The ∇ Language Specification

Camier Jean-Sylvain - CEA, DAM, DIF, F-91297 Arpajon, France

September 25, 2015

Keywords: HPC, Numerical Programming Language, Software Productivity, Performance Portability.

Overview: This document is a preliminary design specification of the ∇ language: a numerical-analysis specific language which provides a new approach for integrating multi-physics large-scale scientific applications.

Warning: Several features may still evolve in the future, meaning that their syntax and semantics are not yet totally determinated.

Contents

1 Introduction

Nabla (∇) is an open-source [1] Domain Specific Language (DSL) introduced in [2] whose purpose is to translate numerical analysis algorithmic sources in order to generate optimized code for different runtimes and architectures. The objectives and the associated roadmap have been motivated since the beginning of the project with the goal to provide a programming model that would allow:

- **Performances**. The computer scientist should be able to instantiate efficiently the right programming model for different software and hardware stacks.
- **Portability**. The language should provide portable scientific applications across existing and fore-coming architectures.
- **Programmability**. The description of a numerical scheme should be simplified and attractive enough for tomorrow's software engineers.
- **Interoperability**. The source-to-source process should allow interaction and modularity with existing legacy codes.

As computer scientists are continuously asked for optimizations, flexibility is now mandatory to be able to look for better concurrency, vectorization and data-access efficiency. The ∇ language constitutes a proposition for numerical mesh-based operations, designed to help applications to reach these listed goals. It raises the level of abstraction, following a bottom-up compositional approach that provides a methodology to co-design between applications and underlying software layers for existing middleware or heterogeneous execution models. It introduces an alternative way, to go further than the bulk-synchronous way of programming, by introducing logical time partial-ordering and bringing an additional dimension of parallelism to the high-performance computing community.

This document releases the language specification corresponding to the preliminary version of its implementation. This document is organized as follows. An overview of the ∇ language features is given in chapter 2: data management and program flow are exposed, definitions and vocabulary are specified by the way. Chapter 3 presents the ∇ language specification. Finally, a commented ∇ port of LULESH[4] is provided in appendix.

This document applies to ∇ applications developers, and some prerequisites are necessary for full understanding: a good mastery of the C language and its standard [3], as well as good knowledge of syntactical grammar notations for programming languages.

2 Language Overview & Definitions

∇ allows the conception of multi-physics applications, according to a logical time-triggered approach. It is a domain specific language which embeds the C language in accordance with its standard [3]. It adds specific syntax to implement further concepts of parallelism and follows a source-to-source approach: from ∇ source files to C, C++ or CUDA output ones. The method is based on different concepts: no central *main* function, a multi-tasks based parallelism model and a hierarchical logical time-triggered scheduling. It adds specific syntax to implement further concepts of parallelism.

To develop a ∇ application, several source files must be created containing *standard functions* and specific *for-loop* function, called *jobs*. These files are provided to the compiler and will be merged to compose the application. The compilation stages operate the transformations and return *source files*, containing the whole code and the required data. An additional stage of compilation with standard tools must therefore be done on this output.

A ∇ program lexically consists of white space (ASCII space, horizontal tab, form feed and line terminators), comments, identifiers, keywords, literals, separators and operators, all of them composed of unicode characters in the UTF-8 encoding. The language does not specify any limits for line length, statement length, or program size. A ∇ program grammatically consists of a sequence of statements, declarations, which are connected to the explicit definitions of: *items*, *functions*, *jobs* and ∇ *variables*. Appendix A gives illustrative examples of such a listing.

2.1 Lexical & Grammatical Elements

To be able to produce an application from ∇ source files, a first explicit declaration part is required. Language *libraries* have to be listed, *options* and the data fields -or *variables*- needed by the application have to be declared. The *options* keyword allows developers to provide different optional inputs to the application, with their default values, that will be then accessible from the command line or within some data input files. Application data fields must be declared by the developer: these *variables* live on *items*, which are some mesh-based numerical elements: the *cells*, the *nodes*, the *faces* or the *particles*.

```
nodes{                          cells{
   ℝ^3 ∂tx;                        ℝ p;
   ℝ^3 ∂t2x;                       ℝ^3 ε;
   ℝ^3 nForce;                     ℝ^3 cForce[nodes];
   ℝ nMass;                        ℝ delv_xi;
};                              };
```

Listing 1: Variables Declaration in ∇

Listing 1 shows two declarations of variables living on *nodes* and *cells*. Velocity (∂tx), acceleration (∂t2x) and force vector (`nForce`), as well as the nodal mass (`nMass`) for *nodes*. Pressure (`p`), diagonal terms of deviatoric strain (ε) and some velocity gradient (`delv_xi`) on *cells*.

2.2 Functions and Jobs Declaration

Two kinds of functions live within a ∇ program. *Functions* are standard functions (as in C) that have only access to *global* variables. *Jobs* are functions that eventually take in input *variables*, eventually produce output *variables* and have also access to *global* variables. Jobs are tied to an *item* (cells, nodes, faces or particles), they implicitely represent a **for-loop** on these ones.

For example, listing 2, from the listing 46 is a *for-loop*, iterating on the *nodes*, set by the developer to be triggered at the logical time '-6.9'. This job uses in its body the '∀' token, which starts another for-loop, for each *cell* the current node is connected to. Listings 3 and 4 are the C and CUDA generated sources.

```
nodes void iniNodalMass(void)
  in (cell calc_volume)
  out (node nodalMass) @ -6.9{
  nodalMass=0.0;
  ∀ cell nodalMass += calc_volume/8.0;
}
```

Listing 2: ∇ Job Declaration, a *for-loop* on *nodes*

```
static inline void iniNodalMass(){
   dbgFuncIn();
   _Pragma("omp parallel for firstprivate(NABLA_NB_CELLS,NABLA_NB_CELLS_WARP,NABLA_NB_NODES)")
   for(int n=0;n<NABLA_NB_NODES_WARP;n+=1){
      node_nodalMass[n]=0.0 ;
      FOR_EACH_NODE_WARP_CELL(c){
         int nw;
         real gathered_cell_calc_volume;
         nw=(n<<WARP_BIT);
         gatherFromNode_k(node_cell[8*(nw+0)+c],
                  node_cell[8*(nw+1)+c],
                  node_cell[8*(nw+2)+c],
                  node_cell[8*(nw+3)+c],
                  cell_calc_volume,
                  &gathered_cell_calc_volume);
         node_nodalMass[n]+=opDiv(gathered_cell_calc_volume, 8.0);
      }
   }
}
```

Listing 3: C generated job, from the *for-loop* of listing 2

```
__global__ void iniNodalMass(int *node_cell,
                             real *cell_calc_volume,
                             real *node_nodalMass){
  const register int tnid = blockDim.x*blockIdx.x+threadIdx.x;
  if (tnid>=NABLA_NB_NODES) return;
  node_nodalMass[tnid]=0.0;
  for(int i=0;i<8;i+=1){
    real gathered_cell_calc_volume=real(0.0);
    gatherFromNode_k(node_cell[8*tnid+i],
                     cell_calc_volume,
                     &gathered_cell_calc_volume);
    node_nodalMass[tnid]+=opDiv(gathered_cell_calc_volume,8.0);
  }
}
```

Listing 4: CUDA generated job, from the *for-loop* of listing 2

Both *functions* and *jobs* can be triggered with new statements: the '@' statements. As soon as they are coupled to this logical time statement, both *functions* and *jobs* do not take standard parameters and return types anymore but are declared to work on *variables*.

