Eleventh International Conference on Computational Fluid Dynamics (ICCFD11), Maui, HI, July 11-15, 2022 **ICCFD11-2022-0802**

XCOMPUTE: Algorithms and Instruction Sequences for CFD/FEA Multiphysics

G. J. Orr[*], R. J. Kwan[*] and M. Doudar[*]
Corresponding author: `graham@xplicitcomputing.com`

[*] Xplicit Computing, Inc. USA

Abstract: Despite exponential growth in CPU & GPU bandwidth, many computer-aided engineering (CAE) software tools still cannot fully leverage hardware across teams and systems. To solve this deficiency, we propose an efficient universal computational pipeline for geometry preparation, physics definition, condition assignment, solver numerics, and other useful data-processing functions common to computational fluid dynamics (CFD) and finite element analysis (FEA). The objects and interfaces of the XCOMPUTE pipeline are detailed here: an executable *instruction* comprises an algorithm/verb bound to one or more argument(s)/noun(s) and sequenced into physical models and solvers to define high-level workflow procedures. Instructions and sequences are dynamically assembled from human-facing building blocks to allow a variety of customizations and optimizations. Algorithms can be defined by static host bytecode and/or implemented as a device kernel as part of a compiled OpenCL program, callable from a solver program. Property-key inputs, outputs, and constants permit algorithms to connect in an executable sequence or be invoked individually as part of a polymorphic compute framework. Multiple networked client and server sessions interact in real-time using the *xcompute* protocol, leveraging new *xccommon* and *xcmessages* libraries. Execution modes include: interactive mode, command line shell, and application-generated cases.

Keywords: Computer-Aided Engineering, Integrated Engineering Environment, Systems Engineering, Multiphysics Simulation, Computational Fluid Dynamics, Signed Distance Field, Parallel Computing.

1 Introduction

In the four years since ICCFD10, computational hardware performance has vastly increased, yet computer-aided engineering (CAE) and simulation software have not experienced comparable performance gains. At the last conference, we presented a paper [1] on the programming basis of our geometry kernel. We have since completed the XCOMPUTE thesis and are working toward the public release of the XCOMPUTE software.

Among some of the algorithms incorporated are the finite element method (FEM), finite difference method (FDM), lattice-Boltzmann method (LBM), level-sets, grid-gen, mesh-gen, and even Langrangian approaches. We demonstrate progress in accelerated steady-state and transient analyses for thermal diffusion, strain-stress, thermo-elasticity, thermal radiation, contacts & coupling, compressible Reynolds-averaged Navier-Stokes (RANS), compressible and incompressible large eddy simulation (LES), far-field quantum-electro-dynamics, and near-field Maxwell electrodynamics.

In our view, it is a grand challenge to develop a unified framework that implements powerful, disparate numerical algorithms such as these and executes them concurrently across heterogeneous, distributed computing resources. It is our intent to deploy this framework in a sustainable way that benefits a broad community of engineers. This paper describes our solution to this challenge.

The goal of this paper is to help developers build well-encapsulated and interoperable modules using simple building blocks. We believe clean inheritance models and abstracted code provide the foundation for design, development, and improvement of integrated CAE systems. The primary characteristic of an orthogonal base set of systems and algorithms is that every concept is unique and presents its own characteristic benefits and challenges with surrounding code and machinery. The agglomeration and arrangement of these smaller units constitutes a complex numerical process or workflow; such a standardization is the only way for a performant unification to minimize error and maximize return.

The rest of this paper is structured as follows. The technical and engineering management motivation for XCOMPUTE are described in Section 2. This is followed by an architectural overview in Section 3. The key building blocks of XCOMPUTE are described in Section 4. One of those building blocks is a *system*; Section 5 describes the members of a system instance and how they interact within and across systems. The execution modes of an *xcompute* process is described in Section 6, including examples of the *Messages*™ file & wire schema in C++ and Python. Finally, a brief conclusion is given in Section 7.

The source code for *Messages* is provided on GitHub under a BSD-3-Clause open source license. [2, 3]

2 Problem Statement

High-performance computing perfomance has roughly doubled every year over the past few decades. For example, the Rmax metric for LINPACK used by the Top500 survey shows a steady, exponential progression from 59.7 gigaflops in June 1993 to 1,102 petaflops (1.102 exaflops) in June 2022 [4] – yielding an 18 million-fold increase over 29 years. This dramatic increase is the result of a spectrum of technical innovations in circuit fabrication technology, CPU architecture, clustering, vectorization, GPU-based accelerators, etc. Software for computer-aided engineering and simulation benefit from some of these innovations, but often cannot keep pace with the broad spectrum of progress. As engineering becomes increasingly dependent on computation, the capabilities of software limit engineering productivity and responsiveness of designs to new insights. Allowing engineers to take greater advantage of these innovations increases the number of available design iterations or the fidelity of simulations.

A 2014 survey [5] of 248 manufacturers found that engineers report spending a third of their time on non-value added work, and 20% of their time working with outdated information, often resulting in wasted effort and rework. An analysis of survey responses determined that the largest contributor to non-value added time is related to trying to find information, indicating that data management practices have a significant impact on engineering efficiency.

Furthermore, a 2004 NASA study [6] found that the cost of fixing errors at later stages in a project can become over 1,000 times more expensive than earlier stages in the product development lifecycle, revealing that costs escalate exponentially. Increased emphasis on finding errors early in the project lifecycle means spending more time and a larger percentage of project costs in the definition phases of a project — more than is usually allocated to the early phases.

The availability and organization of updated data are crucial to the efficiency and ultimate success of complex multidisciplinary engineering projects. Many challenges and bottlenecks exist in modern engineering processes, often stemming from isolated CAE tools and other disconnected software, and these issues frequently compound, causing a cascade of subsequent errors. Sometimes, these issues may be as simple as an improperly named file or an ambiguous function name.

In this paper, we outline XCOMPUTE: an efficient universal computational pipeline for geometry preparation, physics definition, condition assignment, solver numerics, and other useful data-processing functions common to computational fluid dynamics (CFD) and finite element analysis (FEA). XCOMPUTE addresses significant long-term challenges in CAE processes, spanning systems engineering, mesh generation, fluid dynamics, stress mechanics, electronic design, robotics, language processing, and machine learning, together in a collaborative environment.

3 Architecture Overview

XCOMPUTE is a systems simulation and scientific computing platform developed by Xplicit Computing that enables teams to resolve complex numerical computing problems constructed from system and algorithm building blocks. XC machinery leverages contemporary C++ and OpenCL 1.2 to provide concurrent processing and interaction leveraging CPU and GPU parallelism. Systems are managed in a recursive tree, permitting complex ownership and information flow. Systems improve human and machine conceptualization of the problem domains. Each system can execute its own numerical process, typically via a solver, which can be dispatched to CPU or GPU resources.

Ideally, a CAE program is constructed using a base set of class concepts that each have a well-encapsulated role, independent from and without knowledge of non-member classes. These classes should directly interface to human constructs, as well as existing CAE machinery to facilitate complex concepts and collaboration on modern hardware. The architecture should utilize inheritance and polymorphism strategically with minimal single-use code. A single "atomic" class should be customizable to a sufficient degree as a runtime object to express any type of noun, such as a scoped system extensively defined by other objects. As modules are developed beyond the initial set of capabilities, functionalities should be expanded via the specialization of a single type of verb; a global algorithm, intensively defined by itself, exclusively interfaced by others. These characteristic objects should be bound together at runtime to yield a callable instruction with potential for fine and coarse-grain recursion.

