worldwide standard DSL for finite-element discretizations (PF is a major contributor to UFL via FIREDRAKE). The option presented here is to use an extension of UFL (and thereby Python). UFL has been taken up by key international efforts toward exascale eg. `DUNE` (EU) and `MFEM` (US). The main advantages are painless auto-differentiation for Newton’s method; adjoints for optimization / error estimation; ease of adding discretizations and new terms. The input uses the weak (ie. variational) form. In principle, compilers can adapt code to various hardware platforms, though in practice GPU support is in its infancy. There are ‘escape hatches’ to use if a problem does not fit within the current scope of UFL, and the language can also be extended (specifically, PF is happy to extend UFL to tackle problems from NEPTUNE).

B.4 High-level DSL discussion

Following his presentation of PF’s slide, WA asked whether anyone else had any impromptu contributions - there were no additional slides but there was a lively discussion. Steven Wright asked whether there was any DSL for particle methods (he had not discovered such during his researches, just libraries). RF replied that Aidan Chalk is the DSL developer for `PSyclone` and

AC (I think) mentioned that for particles, abstractions would be needed for eg. particle-particle interactions, local interactions, cut-offs etc. : UFL is not designed to handle such things and a different front end would be needed. AC likes the NEPTUNE split into high- and low-level DSLs (`PSyclone` intermediate representation is rather similar). AC showed slide outlining separation of concerns (science content versus performance aspects of the code) - his scheme uses Regent (LLVM framework from ECP) - currently exploratory and not a concrete choice. He continued to discuss his work on particles: there are three key ideas: a particle type object, a set of kernels (eg. pairwise, per-particle), and an overarching program, which is basically the declarations and timestepping loop (at this level, the interface is kept lightweight - aimed at scientific developer). The DSL / framework handles algorithms, implementations, and I/O handling. Users can add new force laws or use inbuilt (eg. Lennard-Jones). I/O handling via DSL was highlighted as an ‘elephant in the room’ - it is desirable to insulate the user from onerous data management. Will Saunders presented a website containing particles materials from his own PhD (which was a DSL for particle methods); he showed particle data, Verlet time-integrator, kernel encapsulating algorithm for time-update, obvious big loop over particles, pairwise interaction kernel. The DSL (written in Python) generates C code for CPUs and GPUs. One consensus was that particle DSLs do exist, though they may be located in the molecular dynamics domain. Different context from NEPTUNE because in MD, there is no tight coupling between particles and the electromagnetic field, though there are some commonalities eg. need to loop over all particles. Xiaohu Guo from STFC indicated that the majority of particle methods fell into two types: fully-discrete and semi-discrete, which classification was probably meant to distinguish SPH as a semi-discrete method, ie. representing only the velocity field of a fluid, from discrete methods which allow for non-Maxwellian behaviour such as PIC and MD (although other interpretations are possible). WS indicated that molecular dynamics involved eg. gathering statistics and calculating transport coefficients. WA added that each of the ‘particles’ in NEPTUNE could actually represents $10^8 - 10^9$ physical particles.

The discussion reverted to AC’s slides, in particular efficiency and details of the implementation in Legion. AC confirmed that all difficulties encountered lay in the transfer of algorithms (eg. from SPH, SWIFT cosmology) to the Legion runtime system: the Legion memory model represents a challenge (C / C++ much easier). Performance problems were those generic to particle methods (the implication is that Legion was not contributing its own performance issues). WA added that one benefit of Legion was its handling of load-balancing. AC is tuning kernel performance, controlling cell size ... he expects native Fortran levels of performance once completed. WA said large cells were to be expected if using higher-order finite elements (the main NEPTUNE problem is switching from particle to fluid representation as the local density increases, and back again in the converse case).

For completeness, WA then presented a slide on Daedalus (a fluid-based code) DSL (he gave a recap that the previous DSL workshop had exposited UFL and the BOUT++ DSL - the latter is not dissimilar to Daedalus’ DSL). In Daedalus, the position of equation terms to left or right of the $=$ sign specifies implicit or explicit treatment. WA showed a slide from ICOSAHOM 2020 the previous week showing that the DSL now handles vector fields. There are thus three examples here of ‘Pythonic’ input DSLs.