2.3 Program Flow

The execution of a ∇ program does not start at the beginning of the program but is driven by the '@' statements. They ensure the declaration of logical time steps that trigger the statement they are related to. The different '@' attributes are gathered and combined hierarchically in order to create the logical time triggered execution graph, used for the scheduling. Different jobs and functions can be declared in multiple files and then be given to the ∇ compiler. Different stages of compilation will take place, one of the most important is the one that gathers all of these '@' statements and produces their execution graph used during code generation.

The introduction of the hierarchical logical time within the high-performance computing scientific community represents an innovation that addresses the major exascale challenges. This new dimension to parallelism is explicitly expressed to go beyond the classical single-program-multiple-data or bulk-synchronous-parallel programming models. The task-based parallelism of the ∇ jobs is explicitly declared via logical-timestamps attributes: each function or job can be tagged with an additional '@' statement. The two types of concurrency models are used: the control-driven one comes from these logical-timestamps, the data-driven model is deduced from the *in*, *out* or *inout* attributes of the variables declaration. These control and data concurrency models are then combined consistently to achieve statically analyzable transformations and efficient code generation.

By gathering all the '@' statements, the ∇ compiler constructs the set of partially ordered jobs and functions. By convention, the negative logical timestamps represent the initialization phase, while the positive ones compose the compute loop. You end up with an execution graph for a single ∇ component. Each ∇ component can be written and tested individually. A nested composition of such logical-timed components becomes a multi-physic application. Such an application still consists in a top initialization phase and a global computational loop, where different levels of ∇ components can be instantiated hierarchically, each of them running there own initialization/compute/until-exit parts.

3 Language Specification

3.1 Syntax Notation

Syntax and semantics are given for terminals and nonterminals that differ with the C language and its standard [3]. In the syntax notation for the grammatical elements used in this document, a colon following a nonterminal introduces its definition.

3.2 Lexical Elements

3.2.1 Keywords

Syntax

```
aligned auto
break
char const continue
do double
else extern
float for
if inline int
long
register restrict return
short signed sizeof static
unsigned
void volatile
while
```

Listing 5: C Standard Keywords

```
all
Bool backCell
cell Cell cells coord
face Face faces forall foreach frontCell
global
in inner inout Integer
node Node nodes
options out outer own
particle Particle particles
Real Real3 Real3x3
this
Uid
with
```

Listing 6: ∇ Additionnal Keywords

Semantics The above tokens are case sensitive and are reserved for use as keywords, and shall not be used otherwise.

3.2.2 Literals

Syntax

```
L                    [a–zA–Z_αβγδεζηθικλμνξοπρρστυφχψωΑΒΓΔΑΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ∂]
```

Listing 7: ∇ Literals

Semantics A literal is a sequence of nondigit characters, including the underscore '_'. Lowercase and uppercase letters are distinct. Literals are used to compose an identifier. The additional Greek letters are the following:

1. Lowercase Greek Letters

Character	Letter	Unicode	Character	Letter	Unicode
α	alpha	03B1	ν	nu	03BD
β	beta	03B2	ξ	xi	03BE
γ	gamma	03B3	*ο*	omicron	03BF
δ	delta	03B4	π	pi	03C0
ε	epsilon	03F5	ρ	rho	03C1
ζ	zeta	03B6	σ	sigma	03C3
η	eta	03B7	τ	tau	03C4
θ	theta	03B8	υ	upsilon	03C5
ι	iota	03B9	ϕ	phi	03D5
κ	kappa	03BA	χ	chi	03C7
λ	lambda	03BB	ψ	psi	03C8
μ	mu	03BC	ω	omega	03C9

2. Uppercase Greek Letters

Character	Letter	Unicode	Character	Letter	Unicode
A	Alpha	0391	N	Nu	039D
B	Beta	0392	Ξ	Xi	039E
Γ	Gamma	0393	O	Omicron	039F
Δ	Delta	0394	Π	Pi	03A0
E	Epsilon	0395	P	Rho	03A1
Z	Zeta	0396	Σ	Sigma	03A3
H	Eta	0397	T	Tau	03A4
Θ	Theta	0398	ϒ	Upsilon	03A5
I	Iota	0399	Φ	Phi	03A6
K	Kappa	039A	X	Chi	03A7
Λ	Lambda	039B	Ψ	Psi	03A8
M	Mu	039C	Ω	Omega	03A9

3. Other Letters

Character	Letter	Unicode
∂	Partial	2202

3.2.3 Digits

Syntax

```
D                   [0-9]                       // Digit
H                   [a-fA-F0-9]                 // Hexadecimal
E                   [Ee][+-]?{D}+               // Exponent
FS                  (f|F|l|L)                   // Floating point suffix
IS                  (u|U|l|L)*                  // Integer suffix
ℍ                   0[x]{H}+{IS}?               // Hexadecimal digit
ℤ                   {D}+{IS}?                   // Integer digit
ℝ                   {D}+{E}{FS}?                // Real digit
ℝ                   {D}*"."{D}+({E})?{FS}?      // Real digit
ℝ                   {D}+"."{D}*({E})?{FS}?      // Real digit
```

Listing 8: ∇ Digits

Semantics Each digit is associated to a unique type. Digits can also be used to create an identifier. In ∇ the digits are the same as in C (see [3]).

3.2.4 Identifiers

Syntax

```
IDENTIFIER          {L}({L}|{D})*
LC                  L?'(\\.|[^\'])+'            // Long-wide character
```

Listing 9: ∇ Identifiers

Semantics An identifier is a sequence of literals and digits. Again, lowercase and uppercase letters are distinct.

3.2.5 Strings

Syntax

```
STRING_LITERAL      L?\"(\\.|[^\\\"])*\"
```

Listing 10: ∇ Strings

Semantics In ∇ the strings are the same as in C (see [3]).

3.2.6 Punctuators

Syntax

```
[ ] ( ) { } . –>
++ — & * + – ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; , ...
= *= /= %= += –= <<= >>= &= ^= |=
<?= >?= ?=
# @ ∀ ∄ ∧ ∨
∞ ² ³ √ ∛ ½ ⅓ ¼ ⅛
· × × ⊗ ⨂ ⋆
```

Listing 11: ∇ Punctuators

Semantics Depending on context, punctuators may specify an operation to be performed. The first five lines are the same as in C (see [3]), the rest is ∇ specific.