XCOMPUTE's design converges on approximately 200 translation units (orthogonal class concepts) and four fundamental software layers at the intersection of human and machine requirements. The code leverages object-oriented template meta-programming, functional programming, and runtime code-generation techniques to maximize ease of development and user modularity, improve pattern and execution regularity, and organize meaningful object hierarchies.

XC heterogeneous applications utilize four software abstraction layers as foundation for runtime: Messages (schema) [2, 3], Common (protocol), Server/Client (application), and OpenCL/OpenGL (SIMD runtime). (See Figure 1.) Applications are constructed using submodules underpinned by only a few standard cross-platform libraries (such as libstdc++, libomp), thereby minimizing external dependencies, reducing complexity, and increasing maintainability of the codebase.

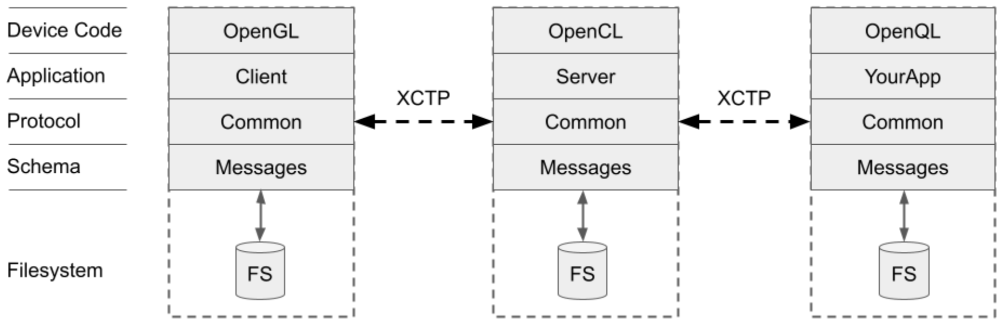

Figure 1: XCOMPUTE software stack, consisting of client, server, and other applications built on and communicating via a common protocol layer and Messages' *standardized file and wire schema.*

The simulation state machine is hosted on one or more servers, typically GPU-accelerated compute clusters, and is identified by cryptographic certificates. Admins manage LAN and WAN access via NAT port forwarding. Invited team members can access and manipulate the server state using the graphical *xcompute-client* or other interfaces on consumer-grade hardware with the built-in *xcompute* protocol; applications leverage hardware-acceleration at coarse (i.e. CPU) and fine (i.e. GPU) levels, requiring dedicated compute and render devices for *xcompute-server* and *xcompute-client*.

4 Building Blocks

4.1 Property-Key

A *property-key* (PK) is the composite of a *property* tag, followed by optional *modifier* tags, enabling:

- Human and machine readable data tagging and retrieval[1].
- Ideal lexicographical sorting & searching (in-session)
- Compatibility comparison by string name (out-of-session)
- Dynamic I/O for algorithms using modular substitution (in-code)

Using property-keys, applications can dynamically allocate memory for data scalars, spatial vectors, or tensors by referencing a particular geometry's cardinality and dimensionality. Therefore, memory allocation for varying objects is automatically handled by the bound instruction and/or sequence as part of the larger program. Additionally, dimensional analysis of physical units is easily summed between property and modifier stages, enabling physical unit calculation and implementation of *Buckingham π theorem* [7].

In numeric applications, all property and modifier objects are constructed with a static singleton-like pattern; semi-unique PKs are built from these. Map ordering is determined using an overloaded less-than (<) operator, comparing homonumerical and heteronumerical PK stages:

$$\{Null\} < \{Property\} < \{Property, Modifier\} < \{Property, Modifier, Modifier\}, \ldots$$

In an application session loaded in RAM, pointers (address references) permit optimal memory usage and comparison speed. However, when serializing data for wire transmission or file save and load, it is ill advised to use pointers across computing sessions; plus, those pointers cannot be compared across heterogeneous memory spaces. Therefore, it becomes necessary to represent the PK in a more literal format; simply, each property and modifier name is concatenated with delimiters into a character string version and returned as the serialized name (providing equivalent lexicographical sorting and handling, but requiring more CPU cycles to compare characters). These string representations of PKs are used in *xcompute-client* as handles for human-facing data map entries. Standard "comma" (',') and "pipe" ('|') characters provide suitable delimiters for concatenating and parsing PK stages between sessions. The PK name is included in *Messages::Vector* as entry and file name, and when loaded from file or wire, the string names are parsed and property and modifier likenesses are found in program memory; immediately, optimized pointer-based PKs are available for rapid data access.

Both property and modifier objects implement distinct internal machinery, but share the following intrinsic attributes:

Table 1: Member attributes of property *and* modifier *classes include a string name, integers for measurable dimensionality (unit exponents), and tensor rank to specify whether the property or modifier acts as a scalar, vector, or matrix. As required during runtime, attributes are combined into property-keys, permitting data tagging and standardized calculation of critical runtime parameters (such as memory sizes and assigned number of rows and columns). See Appendix A for a partial list of standard properties and modifiers.*

Attribute	Type
Name	Serialization string
Units	Dimensional exponent, signed integer
Tensor Rank	0:scalar, 1:vector, 2:matrix

By using PKs, any degree of modification may be applied to provide specificity to any useful data entry — the number of unique variations are essentially unbounded. If an entry already exists, a developer must

[1]Although claimed in U.S. Patent #11,373,019 held by Xplicit Computing Inc., academic and personal use of property-key patterns and logic is authorized to the extent that no service or product revenues are derived from the patented techniques; commercial use in part or whole requires a valid license from XC. Please contact `info@xplicitcomputing.com` for early-access pricing and availability.

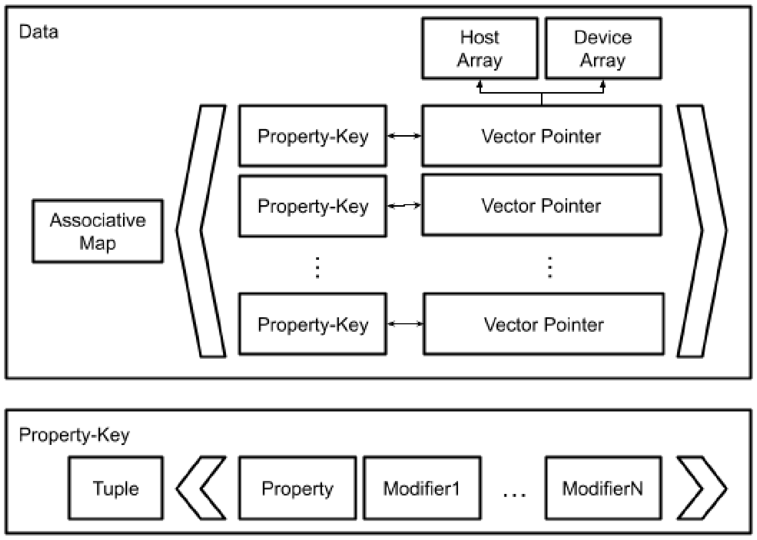

Figure 2: A property-key is used in a data map to lexicographically search the database for the corresponding key. If a match is found, the entry can be returned. In numerical computing contexts, this resolves a DeviceVector which represents data in a host array and/or device buffer. Thus, entries are accessed in logarithmic time using the CPU, returning any coalesced vector for SIMD operations.

consider if the intent of the algorithm is to make a new (modified) PK, or whether to overwrite existing data with the same PK. Modified PKs make it implicitly clear to humans and machines which operations have been performed on a specific data entry. In practice, this leads to the notion of the "key-chain" that becomes the backbone of a complex numerical process spanning thousands of systems and thousands of data entries per system, interoperating as a unified simulation.