Steven Wright asked about coupling between particles and fields - eg. having something in the field timestep to handle particles. WA clarified saying that the Maxwell (or Ampère) equations would be solved in parallel with particle motion, with the particles acting as point source terms for continuous fields eg. total charge in cell acts as source in Gauss' law (with currents and magnetism also). Going the other way (ie. continuum to particle) is more of a problem and is handled by analysis techniques eg. those of Felix Parra (ie. phase space); only in the sheath do we want to use an explicit PIC code. Will Saunders said his idea was to have operators acting on particle state that give output in FEM space with operations to abstract this kind of thing. WA agreed with this approach, mentioning that the operators had a number of necessary properties eg. conserving charge. James Cook made the point that such operators would involve expensive quadratures if weight functions were involved, and indicated that Sonnendrücker approach (delta-functions) might be better. WA concurred that Sonnendrücker is exploring this in the context of Lagrangian formalism and that we should study it; as to how such things would work within UFL, Patrick Farrell should be consulted. JC further indicated that Hamiltonian splitting methods are involved / complicated. WA agreed (citing the equations he presented at the start of the session) and said that a current task is to design a version of UFL incorporating these things. For particle work there is a new call - T/AW084/21 - and also UKAEA staff - WS, JC, Joseph Parker - looking into relevant algorithms. (Note added later - this call recieved no responses.) The aim is to design a high-level DSL, then in a good position to start thinking lower-level.

B.5 Low-level DSL discussion

Will Saunders put up slide on SYCL: WA opened debate re SYCL vs. Kokkos / other. James Cook said he had tried SYCL and HPX (though not Kokkos / RAJA) and had found SYCL very easy to start using (ie. well-designed), while for HPX he had had to work carefully through the examples (this is really the 'learning curve' or the start of the 'productivity' aspect of the code generator). Re learning curve, WA said this is why he favoured LaTeXparser as part of higher-level DSL (though the discretization would need to be in addition); he agreed with the comments re SYCL from his own experience. Steven Wright commented that the issue with SYCL was compiler support: the Intel C++ compiler worked well on Intel hardware, but for CUDA it was necessary to download correct LLVM and build with CUDA support; he implied rather hard to get working, and similar for HipSYCL, although the Codeplay implementation - ComputeCpp - has out-of-the-box Nvidia support. Also from SW: difference between RAJA and Kokkos is 'for' syntax. He indicated that overall, it is not that difficult to switch between these loop parallelism paradigms currently, though new features may change this situation. Gihan Mudalige spoke about his (extensive) experience with SYCL: he was forced to generate different codes for different architectures; had to resolve parallel race conditions when using unstructured meshes; lack of double-precision atomics support meant he had to use a colouring scheme, so that a degree of hand-coding was necessary to get portable performance. Compiler support for SYCL potentially suffers from 'developer lag' vs. native APIs (eg. CUDA). GM also has doubts about Intel FPGA support. (Such support is not expected to be an important issue for NEPTUNE.) WA agreed that SYCL would probably not offer immediate holy grail of write once, perform well everywhere code, and that the main issue was the long-term survival of the language, which might of course also apply to others eg. Kokkos. WA then proposed SYCL as a technology for NEPTUNE and invited participants to disagree at will. GM strongly advised not getting locked-into a particular vendor i.e back-end. RF made the point that surely the DSL was what allowed avoiding this kind of tie-in, and that the real question was simply what option to explore first. WA answered that the choice was C++ with the best available choice of performance portability layer; given the complexity inherent in plasma physics, we need low-level control (with a view to abstracting away implementations in future). Most efficient way to start may be to leverage Intel support (use SYCL) but keeping in mind the need to remain vendor-independent. Steven Wright seemed more agnostic as to choice of low-level DSL but did indicate that his experience with Intel compilers was positive and that using these leverages extensive skills of compiler writers. Will Saunders made the point that abstraction limits the amount of code needed to port between different choices of abstraction layer eg. restricted to loops and data types; WA expressed it as abstraction layers giving a set of restriction guidelines. RF asked the question whether SYCL has been shown to be sufficiently performance-portable, saying that this was the case for Kokkos and so the latter was perhaps a more proven technology: WA cited results by Simon McIntosh-Smith. SW mentioned that OpenCL has been shown to be fairly good (SYCL is basically OpenCL) and also that he has contacts at Intel who might be willing to tune-up code (SW seemed generally happy with the performance portability of SYCL). By contrast, GM was not convinced that performance portability had been demonstrated for particle codes as existing tests usually assume a structured mesh that makes vectorization straightforward ... indirect access makes things more difficult and uncertain (he also mentioned risk of affiliation to a sole vendor, citing historic example of the now-abandoned Intel MIC architecture - ie. Xeon Phi).