3.2.7 Whitespaces

Syntax

```
[ \t\v\f]                    // ignored terminals
```

Listing 12: ∇ Ignored Whitespaces Terminals

Semantics ASCII space, horizontal tab, form feed and line terminators constitute whitespaces.

3.2.8 Comments

Syntax

```
/*                           // ignored comments bloc
//                           // ignored comment line
```

Listing 13: ∇ Ignored Comments

Semantics All text included within the ASCII characters '**/***' and '***/**' is considered as comment and ignored. Nested comments are not allowed. All text from the ASCII characters '**//**' to the end of line is considered as comment and is also ignored.

3.3 Grammatical Elements

3.3.1 ∇ Grammar

Syntax

```
∇_grammar
: with_library                  // Library declaration
| declaration                   // Preprocessing, Includes
| ∇_options_definition          // ∇ options definition
| ∇_item_definition             // Cell, Node, Face & Particle variables definition
| function_definition           // C function definition
| ∇_job_definition              // ∇ job definition
| ∇_reduction                   // ∇ reduction job definition
;
```

Listing 14: ∇ Rule Grammar

Semantics The input stream can be recursively one of the following: library declaration with the '`with`' keyword, standard includes or preprocessing declarations or specific ∇ declarations: options, variables, function or jobs.

3.3.2 Data Types

Syntax

```
type_specifier
: void
| char
| short
| int
| long
| float
| double
| signed
| Bool
| ℝ³ | Real3
| Real3x3
| ℝ | Real
| ℕ | unsigned
| ℤ | Integer
| Cell | Face | Node | Particle
| Uid
;
```

Listing 15: ∇ Grammatical Data Types

Semantics All the data types and type definitions existing in C can be used. ℝ, ℕ and ℤ are aliases to `float/double`, `unsigned int` and `int` respectively. `Uid` is still experimental.

3.3.3 Scopes

Syntax

```
start_scope: '{' ;          // Common starting scope
end_scope
: '}'                       // Standard ending scope
| '}' @ at_constant         // ∇ ending scope with an '@' statement
;
```

Listing 16: ∇ Rule Grammatical Scopes

Semantics Scopes are opened as in C, ∇ adds a possibility to explicitly declare the logical time-step for it with the '`@`'.

3.3.4 Operators

In ∇ the operators are evaluated as in C (see [3]).

New operators coming from new punctuators are still experimental.

3.3.5 Declarations

Library Libraries are introduced by the `with` token: additional keywords are then accessible, as well as some specific programming interfaces. For example, the `aleph` (ℵ) library provides the `matrix` keyword, as well as standard algebra functions to fill linear systems and solve them. An example is given in listing 39.

```
single_library:
| PARTICLES                 // Additionnal 'particle' keyword
| LIB_ALEPH                 // ℵ keywords for implicit schemes
| LIB_SLURM                 // time, limit & remain keywords
| LIB_MATHEMATICA           // Mathematica keywords
;
with_library_list
: single_library
| with_library_list ',' single_library
;
with_library: with with_library_list ';';
```

Listing 17: ∇ Rule library

Options

```
∇_options_definition
: options '{' ∇_option_declaration_list '}' ';'
;
∇_option_declaration_list
: ∇_option_declaration
| ∇_option_declaration_list ∇_option_declaration
;
∇_option_declaration
: type_specifier direct_declarator ';'
| type_specifier direct_declarator '=' expression ';'
| preproc
;
```

Listing 18: ∇ Rule Options Definition

Option variables are used as *constants* in the program. These constants must have a default value and should be changeable from the command line or input config file. An example is given in listing 40.

Item

```
∇_item
: cell
| node
| face
| particle
;
```

Listing 19: ∇ Rule Item Definition

∇_item is used to define the `in` and `out` ∇ variables of each *job*.

Items

```
∇_items
: cells
| nodes
| faces
| global
| particles
;
```

Listing 20: ∇ Rule Items Definition

∇_items are used to define the implicit data each *job* will loop on.

Items Scope

```
∇_scope: own | all;
```

Listing 21: ∇ Rule ∇ Scope

∇_scope are used to define the scope of the implicit data each *job* will loop on.

Items Region

```
∇_region: inner | outer;
```

Listing 22: ∇ Rule Items Region

∇_region are used to define the region on which each *job* will loop on.

Family

```
∇_family
: ∇_items
| ∇_scope ∇_items
| ∇_region ∇_items
| ∇_scope ∇_region ∇_items
;
```

Listing 23: ∇ Rule Family

The ∇_family allow the different combinations of items, scopes and regions. It is used to declare ∇ *jobs* by defining on which *items* they will loop on.

System

```
∇_system
: uid | this | iteration
| nbNode | nbCell
| backCell | frontCell
| nextCell | prevCell
| nextNode | prevNode
| time remain limit
| exit | mail
;
```

Listing 24: ∇ System Keywords Rule

∇_system are additional keywords allowed within ∇ *jobs*:

- `uid` or `this` used within a *job* body refers to the item of the current for-loop.
- `nbNode` or `nbCell` returns the number of items for the one considered.
- `next*` and `prev*` are for example accessible after `cartesian` library declaration.
- `time`, `remain` and `limit` keywords are usable after the `slurm` library declaration.
- `exit` allows to quit current time-line or the program if no higher hierarchical logical-time level exists.

Variable

```
∇_item_definition
: ∇_items '{' ∇_item_declaration_list '}' ';'
;
∇_item_declaration_list
: ∇_item_declaration
| ∇_item_declaration_list ∇_item_declaration
;
∇_item_declaration
: type_name ∇_direct_declarator ';'
| preproc
;
∇_direct_declarator
: IDENTIFIER
| IDENTIFIER '[' ∇_items ']'
| IDENTIFIER '[' primary_expression ']';
```

Listing 25: ∇ Rule Variable Definition

Variables must be declared before *jobs* definition. They are linked to ∇_items. Array of connectivities can be declared, however these arrays should be sized to fit mesh requirements. Examples are given in listings 41 and 42.

Parameters

```
∇_parameter_list
: ∇_parameter
| ∇_parameter_declaration
| ∇_parameter_list ∇_parameter
| ∇_parameter_list ',' ∇_parameter_declaration
;
∇_parameter
: ∇_in_parameter_list
| ∇_out_parameter_list
| ∇_inout_parameter_list
;
∇_in_parameter_list: in '(' ∇_parameter_list ')';
∇_out_parameter_list: out '(' ∇_parameter_list ')';
∇_inout_parameter_list: inout '(' ∇_parameter_list ')' ;
```

Listing 26: ∇ Rule Parameters

Parameter lists declare the *variables* on which the current *job* is going to work with. `in`, `inout` or `out` variables must be declared in the parameter list before use.