After a numerical process mutates desired data, it can overwrite the input data, or it can be written into a new data entry using a modified PK. Often, an algorithm specifies the output property-key including any wildcards to be substituted (or resolved to concrete PKs) denoted by the *AnyProperty* or *AnyModifier* tags. For example, a gradient algorithm should be able to operate on any field, and thus, have $inputs = \{AnyProperty\}$ and $outputs = \{AnyProperty|Gradient\}$. At runtime, a PK is required to specify which data entry to substitute AnyProperty (e.g. Temperature, Pressure, Mass, Energy, etc…).

New properties and modifiers are added to the codebase as required, but transitory properties and modifiers can be constructed dynamically at runtime, permitting the same lexicographical optimized patterns. The downside to dynamic properties and modifiers is that non-standardized entries cannot be guaranteed across computing sessions as those custom names and attributes are not standardized.

Generally, for P properties and M modifiers, we can compute the number of unique property-keys C_k at modifier stage k:

$$C_k = P \times M^k. \tag{1}$$

The total number of uniques PK's is C_{total}, computed as the sum of C_k across all used modifiers to arbitrary depth K – effectively infinite when using unlimited number of modifier stages:

$$C_{total} = P \sum_{k=0}^{K} M^k. \tag{2}$$

At the time of publication, xcompute has defined $P \sim 90$ properties, $M \sim 40$ modifiers, roughly half with intrinsic attributes. More will be added as applications prove necessary.

Given $P = 90$, $M = 40$, and $k = \{0, 1, 2, 3, \ldots\}$ modifier stages, we compute the number of possible property-keys as $C_k = \{90, 3600, 144000, 5760000, \ldots\}$ and $C_{total} = \{90, 3690, 147690, 5907690, \ldots\}$.

4.2 DeviceVector

A *DeviceVector* is a template container for contiguous host and device data entries, utilizing revision numbers for synchronization. DeviceVector inherits from row major *Eigen::Matrix* [8]. Its members are described in Table 2.

Table 2: The members of the DeviceVector container.

Member	Description
Buffer	optional contiguous device memory that corresponds to data entry
Revision	major and minor increment to track memory and value updates, respectively
bool sync()	synchronization data between host and device, performing any necessary *writeToDevice()* and *readFromDevice()* to reconcile revision numbers

4.3 Data

A *Data* container maps PK to vectors whose members are described in Table 3. Other related useful functions are listed in Table 4.

Table 3: The members of the Data container.

Member	Description
Name, Series, Revision, Iteration	basic identifying information for the dataset
map<PropertyKey, DeviceVector> Record	associative container mapping property-key entries to corresponding values

Table 4: Useful functions.

Member	Description
bool contains(PropertyKey)	check whether data has a specific PK entry
DeviceVector get(PropertyKey)	get a specific PK entry by reference
bool set(PropertyKey, value)	set a specific PK entry upfront
bool operator>> (...)	stream data into repeated *Messages::Vector* protocol buffer
bool operator<< (...)	stream data out of repeated *Messages::Vector* protocol buffer
bool save(string)	save multiple XCO data to file directory
bool load(string)	load multiple XCO data from file directory

4.4 Geometry

A *Geometry* container allows for structured and unstructured elemental topology, specifying how nodal point positions are connected via discrete edges, faces, or cells. Users and algorithms can specify regions of interest as subsets of elemental topologies: groups of nodes, loops of edges, surfaces of faces, and volumes of cells.

Nearly every analysis requires a computable geometry, often starting with a boundary shape representation such as an STL file. Creating engineering-scale volumetric meshes and grids has been an ongoing pain point [9]; fortunately, the Signed Distance Field (SDF), is an elegant intermediate computable shape format, able to support everything from complex volumes to wafer-thin substrates. SDF acts as a flat shape representation that can be sampled directly by CFD numerics, permitting creation of computable discrete topologies.

See ICCFD10-145 [1] for more information on polymorphic computable geometries; see Appendix B for more information on the emergent Signed Distance Field.

4.5 Algorithm

An *algorithm* is an intrinsic operator (verb) — a reusable global callable function-class defining a procedure to achieve a desired numeric operation.

An algorithm is defined by its procedure (as code) and the argument types required for processing. Objects are not directly bound to an algorithm, since algorithms have a singleton-like pattern; they are referenced and invoked as needed within other algorithms, models, and solvers bound to instruction and argument objects — itself seldom explicitly constructed or mutated. Property-key inputs and outputs can be specified upon algorithm construction, later interpreted by sequencing machinery to connect relevant data inputs/outputs in a chain. An algorithm's callable operator dispatches static host bytecode, and provides an analogous method to define dynamic OpenCL code fragments to be assembled and compiled into a device program at runtime.

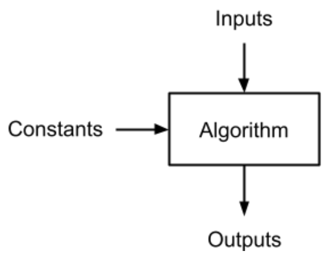

Figure 3: An algorithm defines functional code which can operate on objects including data inputs, constants, and outputs.

Both methods leverage C++ functional polymorphism to specialize algorithm behavior while maintaining a standard interface, including:

4.5.1 Requirements

A set of object types expected as bound arguments to execute the defined set of routines. Types are specified upfront by the developer within an algorithm constructor which includes a string comment to describe the object. Requirements are checked against bound arguments during solver assembly.

4.5.2 Inputs

PKs expected as provided data. Input data is assumed to be pre-allocated by an upstream process, which might be the result of another algorithm's outputs or user's inputs.

4.5.3 Outputs

PKs produced as resultant data. If an entry does not exist, output data is automatically pre-allocated by the solver so that algorithm results may be stored.

4.5.4 Constants

PKs expected as non-varying constants. An input can be substituted for a data constant; true constants are rare, as often these parameters vary.

4.5.5 Functions

A set of key functions support the instantiation and use of algorithms. Some of their signatures are given in the listing below. Important functions include: "*init*" initializes a set of default arguments to an instruction. "*bind*" associates an instruction with required argument objects. "*prepare*" populates any includes, and allocates memory required to invoke the instruction. "*operator()*" invokes the algorithm. "*clCode*" generates OpenCL kernel source programmatically.

```
virtual bool init(Instruction&)
virtual bool bind(Instruction&)
virtual bool prepare(Instruction&)
virtual bool operator()(Instruction&)
virtual bool operator()(const heterogeneous_map<Arguments>&)
virtual string clCode(const heterogeneous_map<Arguments>&)
```

Below is a generic example algorithm passed a set of arguments from an instruction:

```
bool ExampleAlgorithm::operator()(const heterogeneous_map<Arguments>& args)
{
    auto& data = args.get<Data>();
    // check to ensure data contains the input PK
    if (!data.contains(inputs[0]))
        return false;
    // get references to vectorized data entries using PKs
    auto& in_vec = data(inputs[0]);
    auto& out_vec = data(outputs[0]);
    // iterate through all entries in parallel and write to output
#pragma omp parallel for
    for (auto i=0; i<in_vec.size(); ++i)
    {
        out_vec[i] = some_function(in_vec[i]);
    }
    return true;
}
```

4.6 Arguments

Arguments contain bound objects to be operated upon, often passed into an algorithm's callable function. Arguments can be of arbitrary type, but often contain a pointer to a system, and sometimes references to data and geometry or other bound objects if the algorithm performs some special operation on those types. A standard overridden function signature with a type map is required to permit variadic argument patterns while maintaining a standard function interface. If a plurality of objects of the same type are required, then a container is implied and warranted. The optimal template container type can usually be implied by the nature of the contained objects and use-pattern (e.g. *std::set* vs *std::vector*, etc); it is impossible to know which types and containers will be most useful in a broad-use computing platform so it is only appropriate that a unique mapping between types and bound objects be provided as arguments.