Job

```
∇_prefix: ∇_family | ∀ ∇_family ;
∇_job_definition
: ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' compound_statement
| ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' ∇_param_list compound_statement
| ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' @ at_constant compound_statement
| ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' @ at_constant if '(' constant_expression ')'
    compound_statement
| ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' ∇_param_list @ at_constant
    compound_statement
| ∇_prefix decl_specifiers IDENTIFIER '(' param_type_list ')' ∇_param_list @ at_constant if '('
    constant_expression ')' compound_statement
;
```

Listing 27: ∇ Rule Job Definition

Each *job* is tied to the *item* with which it is declared, via the *prefix* and the *family*. A *job* can or cannot be triggered by an `@` statement. If such `@` statement is given, an additional `if` statement is again accessible to allow different treatments from `option` variables for example. The `if` condition is for now a `constant_expression` but this limitation should be removed in future version.

Function

```
function_definition
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator declaration_list @ at_constant compound_statement
| declaration_specifiers declarator compound_statement
| declaration_specifiers declarator @ at_constant compound_statement
;
```

Listing 28: ∇ Rule function Definition

Functions are declared as in C. Similarly to *job* declaration, a function can be trigged by `@` statements. Optionnal `if` are not yet possible, but should be possible in future version.

3.3.6 Expressions

Primary Expressions

```
primary_expression
: #
| ∞
| ∇_item
| ∇_system
| ℍ  | ℤ | ℝ
| ½ | ⅓ | ¼ | ⅛
| IDENTIFIER
| QUOTE_LITERAL
| STRING_LITERAL
| '(' expression ')'
;
```

Listing 29: ∇ Rule Primary Expressions

Last four lines are as in C: an identifier, a constant, a string literal and a parenthesized expression are primary expressions. The first expressions come with new ∇ ponctuators: ∇_system or ∇_item keywords, few mathematical constants and type declarations. The `#` terminal allows to retrieve the iteration count within a *job* body. It is still experimental: there is a possible confusion with nested `foreach` statements.

Postfix Expression

```
postfix_expression
: primary_expression
| postfix_expression FORALL_NODE_INDEX
| postfix_expression FORALL_CELL_INDEX
| postfix_expression FORALL_MTRL_INDEX
| postfix_expression '[' expression ']'
| REAL '(' ')'
| REAL '(' expression ')'
| REAL3 '(' ')'
| REAL3 '(' expression ')'
| REAL3x3 '(' ')'
| REAL3x3 '(' expression ')'
| postfix_expression '(' ')'
| FATAL '(' argument_expression_list ')'
| postfix_expression '(' argument_expression_list ')'
| postfix_expression '.' IDENTIFIER
| postfix_expression '.' ∇_item '(' ℤ ')'
| postfix_expression '.' ∇_system
| postfix_expression PTR_OP primary_expression
| postfix_expression INC_OP
| postfix_expression DEC_OP
| postfix_expression SUPERSCRIPT_DIGIT_TWO
| postfix_expression SUPERSCRIPT_DIGIT_THREE
| aleph_expression
;
```

Listing 30: ∇ Rule Postfix Expression

Postfix expressions have been augmented with several types constructors possibilities: this is a work in progress and will evolve soon in a future version. `SUPERSCRIPT_DIGIT_*` terminals are introduced here but should move to become operators.

Unary Expression

```
unary_expression
: postfix_expression
| '√'  unary_expression
| '∛'  unary_expression
| '++' unary_expression
| '--' unary_expression
| '&'  unary_expression
| unary_prefix_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')';
```

Listing 31: ∇ Rule Unary Expression

Unary expressions are like in C, with the possibility to add some prefix expressions. Two examples are given here with the additionnal $\sqrt{}$ and $\sqrt[3]{}$ ones. Several other operations should arrive with future version.

Multiplicative Expression

```
multiplicative_expression
: cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
| multiplicative_expression '×' cast_expression // for vectors cross product.
| multiplicative_expression '·' cast_expression // for vectors dot products.
| multiplicative_expression '⊗' cast_expression
| multiplicative_expression '*' cast_expression; // for the products matrices, and tensors.
```

Listing 32: ∇ Rule Multiplicative Expression

First multiplicative expressions are as in C. Unicode characters and expressions are still to be normalized and will evolve in future versions of the specifications.

Assignment Expression

```
assignment_expression
: conditional_expression
| unary_expression assignment_operator assignment_expression
| unary_expression assignment_operator logical_or_expression '?' expression
;
```

Listing 33: ∇ Rule Assignment Expression

Assignment expressions are as in C, with an additional construction: the '?' statement without a ':' has the same semantic as the '?:' but with the last expression ommitted, meaning here '*else unchanged*'.

Aleph Expressions

```
aleph_vector: rhs | lhs ;
aleph_expression
: aleph_vector
| ℵ aleph_vector reset
| ℵ solve
| ℵ aleph_vector newValue | addValue | setValue
| ℵ matrix addValue | ℵ matrix setValue
| ℵ lhs getValue | ℵ rhs getValue
;
```

Listing 34: ∇ Rule Aleph Expression

ℵ expressions are usable after the 'with ℵ' declaration. It is a minimal interface proposed to construct implicit schemes.

3.3.7 Statements

Expression Statement

```
expression_statement
: ';'
| expression ';'
| expression @ at_constant';'
;
```

Listing 35: ∇ Rule Expression Statement

Expressions are as in C. A null statement, consisting of just a semicolon, performs no operations. For each standard expression, an `@` statement can be adjoined to declare the logical time at which it must be triggered.

Iteration Statement

```
iteration_statement
: ∀ ∇_item statement
| ∀ ∇_item @ at_constant statement
| ∀ ∇_matenv statement
| ∀ ∇_matenv @ at_constant statement
| ∀ IDENTIFIER cell statement
| ∀ IDENTIFIER node statement
| ∀ IDENTIFIER face statement
| ∀ IDENTIFIER particle statement
| while '(' expression ')' statement
| do statement while '(' expression ')' ';'
| for '(' expression_statement expression_statement ')' statement
| for '(' expression_statement expression_statement expression ')' statement
| for '(' type_specifier expression_statement expression_statement ')' statement
| for '(' type_specifier expression_statement expression_statement expression ')' statement
;
```

Listing 36: ∇ Rule Iteration Statement

The ∀ terminals can be one of: ∀, `foreach` or `forall`. Last six lines are the iteration statements of C. First ones are for ∇ *jobs*, to nest deeper for-loop constructions.

Reduction Statement

```
∇_reduction
: ∀ ∇_items IDENTIFIER <?= IDENTIFIER  @ at_constant ';'
| ∀ ∇_items IDENTIFIER >?= IDENTIFIER  @ at_constant ';'
;
```

Listing 37: ∇ Rule Reduction

3.4 Logical-Time Elements

3.4.1 @ Definition

```
at_single_constant
: ℤ | ℝ
| '-' ℤ | '+' ℤ
| '-' ℝ | '+' ℝ
;
at_constant
: at_single_constant
| at_constant ',' at_single_constant
```

Listing 38: ∇ Rule @ constant

3.4.2 Time-line conventions

Conventionally, the timeline is decomposed as follow:

Time Range	Compute Phase
-∞	First time initialization
]-∞,-0.0[	Standard initialization
0.0	Initialization after a restart
]+0.0, +∞[	Compute loop
+∞	Exit time

A ∇ Port of the LULESH [4] proxy application

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) proxy application is being developed as part of the NNSA Advanced Scientific Computing (ASC) National Security Applications (NSapp) Co-design Project at Lawrence Livermore National Laboratory (LLNL). This benchmark solves one octant of the spherical Sedov blast wave problem using Lagrangian hydrodynamics for a single material, following the gamma law gas equation state model. The initial conditions include a point source of energy deposited at the origin. Symmetry boundary conditions are imposed on the `xy`, `xz`, and `yz` coordinate planes.

Equations are solved using a staggered mesh approximation with an explicit time stepping algorithm to advance the solution. Thermodynamic variables are approximated as piece-wise constant functions within each element and kinematic variables are defined at the element nodes. Each discrete Lagrange time step advances the solution variables from the current time t^n to their values at the new time t^{n+1}: starting with variables at mesh nodes, then element variables based on the new node variable values are updated. The implementation uses the Flanagan-Belytschko kinematic hourglass filter.

Most of the comments in green have been taken from the original implementation. ∇'s keywords are in blue.

```
with cartesian;
```

Listing 39: LULESH Library Declaration

```
// ****************************************************************************
// * Options
// ****************************************************************************
options{
  ℝ option_dtfixed            = -1.0e-7;  // fixed time increment
  ℝ option_δt_initial         = 1.0e-7;   // variable time increment
  ℝ option_δt_courant         = 1.0e+20;
  ℝ option_δt_hydro           = 1.0e+20;
  ℝ option_δt_mult_lower_b    = 1.1;
  ℝ option_δt_mult_upper_b    = 1.2;
  ℝ option_initial_energy     = 3.948746e+7;
  ℝ option_stoptime           = 1.0e-2;   // end time for simulation
  ℝ option_hgcoef             = 3.0;      // hourglass control
  ℝ option_qstop              = 1.0e+12;  // excessive q indicator
  ℝ option_monoq_max_slope    = 1.0;
  ℝ option_monoq_limiter_mult = 2.0;
  ℝ option_e_cut              = 1.0e-7;   // energy tolerance
  ℝ option_p_cut              = 1.0e-7;   // pressure tolerance
  ℝ option_q_cut              = 1.0e-7;   // q tolerance
  ℝ option_u_cut              = 1.0e-7;   // node velocity cut-off value
  ℝ option_qlc_monoq          = 0.5;      // linear term coef for q
  ℝ option_qqc_monoq          = 0.666666666666666666; // quadratic term coef for q
  ℝ option_qqc                = 2.0;
  ℝ option_eosvmax            = 1.0e+9;
  ℝ option_eosvmin            = 1.0e-9;
  ℝ option_pmin               = 0.0;      // pressure floor
  ℝ option_emin               = -1.0e+15; // energy floor
  ℝ option_dvovmax            = 0.1;      // maximum allowable volume change
  ℝ option_dtmax              = 1.0e-2;   // maximum allowable time increment
  Bool option_chaos             = false;
  ℝ option_chaos_seed         = 1.1234567890123456789;
  Integer option_max_iterations  = 32768;
};
```

Listing 40: LULESH Options Declaration

```
// ********************************************************************************
// * Node Variables
// ********************************************************************************
nodes{
  ℝ³ ∂x;            // Velocity vector
  ℝ³ ∂∂x;           // Acceleration vector
  ℝ³ nForce;        // Force vector
  ℝ nodalMass;      // Nodal mass
};
```

Listing 41: LULESH Nodes Variables Declaration

```
// ********************************************************************************
// * Element Variables
// ********************************************************************************
cells{
  ℝ p;              // pressure
  ℝ e;              // internal energy, (to synchronize)
  ℝ q;              // artificial viscosity
  ℝ v;              // relative volume
  ℝ calc_volume;    // instant relative volume
  ℝ vdov;           // relative volume change per volume
  ℝ delv;           // relative volume change
  ℝ volo;           // reference (initial) volume
  ℝ arealg;         // characteristic length
  ℝ³ ε;             // diagonal terms of deviatoric strain  dxx(),dyy(),dzz()
  ℝ ql;             // artificial viscosity linear term, (to synchronize)
  ℝ qq;             // artificial viscosity quadratic term, (to synchronize)
  ℝ³ cForce[nodes];
  // Temporaries ////////////////////////////////
  ℝ delv_xi;        // velocity gradient
  ℝ delv_eta;
  ℝ delv_zeta;
  ℝ delx_xi;        // coordinate gradient
  ℝ delx_eta;
  ℝ delx_zeta;
  ℝ phixi;
  ℝ phieta;
  ℝ phizeta;
  ℝ vnew;           // new relative volume
  ℝ elemMass;       // mass
  // EoS ////////////////////////////////////////
  ℝ e_old;
  ℝ delvc;
  ℝ p_old;
  ℝ q_old;
  ℝ compression;
  ℝ compHalfStep;
  ℝ work;
  ℝ p_new;
  ℝ e_new;
  ℝ q_new;
  ℝ bvc;
  ℝ pbvc;
  ℝ vnewc;
  ℝ pHalfStep;
  ℝ sound_speed;
  // Boundary Conditions Flags //////////////////////////////////////////////
  Integer elemBC;          // symmetry/free-surface flags for each elem face
  // Reductions /////////////////////////////
  ℝ δt_cell_hydro;
  ℝ δt_cell_courant;
};
```

Listing 42: LULESH Cells Variables Declaration

```
// ****************************************************************************
// * Global Variables
// ****************************************************************************
global{
  ℝ δt_courant;          // Courant time constraint
  ℝ δt_hydro;            // Hydro time constraint
};
```