4.7 Instruction

An *instruction* is an executable instance comprising an algorithm bound to arguments as part of a compute program.

An instruction performs the operation of binding objects together in a temporary callable instance that can be invoked with a managed sequence. Inputs and outputs can be functionally queried, returning underlying PK inputs and outputs with relevant *AnyProperty* and *AnyModifier* substitutions (necessitating a property-key as an algorithm requirement). An instruction may be invoked individually with its callable operator, to be called from a sequence (and/or solver). This calling structure permits instructions to call either host functions or device kernel functions as part of a larger compute program. Any algorithmic preparation or post-processing can be included in optional before and after sequences, which can be compiled and run in siloed (hierarchical) or flattened (shared) execution modes.

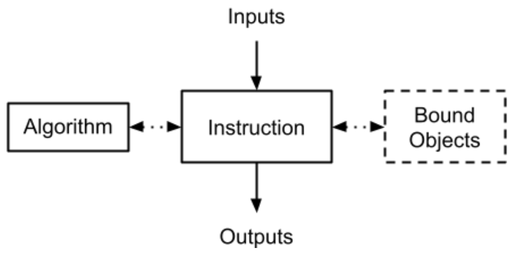

Figure 4: Instructions bind algorithms and arguments into a temporary callable that often operates on data inputs yielding data outputs.

- Algorithm - a pointer to the algorithm that defines the instruction behavior.
- Arguments - objects bound at runtime prior to solver compilation and execution.
- Kernel - an optional device program pointing to a specific kernel function.
- Before Sequence - optional algorithms that must be run before this instruction

- After Sequence - optional algorithms that must be run after this instruction

The C++ signatures of some important functions are given below:

```
bool operator()
set<PropertyKey> getInputs()
set<PropertyKey> getOutputs()
size_t SIMD()
string prepareKernelSource(...)
string getIncludeSource()
string getBodySource()
bool buildKernel(Program&)
bool setKernelArguments()
bool hasRequiredObjects()
```

The device kernel can be called from a sequence (and device program) in lieu of host bytecode:

```
for (auto& instruction : sequence)
{
    // internally calls either host bytecode,
    // or invokes kernel->launch() if available
    if ( !instruction( ) )
        break;
}
```

4.8 Model

A *model* is a collection of algorithms defining calculation rules to achieve a desired mechanism.

Model typically specifies spatial scheme, while Solver specifies the temporal integration method. Explicit flux-based methods are interoperable as we can accumulate contributions to conserve Degrees-of-Freedom (DOF). Implicit matrix methods cannot be combined as they require a linear assembly stage. These model members are listed in Table 5. Other useful functions are listed in Table 6.

Table 5: Model members.

Members	**Description**
Degrees-of-Freedom	Independent variables to be solved, often representing transported quantities or proxy; explicit flux-based methods typically transport conserved intrinsic quantities (strong form); implicit matrix methods typically transport non-conserved measurable proxy (weak form)
Calculation Rules	Mapping of PKs to set of algorithms with said PK outputs
Compatible Conditions	Region-specific algorithms to provide closure to system space-time boundaries
Compatible Solvers	Numerical methods that complement the integration strategy of the physical model

Table 6: Useful model functions.

Member	**Description**
bool canCalculate(set<PropertyKey>)	Check whether the specified output can be calculated given the algorithms in the model
Sequence getSequence(set<PropertyKey>)	Get an ordered list of instructions that represent the algorithmic path starting from constants and inputs to achieve specified outputs

4.9 Physics

Physics is a type of model. It can contain one or more models and superposed sub-models.

All algorithms are unbound until a physics model is connected to a system, binding members to arguments to yield an executable instruction sequence. Thus, physical models define desired numerical behavior, but they are independent of the objects yet to be bound for processing.

Consider the application of a general Cauchy (Dirichlet-Neumann) problem in a classical digital computer, where some numerical process (e.g. PDE) is represented by discrete operator T applied to the state defined by K degree-of-freedom $\Phi = \varphi_1, \varphi_2, \ldots, \varphi_K$ quantities to yield some discrete change $\Delta\Phi$ or state Φ in geometric domain Ω, and boundary conditions $\Phi_o(\delta\Omega)$ and any additional arguments...

$$(\Delta)\Phi(\Omega) = T(\Phi(\Omega), \Phi_o(\delta\Omega), \ldots). \tag{3}$$

The (change of) state in the domain is equal to the transformation that occurs upon the state within the domain given some set of prescribed conditions (or mediating links to other systems) on the boundaries. The far-left (Δ) expression is to indicate generality for explicit and implicit schemes.

T can be conceived as the collection of algorithms that collectively operate as the physical model and numerical method – processing upon states and prescribed conditions to transport through nodes and/or elements. This transformation can be linear or nonlinear. This description serves to illustrate commonalities between disparate methods and the importance of regularity and modularity across numerical platforms; complex behavior can result from the combination of contributions from smaller well-defined (individually validated) sub-processes T_m with their own instruction sequences of algorithms A, bound to its state $\Phi(\Omega)$, conditions $\Phi_o(\delta\Omega)$, and arguments...

$$T = \sum T_m, \tag{4}$$

where $T_m = \{A_{begin}, \ldots, A_{end}\}(\Phi(\Omega), \Phi_o(\delta\Omega), \ldots)$.

4.10 Sequence

A *sequence* is an executable container of instructions that assists in object binding and solver preparation (see Figure 5).