Listing 43: LULESH Global Variables Declaration

```
// ****************************************************************************
// * Initialization Part @ ]−∞,−0.0[
// ****************************************************************************

// ****************************************************************************
// * ini
// ****************************************************************************
void ini(void) @ −10.0{
  δt=(option_chaos)?option_δt_initial*option_chaos_seed:0.0;
  δt_hydro=option_δt_hydro;
  δt_courant=option_δt_courant;
}

nodes void choasNodes(void) out (node coord) @ −12.0{
  coord *= (option_chaos)?option_chaos_seed:1.0;
}

// ****************************************************************************
// * Set up boundary condition information
// * Set up elemement connectivity information
// ****************************************************************************
#define XI_M          0x003
#define XI_M_SYMM     0x001
#define XI_M_FREE     0x002
#define XI_P          0x00C
#define XI_P_SYMM     0x004
#define XI_P_FREE     0x008
#define ETA_M         0x030
#define ETA_M_SYMM    0x010
#define ETA_M_FREE    0x020
#define ETA_P         0x0C0
#define ETA_P_SYMM    0x040
#define ETA_P_FREE    0x080
#define ZETA_M        0x300
#define ZETA_M_SYMM   0x100
#define ZETA_M_FREE   0x200
#define ZETA_P        0xC00
#define ZETA_P_SYMM   0x400
#define ZETA_P_FREE   0x800
cells void iniCellBC(void) in (node coord) out (cell elemBC) @ −9.5{
  const ℝ zero = 0.0;
  const ℝ maxBoundaryX = X_EDGE_TICK*X_EDGE_ELEMS;
  const ℝ maxBoundaryY = Y_EDGE_TICK*Y_EDGE_ELEMS;
  const ℝ maxBoundaryZ = Z_EDGE_TICK*Z_EDGE_ELEMS;
  elemBC=0; // clear BCs by default
  foreach node{
    elemBC |= (coord.x==zero)?XI_M_SYMM;
    elemBC |= (coord.y==zero)?ETA_M_SYMM;
    elemBC |= (coord.z==zero)?ZETA_M_SYMM;
    elemBC |= (coord.x==maxBoundaryX)?XI_P_FREE;
    elemBC |= (coord.y==maxBoundaryY)?ETA_P_FREE;
    elemBC |= (coord.z==maxBoundaryZ)?ZETA_P_FREE;
  }
}
```

Listing 44: LULESH Initialization part -∞ ⇒ -0.0

```
// ****************************************************************************
// * Cells initialization
// ****************************************************************************
∀ cells void iniCells(void) in (node coord)
  out (cell v, cell e, cell volo, cell elemMass, cell calc_volume,
       cell p, cell q, cell sound_speed) @ -8.0{
  ℝ³ X[8];
  const ℝ chaos = (((ℝ)uid)+1.0)*option_chaos_seed;
  v=1.0;
  ∀ node X[n]=coord;
  e=(option_chaos)?chaos:(uid==0)?option_initial_energy:0.0;
  sound_speed=p=q=(option_chaos)?chaos;
  volo=elemMass=calc_volume=computeElemVolume(X);
}
```

Listing 45: LULESH Cells Initialization

```
nodes void iniNodalMass(void) in (cell calc_volume) out (node nodalMass) @ -6.9{
  nodalMass=0.0;
  ∀ cell{
    nodalMass+=calc_volume/8.0;
  }
 }
```