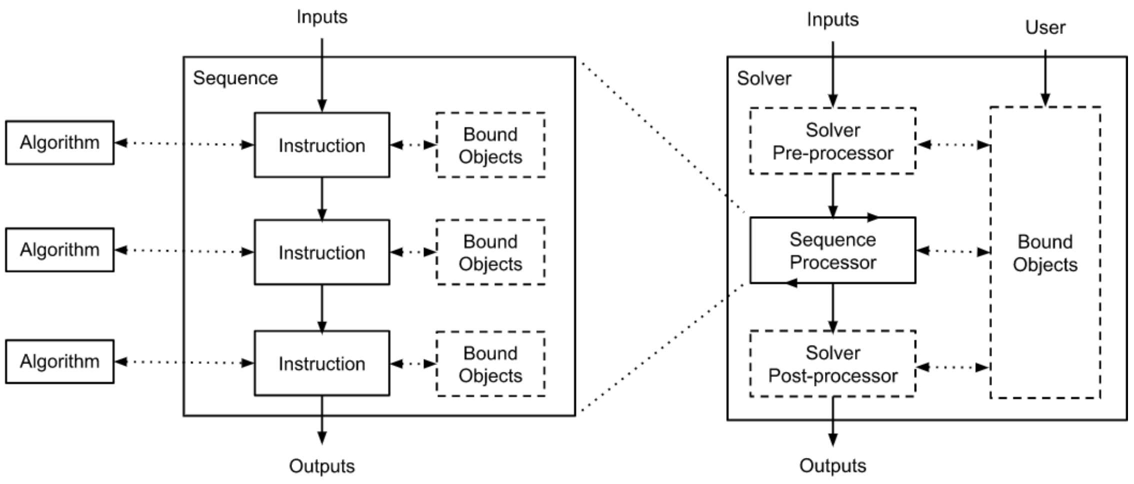

Figure 5: A sequence (left) consists of one or more instructions bound to their respective algorithms and objects. Such sequences can be used independently, or embedded in the solver as the main iterative processor (right). A solver can optionally include a preprocessor and postprocessors, which are also solvers (containing sequences). Often, objects are bound at a high level through a sequence or solver, but object binding at the instruction level is also useful for fine-grain control. [1]

Algorithms are hardcoded and immutable in static bytecode, but with the help of runtime containers, unique high-level behavior can be achieved via custom sequences to comprise physical models and numerical methods (aka. "solvers"). Together, the physical model and solver transform the PDE into a discrete ODE, typically integrating in spatial and temporal dimensions, respectively. Given a sufficient numerical-physical stability (e.g. CFL criterion), information is exchanged throughout the domain in space-time and a solution converges to some acceptable criteria.

Let's first consider the following code snippet, which depicts a simple runtime sequence that is populated with two instructions and executed from front-to-back:

```
sequence.push_back(Instruction(&SecondAlgorithm, someSystem, ...) );
sequence.push_front(Instruction(&FirstAlgorithm, someSystem, ...) );
// ...
sequence.execute();
```

In order to change physical behavior during runtime, T must be functionally changed, which is not possible for static binaries created with modern compilers unless virtualization is employed. We essentially must abstract a virtual computing machine inside of software to permit the customization of instruction sequences during runtime. This way, developers and users can create new complex functionalities with building blocks. To facilitate this, when an instruction is constructed, it references an algorithm that defines the behavior in code. The instruction also binds arguments such as a system, data, and geometry by reference prior to execution. The state proceeds through the sequence as input and output property-keys for each algorithm, forming compute pipeline T. Within each algorithm's usable executable function ("callable operator"), instruction arguments can be accessed. Consider the following code to access a bound system:

```
auto someSystem = instruction.args.get<std::shared_ptr<System> >()
```

In this example, the system's shared pointer *typeid* enables lexicographical map look-up by desired type, dereferencing a numerical system (e.g. some simulated component), which has data, geometry, physics, and conditions. Some algorithms such as conditions require a separate bound Data to house those desired boundary values separate from the system's data. Most algorithms will only have 2-3 required argument objects. Once an object has been inserted into an instruction's argument type-map, it can be accessed from within the algorithm's callable operator. In the rare situation when more than one specific type is mandated for an algorithm, a Standard Library container is warranted, with careful consideration given to the type, depending on the problem statement. It is recommended to minimize the number of argument requirements, making the modular interface and intended objects' purpose clear to developers and maintaining generality where possible, such as in the following algorithms overridden callable operator:

```
virtual bool Algorithm::operator()(Instruction&){//static c++ function here...}
```

In order for software users to alter the definition of a physical model or solver, T must be updated. To mimic dynamic dispatch on device, a standardized callable function signature is required to create and call optimized functions for given argument types. If the numerical program is written in a static language like C or C++, then the application must be stopped and re-compiled against source code. If the method is expressed in code that can leverage just-in-time (JIT) compilation such as OpenCL, then the application need not be stopped; code is assembled providing an opportunity to include structural and functional runtime optimizations. This process is automated, following virtual function overrides and returning code fragments for each algorithm.

```
virtual std::string Algorithm::code(Instruction&){
  /* dynamic opencl code-gen here... */}
```

4.11 Solver

A *solver* is an executable sequence and heterogeneous compute program manager. The solver base class is useful to assemble custom complex scripts at runtime. Derived variations override the build function to populate its sequences given a numerical method as shown in Table 7.

Most compute programs are heterogeneous, comprising a mixture of host- and device-capable algorithms managed by each system's solver. Because compiled host binaries cannot change while running, end-users

Table 7: List of some of the available solvers.

Members	Description
Explicit solvers	Solve-Tree, Runge-Kutta (FDM, FVM), Stream-Collide (LBM)
Implicit solvers	Linear-Steady (FEM), Linear-Euler (FEM), Linear-Newmark (FEM)

must redefine instruction sequences to achieve custom workflows. Given a numerical process defined by a Solver (populated with physics and conditions), instructions are defined as algorithms bound to some arguments. Device-capable algorithms are processed using specialized mutable C-like code (for a single thread within a SIMD work group), compiled by and targeting an OpenCL ICD runtime driver, enabling compute programs to emulate C++ dynamic-dispatch (enabling optimizations per bound specific type) across available devices. During solver execution, the host invokes device function kernels while managing device buffer synchronization. When argument types change, the solver must match the arguments for the corresponding device kernel program.

Solvers achieve high-level executability using the members in Table 8. Other useful solver functions are given in Table 10. The parts of the device program are given in Table 9.

Table 8: List of solver members.

Members	Description
Main Sequence	iterative loop
Preprocessor Solver	optional, once
Postprocessor Solver	optional, once

Table 9: Device Programs

Members	Description
Program_id	OpenCL handle
Source	string of OpenCL code defining function kernels to be compiled into a device program
Kernel	compiled device executable resulting from program source and bindings

Table 10: Other useful solver functions.

Members	Description
virtual bool build()	populate instruction sequence(s)
bool execute()	prepare and run this solver and pre/post sequence

4.12 System

A *system* is an extrinsic object (noun) — a scoped numerical domain with abstract or physical form and behavior, representing components, assemblies, or fields. Only a single type/kind of system exists in a computing sense — for all systems are human constructs; boundaries must be argued for most utility (plus, all variations can't be compiled into the software a-priori!). Generic systems are defined (and specialized) by specific runtime assignments ("state" — such as geometry, physics, boundary conditions, numerical solvers, and anything else that defines an abstract or physical system). A system can also supervene and manage any number of subsystems in a parent-child (uni-directional) relational tree hierarchy; spatial and temporal subspaces each have an assignable host and device resources, allowing a complex problem to be parallelized on SIMD hardware at more reasonable scales. Many of the members that define a system contain (or are)

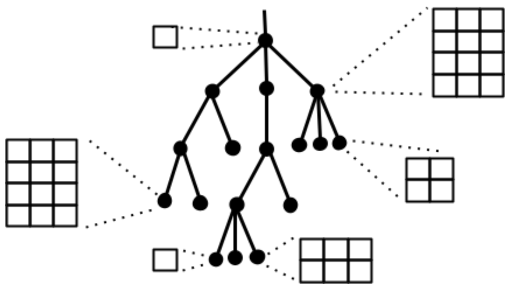

Figure 6: Systems are managed in a recursive tree, permitting complex ownership and information flow. Systems improve human and machine conceptualization of the problem domains. Each system can execute its own numerical process (typically via a solver), which can be dispatched to CPU or GPU resources.

algorithms which operate on systems and their members, and so, the definition of *something* is a (recursive) composite of declarative and procedural elements, permitting near-infinite complexity as a whole, or the desired level of fidelity as a human-machine representation of engineering concepts and processes.

5 System Members

This section provides an overview of the members that are defined within a system instance and how these high-level objects interact within and across systems. Generally, any given system may utilize the following attributes to implement an abstract or physical concept:

5.1 Data

Data are scalars and vectors for singular and nodal values stored in a convenient format accessed as *system.data(PK)*. In addition to numeric values, a given system's data defines its name, owner, and permissions; such attributes are intrinsic to data but can be utilized by parent objects. The dimensions of data entries are determined by the number of nodes in the system. If a geometry exists, the number of rows is equal to the number of nodes (or elements), while the number of columns is dictated by the number of components associated with the given PK and spatial dimensionality.

5.2 Geometry

Geometry is an optional pointer (address reference) to a local geometry, which is transformed to a global space instance using a model matrix. This permits a geometry to be shared between one or more systems and greatly reduces memory consumption with large numbers of duplicates or patterns. The global model matrix is computed as the recursive product of parent local matrices, providing linear complexity scaling for refresh starting at the root system and performing similar breadth-first recursive refresh to all subsystems. Geometries can be shared across systems and managed by the server application.

5.3 Physics

Physics is an optional pointer (address reference) to a physical model, which contributes algorithms to the System's solver in preparation for execution. As a model is defined to be a collection of algorithms to achieve an approximate result, physics is defined as the composition of models, itself a type of model. In the same way geometries can be patterned, physics can be referenced and managed at a higher level, eliminating duplication of setup steps. Most physical models utilize algorithms to define a system's spatial transport and state; similar method families may be combined.

5.4 Conditions

Conditions are a sequence of algorithms applied to temporal and spatial boundaries to provide numerical closure. Initial conditions are typically applied to systems as part of a preprocessor, while boundary conditions are applied to regions repeatedly within the main solver sequence. Conditions are bound to instructions and inserted into the sequence; upon building a solver the algorithms are consolidated into the proper execution sequence. Explicit methods often assert conditions on *system.data* entries, while implicit methods typically manipulate the *system.adjacency* matrix and/or state.

5.5 Links and Contacts

Links and contacts are a sequence of algorithms applied to couple abstract and spatial boundaries in lieu of conditions. Links and contacts implement an underlying duplexed coupling algorithm, permitting information to flow between boundaries in adjacent systems. Coupling is essentially a dynamic variation of static conditions, whereby directional flow control and different sampling approaches enable coupling to be customized to the specific configuration. Links are user-defined couplings, while contacts are automatically determined based on spatial proximity and overlap.

5.6 Materials

Materials are optional data that define the intrinsic physical properties of a given substance to be referenced in one or more regions or as defaults for the system. Materials are implemented as the mapping between assigned regions and corresponding data sets managed by global *Constants.materials*. Typically, material properties are applied to regions that have the same dimensionality of the system; a 2D domain applies material properties to surfaces, while a 3D domain applies material properties to volumes. If materials are not specified, the system reverts to defaults defined in *system.physics*. Explicit schemes tend to apply material properties to nodes, while implicit schemes tend to apply material properties to elements, though this tends to be dependent on the specific numerical method.

5.7 Solvers

Solvers are a list of numerical methods to be executed as part of a workflow. Typically, there is a single system per solver instance; solvers are bound and executed against their owning system. A solver usually defines a self-contained numerical process such as mesh generation, finite volume, or finite element methods using a main iterative sequence plus optional recursive preprocessor and postprocessor stages. A solver contains faculties to compile a master sequence, assemble OpenCL code fragments, and generate a device compute program. One or more solvers can be listed to define a system's workflow.

5.8 Subsystems

Subsystems are a container of child systems used to further resolve a numerical domain into constituent parts, constructing a supervening tree hierarchy to change the computational complexity of the problem to abstractions more suitable for human and machine interaction at each level of fidelity and numeric parallelism. A given subsystem has direct access to its children but does not have direct access to its parent; a one-way organization allows systems to be replicated and emplaced under new systems as desired.

6 Execution Modes

There are three known ways to define and execute an *xcompute* process: shell command line interpreter, Protobuf-generated I/O, and interactive server-client protocol. Execution of the process can be customized by one or more configuration files.

6.1 Configuration

To set default application runtime parameters (such as favored device, max iterations, convergence criteria, input/output preferences) each *xcompute* session can load a simple human-editable *.cfg file at launch and/or throughout runtime. Different applications (e.g. server vs client) have one or two configuration files that serve characteristic purposes. In all cases, the application expects at least one config file in the local execution or install path. If one is not provided, warnings are generated and defaults may lead to unexpected or undefined behavior. Configurations can be serialized using *Messages::Variables*.

Server-side config contains compute resource defaults and service parameters such as port numbers. Client-side config contains interface library locations, stylesheet locations, and user-facing defaults.

6.2 Shell Interpreter

A command line interface for XCOMPUTE is under development. While an interactive client facilitates more exploration of a problem, repetitive analysis of a problem is also required, where a simple scripted interface may be preferable to the interactive client for batched studies.

The shell interpreter would allow a problem to be expressed on the command line. Geometries would be inputted from external files; boundary conditions can then be applied to surfaces. The described system could then be submitted for execution.

Several external file formats are expected as potential inputs to the command:

- CSV for data and conditions
- VTU for external visualization tools
- MSH for gmsh import and export
- STL for discrete surface representation
- SDF for signed distance field (compressed)

6.3 Protobuf-Generated I/O

To share numerical information across computing sessions, Xplicit Computing created the *Messages*$^{\text{TM}}$ file and wire schema, based on Google's Protocol Buffer mechanism, known commonly as Protobuf. Its purpose is to allow engineers and scientists to efficiently and seamlessly share numerical computing data across computing platforms and programming languages. *Messages* language bindings provide an easy way to utilize existing workflow tools to generate case files for use in the *xcompute* environment.

Messages provides flexible encoding/decoding similar to XML and JSON but faster and denser. A machine-generated *Messages* library contains utilities to flatten and reconstruct object-oriented and vectorized data structures encountered in numerical simulation setup, expression, and results (e.g. systems engineering, CFD, FEA, EDA, and geometry processing).

Protobuf is a platform agnostic serialization mechanism for structured data. The *Messages* schema is defined by .proto files that are passed into the *protoc* Protobuf compiler to generate high-performance binary-encoded accessors for various languages. Xplicit Computing's applications are built on these generated bindings, and end-users can also apply those same standards to their own custom integration for an integrated workflow with high continuity and performance.

The *Messages* schema is defined by the .proto files listed in Table 11.

Table 11: XC Messages Schema and files

Filename	**Description**
vector.proto	XCO numeric data object arrays using packed arena allocation
spatial.proto	XCG geometry/topology of elements and regions discretization
concept.proto	XCS system case setup, models, parameters, associations
meta.proto	XCM metadata and user-graphics media for a specific system

The bindings supported by protoc include C++, Java, Python, and others. Additionally, Google and third-parties provide plug-ins to protoc for over 30 languages.