Listing 46: LULESH Nodal Mass Initialization

```
// ****************************************************************************
// * Compute part @ ]+0,+∞[
// ****************************************************************************

// ****************************************************************************
// * timeIncrement
// * This routine computes the time increment δtn for the
// * current timestep loop iteration. We aim for a "target" value of t_final-tn
// * for δtn . However, the actual time increment is allowed to grow by a
// * certain prescribed amount from the value used in the previous step and is
// * subject to the constraints δt_Courant and δt_hydro described in Section 1.5.3.
// ****************************************************************************
void timeIncrement(void) @ 0.1 {
  const ℝ target_δt = option_stoptime - time;
  const ℝ max_δt = 1.0e+20;
  const ℝ new_δt_courant = (δt_courant < max_δt)?½*δt_courant:max_δt;
  const ℝ new_δt_courant_hydro = (δt_hydro < new_δt_courant)?δt_hydro*2.0/3.0:new_δt_courant;
  const ℝ now_δt = new_δt_courant_hydro ;
  const ℝ old_δt = (GlobalIteration==1)?option_δt_initial:δt;
  const ℝ ratio = now_δt / old_δt ;
  const ℝ up_new_δt = (ratio >= 1.0)?(ratio < option_δt_mult_lower_b)?old_δt:now_δt:now_δt;//option_δ
      t_mult_lower_b
  const ℝ dw_new_δt = (ratio >= 1.0)?(ratio > option_δt_mult_upper_b)?old_δt*option_δt_mult_upper_b:
      up_new_δt:up_new_δt;
  const ℝ new_δt = (dw_new_δt > option_dtmax)?option_dtmax:dw_new_δt;
  const ℝ δτ = (option_dtfixed <= 0.0)?(GlobalIteration != 1)?new_δt:old_δt:old_δt;
  const ℝ scaled_target_δt = (target_δt>δτ)?((target_δt<(4.0*δτ/3.0))?2.0*δτ/3.0:target_δt):target_δt;
  const ℝ scaled_δt = (scaled_target_δt < δτ)?scaled_target_δt:δτ;
  δt = reduce(ReduceMin, scaled_δt);
  if (GlobalIteration >= option_max_iterations) exit;
}

// ****************************************************************************
// * Sum contributions to total stress tensor
// * pull in the stresses appropriate to the hydro integration
// * Initialize stress terms for each element. Recall that our assumption of
// * an inviscid isotropic stress tensor implies that the three principal
// * stress components are equal, and the shear stresses are zero.
// * Thus, we initialize the diagonal terms of the stress tensor to
// * -(p + q) in each element.
// ****************************************************************************
∀ cells void initStressTermsForElems(void)
  in (node coord, cell p, cell q)
  out (cell ε, cell cForce) @ 0.3 {
  const ℝ chaos = (((ℝ)0.0)+1.0)*option_chaos_seed;
```

```
  const ℝ sig = (option_chaos)?chaos:-p-q;
  ℝ³ fNormals,dj,x[8],B[8];
  ∀ node x[n] = coord;
  ε = dj = -1/4*((x[0]+x[1]+x[5]+x[4])-(x[3]+x[2]+x[6]+x[7]));
  calcElemShapeFunctionDerivatives(x,B);
  ∀ node B[n]=0.0;
  Σ_FaceNormal(B,0,1,2,3,x);
  Σ_FaceNormal(B,0,4,5,1,x);
  Σ_FaceNormal(B,1,5,6,2,x);
  Σ_FaceNormal(B,2,6,7,3,x);
  Σ_FaceNormal(B,3,7,4,0,x);
  Σ_FaceNormal(B,4,7,6,5,x);
  ∀ node cForce = -sig*B[n];
}

nodes void fetchCellNodeForce0(void) in (cell cForce) out (node nForce)@0.301 {
  ℝ³ Σ=0.0;
  ∀ cell{
    Σ+=cForce;
  }
  nForce=Σ;
}

// ****************************************************************************
// * calcFBHourglassForceForElems
// * Calculates the Flanagan-Belytschko anti-hourglass force
// * calcFBHourglassForceForElems
// ****************************************************************************
cells void ΣElemStressesToNodeForces(void)
  in (node coord, cell volo, cell v,
      cell sound_speed, cell elemMass,
      node ∂x)
  out (cell cForce) @ 1.3{
  const ℝ γ[4][8]={{ 1., 1.,-1.,-1.,-1.,-1., 1., 1.},
                   { 1.,-1.,-1., 1.,-1., 1., 1.,-1.},
                   { 1.,-1., 1.,-1., 1.,-1., 1.,-1.},
                   {-1., 1.,-1., 1., 1.,-1., 1.,-1.}};
  ℝ η0[4],η1[4],η2[4],η3[4] ;
  ℝ η4[4],η5[4],η6[4],η7[4];
  ℝ³ x[8],xd[8],dvd[8],η[8];
  const ℝ hourg=option_hgcoef;
  const ℝ τv = volo*v;
  const ℝ volume13=∛(τv);
  const ℝ θ = -hourg*0.01*sound_speed*elemMass/volume13;
  const ℝ determ = τv;
  ∀ node x[n] = coord;
  ∀ node xd[n] = ∂x;
  dvd[0]=∂Volume(x[1],x[2],x[3],x[4],x[5],x[7]);
  dvd[3]=∂Volume(x[0],x[1],x[2],x[7],x[4],x[6]);
  dvd[2]=∂Volume(x[3],x[0],x[1],x[6],x[7],x[5]);
  dvd[1]=∂Volume(x[2],x[3],x[0],x[5],x[6],x[4]);
  dvd[4]=∂Volume(x[7],x[6],x[5],x[0],x[3],x[1]);
  dvd[5]=∂Volume(x[4],x[7],x[6],x[1],x[0],x[2]);
  dvd[6]=∂Volume(x[5],x[4],x[7],x[2],x[1],x[3]);
  dvd[7]=∂Volume(x[6],x[5],x[4],x[3],x[2],x[0]);
  cHourglassModes(0,determ,dvd,γ,x,η0,η1,η2,η3,η4,η5,η6,η7);
  cHourglassModes(1,determ,dvd,γ,x,η0,η1,η2,η3,η4,η5,η6,η7);
  cHourglassModes(2,determ,dvd,γ,x,η0,η1,η2,η3,η4,η5,η6,η7);
  cHourglassModes(3,determ,dvd,γ,x,η0,η1,η2,η3,η4,η5,η6,η7);
   calcElemFBHourglassForce(xd,η0,η1,η2,η3,η4,η5,η6,η7,θ,η);
  ∀ node cForce = η[n];
}

nodes void ΣCellNodeForce(void) in (cell cForce) out (node nForce)@ 1.4 {
  ℝ³ Σ=0.0;
  foreach cell Σ+=cForce;
  nForce+=Σ;
}

// ****************************************************************************
// * The routine CalcAccelerationForNodes() calculates a three-dimensional
// * acceleration vector A at each mesh node from F.
// * The acceleration is computed using Newton's Second Law of Motion,
```

```
// * F = m0 A, where m0 is the mass at the node.
// * Note that since the mass in each element is constant in time for our calculations,
// * the mass at each node is also constant in time.
// * The nodal mass values are set during the problem set up.
// ****************************************************************************
nodes void ∂∂ForNodes(void)
  in (node nForce, node nodalMass) out (node ∂∂x) @ 2.8{
  ∂∂x = nForce/nodalMass;
}

// ****************************************************************************
// * The routine ApplyAccelerationBoundaryConditions() applies symmetry boundary
// * conditions at nodes on the boundaries of the mesh where these were specified
// * during problem set up. A symmetry boundary condition sets the normal
// * component of A at the boundary to zero.
// * This implies that the normal component of the velocity vector U will
// * remain constant in time.
// ****************************************************************************
outer nodes void ∂∂BCForNodes(void)
  out (node ∂∂x) @ 2.9 {
  ∂∂x.x=(coord.x==0.0)?0.0;
  ∂∂x.y=(coord.y==0.0)?0.0;
  ∂∂x.z=(coord.z==0.0)?0.0;
}

// ****************************************************************************
// * The routine CalcVelocityForNodes() integrates the acceleration at each node
// * to advance the velocity at the node to tn+1.
// * Note that this routine also applies a cut-off to each velocity vector value.
// * Specifically, if a value is below some prescribed value, that term is set to zero.
// * The reason for this cutoff is to prevent spurious mesh motion which may arise
// * due to floating point roundoff error when the velocity is near zero.
// ****************************************************************************
nodes void ∂ForNodes(void) in (node ∂∂x) inout (node ∂x) @ 3.0{
  ∂x += ∂∂x*δt ;
  ∂x.x = (norm(∂x.x)<option_u_cut)?0.0;
  ∂x.y = (norm(∂x.y)<option_u_cut)?0.0;
  ∂x.z = (norm(∂x.z)<option_u_cut)?0.0;
}

// ****************************************************************************
// * The routine CalcPositionForNodes() performs the last step in the nodal
// * advance portion of the algorithm by integrating the velocity at each node
// * to advance the position of the node to tn+1.
// ****************************************************************************
nodes void coordForNodes(void) in (node ∂x) @ 3.1{
  coord += ∂x*δt;
}

// ****************************************************************************
// * calcElemVolume
// ****************************************************************************
cells void calcElemVolume(void)
  in (node coord, node ∂x,cell v, cell volo, cell delx_xi, cell delv_xi,
      cell delx_eta, cell delv_eta, cell delx_zeta, cell delv_zeta)
  inout (cell ε) out (cell delv, cell vnew, cell calc_volume, cell vdov, cell arealg) @ 4.0{
  const ℝ dt2= ½*δt;
  const ℝ δ = 1.e-36;
  ℝ³ B[8],X[8],Xd[8];
  ℝ DetJ,volume,ρVolume;
  foreach node X[n]=coord;
  foreach node Xd[n]=∂x;
  volume = calc_volume = computeElemVolume(X);
  vnew = ρVolume = volume/volo;
  delv = ρVolume - v;
  arealg = calcElemCharacteristicLength(X,volume);
   {
    const ℝ vol = volo*vnew;
    const ℝ nrm = 1.0/(vol+δ);
    const ℝ³ di =  ¼*((X[1]+X[2]+X[6]+X[5])-(X[0]+X[3]+X[7]+X[4]));
    const ℝ³ dj = -¼*((X[0]+X[1]+X[5]+X[4])-(X[3]+X[2]+X[6]+X[7]));
    const ℝ³ dk =  ¼*((X[4]+X[5]+X[6]+X[7])-(X[0]+X[1]+X[2]+X[3]));
    const ℝ³ a_xi = (dj×dk);
```

```
    const ℝ³ a_eta = (dk×di);
    const ℝ³ a_zeta = (di×dj);
    const ℝ³ dv_xi   =  ¼*((Xd[1]+Xd[2]+Xd[6]+Xd[5])−(Xd[0]+Xd[3]+Xd[7]+Xd[4]));
    const ℝ³ dv_eta = −¼*((Xd[0]+Xd[1]+Xd[5]+Xd[4])−(Xd[3]+Xd[2]+Xd[6]+Xd[7]));
    const ℝ³ dv_zeta = ¼*((Xd[4]+Xd[5]+Xd[6]+Xd[7])−(Xd[0]+Xd[1]+Xd[2]+Xd[3]));
    delx_xi = vol/√(a_xi·a_xi+δ);
    delx_eta = vol/√(a_eta·a_eta+δ);
    delx_zeta = vol/√(a_zeta·a_zeta+δ);
    delv_zeta = (a_zeta*nrm)·dv_zeta;
    delv_xi = (a_xi*nrm)·dv_xi;
    delv_eta = (a_eta*nrm)·dv_eta;
  }
  foreach node X[n] −= dt2*Xd[n];
  DetJ=calcElemShapeFunctionDerivatives(X,B);
  ε=calcElemVelocityGradient(Xd,B,DetJ);
  vdov = ε.x+ε.y+ε.z;
  ε −= ⅓*vdov;
}
```