In the case of XCOMPUTE, *Messages* .proto files are compiled into C++ source code, which are compiled into object code modules, which are collected and archived into library *libxcmessages*. This library is linked with *libxccommon* and other *xcompute* object modules to produce the primary *xcompute-server* and *xcompute-client* executable applications.

To ease the process of adopting *Messages* into other projects, Xplicit Computing publishes a developer's guide "Introduction to Messages" [3] which provides C++ and Python code examples of how serialization and deserialization (parsing) are done, how values are assigned into *Messages* and how they are retrieved after deserialization. The source code for *Messages* is provided on GitHub under a BSD-3-Clause open source license so that users can easily incorporate *Messages* in their project [2].

The example below shows the C++ binding of *Messages*. The protoc-generated header file "vector.pb.h" is included. The name of the message container is set to a property-key. Values are interleaved in groups of three for a 3d domain.

```
#include "vector.pb.h"
Messages::Vector64 msg; // first, create an empty message container
msg.set_name("Position|Value"); // set the name field with a string
msg.set_components(3); // or however many vector dimensions, ie. 3 for xyz
msg.add_values(pos.x); // push back x value, e.g. from some glm::dvec
msg.add_values(pos.y); // push back y value
msg.add_values(pos.z); // push back z value
```

The same is achieved in Python by:

```
import vector_pb2 as vector
msg = vector.Vector64()                  # create an empty message container
msg.name = "Position|Value"
msg.components = 3
msg.values.append(pos.x)
msg.values.append(pos.y)
msg.values.append(pos.z)
```

The protoc-generated language bindings provide a path for integration with existing scientific and engineering software tools and scripts, and thus a means of augmenting existing workflows. Values could be been appended by way of a serial or parallel loop. Furthermore, higher level functions could be defined to simplify the code even further.

The Python code below describes a thermal model with two boundary conditions, and references *xcompute* internal machinery to run a finite element simulation. The modules "concept_pb2" and "vector_pb2" are part of the Python language binding generated by protoc (versions 2 and 3):

```
import concept_pb2 as concept
import vector_pb2 as vector

system = concept.System()        # initialize a system description

model = concept.Model()
model.name = "Thermal␣Diffusion␣Model"
system.models.append(model)

# add a temperature dirichlet condition to surface 1
system.conditions.append(fe_temperature(surface=1,
        propkeys={"Temperature": 300}))

# add a convection robin condition to surface 2
system.conditions.append(fe_convection(surface=2,
        propkeys={"Temperature|Reference": 1000, "HeatTransfer|Coefficient": 1.0}))
```

An additional module of about 50 lines of Python code are referenced to convert the boundary conditions into suitable *Messages* serialization. The result can be saved as a file and later read by the *xcompute-server* to prepare the simulation. Note that the serialization here is produced with Python, but the *xcompute-server* deserialization utilizes C++. This lays the groundwork for a series of cases to study the underlying problem of interest.

6.4 xc::io (Server-Client)

Numerical throughput is typically limited by available processing power, local working memory, or communication bandwidth between devices or hosts, and cache-coherency considerations. Although processing power continues to climb, transport between host and devices remains a primary expense moving towards heterogeneous architectures. In a large-scale distributed environment, this data locality impediment is exacerbated due to limited network bandwidth (as compared to local RAM or PCIe). In order to approach theoretical throughput within the hardware and compatibility across CAE software contexts, an effective systems-of-systems conceptualization (or decomposition) is required for efficient message construction and transport on top of an open schema.

In conjunction with the *Messages* library, the *xcompute* protocol provides for efficient sharing of structured numerical data between servers and clients or between servers and servers. Upon changes to a numerical system, a server application pushes a *Messages::Meta* manifest (defined in meta.proto) to each connected client as an outline of the numerical domain and data available to be requested for each system. Connected clients receive these meta messages into the respective metaobject's buffer, and if it doesn't exist, creates a new metaobject for a corresponding system's global unique id. Within the following client refresh cycle, each relevant metaobject iterates through its members comparing revision numbers against those in the meta-buffer, and as required, updating said members by calling blocking get/pull requests from the server by id. Client-side user events invoke any number of non-blocking do/command functions on the server to manipulate the simulation state machine, and affected systems push meta messages to post changes.

To minimize unintended exposure of work product and intellectual property, the governing numerical system setup is expressly not defined in meta messages, but rather in respective server-side messages defined in *Messages* Protobuf concept.proto and spatial.proto files. Attributes are requested and fulfilled a la carte from the meta manifest, and numerical data is mostly transmitted over networks for requested attributes as encoded arrays of single-precision floats. Administrators can specify users' permissions for import and export privileges to control project data. For each system, data attributes are managed by the property-key as *Messages::Vector32* or *Messages::Vector64* for accelerated I/O and flexible use. These vector types are defined in a fourth vector.proto optimized for packed arena allocation of array-like attributes.

6.4.1 XCOMPUTE transport protocol (XCTP)

To facilitate network communication, the *xcompute* transport protocol (XCTP) over TCP/IP is introduced, with the service name 'xcompute' on IANA-reserved port 11235 [11]. Either server- or client-side, a message is either ingress or egress, and contains the necessary metadata to describe its payload. The message payload may be a Protobuf message or a data primitive (e.g., *std::string*, *int*, and IEEE-754 float and double).

XCTP version 1.0 has two layers; first, a header describing each message's payload, and second, a header providing a description of one or more concatenated messages in a packet, as shown below.

- Message: [Message Header] {payload}
- Packet: [Packet Header] {Message1, ..., MessageN}

A Message Header comprises a quintuple of:

Table 12: Message Header

Field	Description
type:	enumeration describing the payload type, such as a function name, whether is it an actionable payload, or the data-type
length:	byte-length of the message
container-type:	type of container as an enumeration, such as fundamental (eg. built-in type), aggregate (eg. vector, set, protobuf data-type, etc), or null
container-element:	unique container-element index, nullable
container-size:	aggregate element count

A Packet Header comprises an septuple of:

Table 13: Packet Header

Field	Description
packet-length:	cumulative byte-length of the header and message(s)
protocol-version:	version of the XCTP transport being relayed
session-id:	monotonically increasing identifier describing the connection state
sequence-nbr:	monotonically increasing unique identifier of the packet
message-count:	the count of messages in the packet-payload
date-time:	UTC send time of the packet out the wire
thd-id:	thread-id identifying the issuing thread of this packet (only relevant client-side)

Combined, these fields given in Table 12 and 13 make for a self-describing protocol fit for TCP and UDP multicast communication.

7 Conclusion

In this paper, we outlined XCOMPUTE, a unified framework that utilizes numerical algorithms to solve multiphysics problems across heterogeneous, distributed computing resources. We presented the architecture, definitions, objects, code, and execution modes incorporated within XCOMPUTE, including concepts, logic, and examples. These modular building blocks provide a foundation for the design, development and improvement of next-generation integrated CAE systems.

We also released the *Messages* file & wire schema as an open-source tool to allow efficient sharing of numerical computing data across platforms and programming languages. We showed how existing workflows can use *Messages* to generate case files for the *xcompute* environment.

Queries about XCOMPUTE software availability and license terms can be sent to:
`info@xplicitcomputing.com`

Acknowledgements

This is paper benefited greatly from comments and critical review by other members of the Xplicit Computing team. In particular, our thanks go to F. Oz, D. Indictor, S. Suresh, and J. J. Maxwell.

Appendix A

Properties (partial list)

AnyProperty, Position, Angle, Area, Length, SignedDistance, Volume, Curvature, Element, Feature, Quality, Acceleration, Displacement, Mass, Moment, Momentum, Traction, Velocity, Energy, Pressure, Temperature, Enthalpy, Entropy, CFL, Cp, Cv, Gamma, SpeedOfSound, Time, Stress, HeatTransfer, Poisson, Modulus, SmallStrain, Conductivity, Expansion, YoungsModulus, Distribution, Probability, Diffusion, Viscosity, Vorticity, Voltage, Charge, Permittivity, Permeability, Emissivity, Absorptivity, Aluminum, Beryllium, Boron, Carbon, Electron, Fluorine, Helium, Hydrogen, Lithium, Magnesium, Neon, Neutron, Nitrogen, Nitric, Nitrous, Oxygen, Sodium, Copper...

Modifiers (partial list)

AnyModifier, Coefficient, Count, Density, Net, Size, Approx, Exact, Limit, Residual, Scratch, Equilibrium, Ratio, Reference, Standard, Total, Stagnation, Critical, Static, Dynamic, Molar, Turbulent, Electrical, Thermal, Adjacency, Assembly, Normal, Tangent, Filtered, Gaussian, Magnitude, Max, Mean, Median, Min, Mode, RMS, Standard, Deviation, Sum, Curl, Difference, Divergence, DT, DX, DY, DZ, Flux, Gradient, Integral, Shear, Compressive, Tensile, Torsional, Speed, Anion, Carbide, Cation, Chloride, Diatom, Dioxide, Fluoride, Hydride, Hydroxide, Monoxide, Nitride, Oxide...

Appendix B

Emergence of the Signed Distance Field (SDF)

Given some computational domain, to solve the SDF we compute and store the radial distance to boundary $\delta\Omega$ at discrete positions x[n] to minimize scalar field $\phi[n] = min||x(\delta\Omega) - x[n]||$ which intrinsically has slope magnitude of $1 = |\Delta\phi|$ with gradient defined to be outward normal to the implicit boundary [12, 13]. SDF initialization complexity for F faces is approximately proportional to N$_{shell}$ $log(F)$, so to save on computational costs, the hyperbolic characteristics of the eikonal equation permits the boundary SDF narrowband of N$_{shell}$ nodes to be solved exactly and extended approximately using fast-marching or equivalent. Extension typically proceeds two orders of magnitude faster than initialization.

The SDF field can be sampled for spatial-physics wall functions, gradients, curvature, and other SDF-derived differential geometry quantities [12]. It can be used to generate unstructured meshes or structured grids, each requiring special procedures to properly interpret elements in regions in close proximity to boundaries; resulting elements near or on boundaries of computable grids and meshes will be erroneously deleted (or marked invalid) preventing the ability to apply conditions. Such subtleties span the gap between academic and working capabilities on the topic.

Although it is possible to solve SDF on meshes, spatial sampling against unstructured geometries incurs complexity that is uncompetitive with constant time sampling on structured grids. The declarative nature of unstructured element topologies requires 20-50 times more memory than a procedural structured grid [1]. Therefore, within the scope of *xcompute* we assume that sampling occurs against grids to minimize memory footprint and accelerate computations. In practice, these background grids might not be useful to end-users, so the application should show or hide such support geometries as fit.

Creating a structured grid requires a few parameters: nodal resolution, spatial domain extents, and whether or not to expose regions to users (which defaults to false for background grids to save resources). A grid can be constructed with a smooth metric grading, approximating a desired total node count. Conversely for structured grids, it is more useful to explicitly specify the I-J-K dimensions to control anisotropicity. Given a structured grid, any other geometry can easily and efficiently sample the background to first-order accuracy in constant time by quantizing the position into a I-J-K location code and computing the node-element weightings from the remainder. Since the SDF is typically linearly-varying, the sampling matches accuracy of the underlying field and returns accurate interpolations of SDF values, which is critical near zero where the boundary is defined by default.

For SDF to be useful to existing workflows, a method exists to convert explicit topologies (such as from an STL file) into implicit fields. The numerical complexity to initialize SDF around discrete surfaces restricts naive implementations to small problems [9, 13]. To scale to millions of elements, the collection of simplexes surrounding each reference face must be identified and then trimmed into a narrowband shell. Each node must then search for the minimum distance to the closest face using an octree to approximate far faces. This proceeds quickly in parallel implementations, but robustly determining the SDF sign (for inside-outside) requires an expensive winding number calculation; steradians are accumulated for each narrowband node in a parallel stack queue, dominating the SDF initialization wall time (and thus the interest in reusing SDF shells). Once the SDF value and sign are correctly determined for the narrowband, the SDF is said to have been initialized and can be extended to the majority of nodes in the grid using a fast-marching method.

A rich variety of operations are available for implicit SDF shapes, including: sampling, boolean operations, blending, contouring, thinning & thickening. Some of these procedures are required while preparing meshes and grids. The intrinsic properties of SDF and its derivatives allow spatial-physical algorithms to make better informed decisions and projections. Given any position on an SDF field and its gradient, one can estimate the vector to the closest surface in a few clock cycles in constant-time within first-order accuracy. Per-Olof Persson's work briefly outlines general procedures to utilize SDF in conjunction with other field data to directly optimize shapes against physics [12, 13].

The SDF narrowband and hyperbolic solution can be exploited with index-based compression schemes; datagram size is proportional to the number of unique narrowband values while extension values are discarded, ignored, or assigned a fiducial value. Simple shapes have extreme compression ratios, while the compression size for complex shapes are proportional to the product of topological dimensionality, relative surface area, and resolution. Tolerance handling is also important; information quantizing and loss are not permitted in the SDF datagram floating-point data. Once an SDF datagram has been received, it is uncompressed, evaluated into a structured grid, and expanded into a full SDF field for use.

References

[1] G. J. Orr. Unified Geometries for Dynamic HPC Modeling, 2018.

[2] Xplicit Computing. Messages. `https://github.com/XplicitComputing/messages`, 2022.

[3] Xplicit Computing. Introduction to Messages. `https://github.com/XplicitComputing/messages/blob/master/doc/xcmessages.pdf`, 2022.

[4] Top500. Top #1 Systems. `https://top500.org/resources/top-systems`, 2022.

[5] Tech-Clarity. Tech-Clarity Perspective: Reducing Non-Value Added Work in Engineering. `https://tech-clarity.com/documents/Tech-Clarity-Perspective-DDAa.pdf`, 2014.

[6] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. 8.4.2 error cost escalation through the project life cycle. In *INCOSE International Symposium*, volume 14, pages 1723–1737. Wiley Online Library, 2004.

[7] Harald Hanche-Olsen. Buckingham's pi-theorem. *NTNU*, 2004.

[8] Eigen project. The Matrix class. `https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html`.

[9] William Dawes, Simon Harvey, Simon Fellows, Neil Eccles, D Jaeggi, and Will Kellar. A practical demonstration of scalable, parallel mesh generation. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 981, 2009.

[10] Google Inc. Protocol Buffers 3. `https://developers.google.com/protocol-buffers`.

[11] IANA. Xcompute service name and port assignment. `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=11235`.

[12] Per-Olof Persson. The level set method. *Lecture notes, MIT*, 16:1–7, 2005.

[13] Per-Olof Persson. *Mesh generation for implicit geometries*. PhD thesis, Massachusetts Institute of Technology, 2005.