Listing 47: LULESH Compute Loop 0.0 ⇒ +4.0

```
// ****************************************************************************
// * This routine performs the second part of the q calculation.
// ****************************************************************************
cells void calcMonotonicQForElemsByDirectionX(xyz direction)
  in (cell elemBC, cell delv_xi) out (cell phixi){
  const ℝ monoq_limiter_mult = option_monoq_limiter_mult;
  const ℝ monoq_max_slope = option_monoq_max_slope;
  Integer bcSwitch;
  ℝ delvm=0.0;
  ℝ delvp=0.0;
  const ℝ ptiny = 1.e-36;
  const ℝ nrm = 1./(delv_xi+ptiny);
  bcSwitch = elemBC & XI_M;
  delvm = (bcSwitch == 0)?delv_xi[prevCell];
  delvm = (bcSwitch == XI_M_SYMM)?delv_xi;
  delvm = (bcSwitch == XI_M_FREE)?0.0;
  delvm = delvm * nrm ;
  bcSwitch = elemBC & XI_P;
  delvp = (bcSwitch == 0)?delv_xi[nextCell];
  delvp = (bcSwitch == XI_P_SYMM)?delv_xi;
  delvp = (bcSwitch == XI_P_FREE)?0.0;
  delvp = delvp * nrm ;
  phixi = ½ * (delvm + delvp) ;
  delvm *= monoq_limiter_mult ;
  delvp *= monoq_limiter_mult ;
  phixi = (delvm < phixi)?delvm;
  phixi = (delvp < phixi)?delvp;
  phixi = (phixi < 0.)?0.0;
  phixi = (phixi > monoq_max_slope)?monoq_max_slope;
  }

cells void calcMonotonicQForElemsByDirectionY(xyz direction)
  in (cell elemBC, cell delv_eta) out (cell phieta){
  const ℝ monoq_limiter_mult = option_monoq_limiter_mult;
  const ℝ monoq_max_slope = option_monoq_max_slope;
  Integer register bcSwitch;
  ℝ register delvm=0.;
  ℝ register delvp=0.;
  const ℝ ptiny = 1.e-36;
  const ℝ nrm = 1./(delv_eta+ptiny);
  bcSwitch = elemBC & ETA_M;
  delvm = (bcSwitch == 0)?delv_eta[prevCell];
  delvm = (bcSwitch == ETA_M_SYMM)?delv_eta;
  delvm = (bcSwitch == ETA_M_FREE)?0.0;
  delvm = delvm * nrm ;
  bcSwitch = elemBC & ETA_P;
  delvp = (bcSwitch == 0)?delv_eta[nextCell];
  delvp = (bcSwitch == ETA_P_SYMM)?delv_eta;
  delvp = (bcSwitch == ETA_P_FREE)?0.0;
  delvp = delvp * nrm ;
  phieta = ½*(delvm + delvp) ;
  delvm *= monoq_limiter_mult ;
  delvp *= monoq_limiter_mult ;
  phieta = (delvm  < phieta)?delvm;
  phieta = (delvp  < phieta)?delvp;
  phieta = (phieta < 0.0)?0.0;
  phieta = (phieta > monoq_max_slope)?monoq_max_slope;
}

cells void calcMonotonicQForElemsByDirectionZ(xyz direction)
  in (cell elemBC, cell delv_zeta) out (cell phizeta){
  const ℝ monoq_limiter_mult = option_monoq_limiter_mult;
  const ℝ monoq_max_slope = option_monoq_max_slope;
  Integer bcSwitch;
  ℝ delvm=0.;
  ℝ delvp=0.;
  const ℝ ptiny = 1.e-36;
  const ℝ nrm = 1./(delv_zeta+ptiny) ;
  bcSwitch = elemBC & ZETA_M;
  delvm = (bcSwitch == 0)?delv_zeta[prevCell];
  delvm = (bcSwitch == ZETA_M_SYMM)?delv_zeta;
  delvm = (bcSwitch == ZETA_M_FREE)?0.0;
```

```
  delvm = delvm * nrm ;
  bcSwitch = elemBC & ZETA_P;
  delvp = (bcSwitch == 0)?delv_zeta[nextCell];
  delvp = (bcSwitch == ZETA_P_SYMM)?delv_zeta;
  delvp = (bcSwitch == ZETA_P_FREE)?0.0;
  delvp = delvp * nrm ;
  phizeta = 1/2*(delvm+delvp);
  delvm *= monoq_limiter_mult ;
  delvp *= monoq_limiter_mult ;
  phizeta = (delvm < phizeta )?delvm;
  phizeta = (delvp < phizeta )?delvp;
  phizeta = (phizeta < 0.0)?0.0;
  phizeta = (phizeta > monoq_max_slope )?monoq_max_slope;
}

void calcMonotonicQForElems(void)@ 4.7{
  calcMonotonicQForElemsByDirectionX(MD_DirX);
  calcMonotonicQForElemsByDirectionY(MD_DirY);
  calcMonotonicQForElemsByDirectionZ(MD_DirZ);
}

cells void calcMonotonicQForElemsQQQL(void)
  in (cell vdov, cell elemMass, cell volo, cell vnew,
      cell delx_xi, cell delv_eta, cell delx_eta,
      cell delv_zeta, cell delx_zeta, cell delv_xi,
      cell phixi, cell phieta, cell phizeta)
  out(cell qq, cell ql)@ 4.72{
  const ℝ rho = elemMass/(volo*vnew);
  const ℝ qlc_monoq = option_qlc_monoq;
  const ℝ qqc_monoq = option_qqc_monoq;
  const ℝ delvxxi   = delv_xi*delx_xi;
  const ℝ delvxeta  = delv_eta*delx_eta;
  const ℝ delvxzeta = delv_zeta*delx_zeta;
  const ℝ delvxxit  = (delvxxi>0.0)?0.0:delvxxi;
  const ℝ delvxetat = (delvxeta>0.0)?0.0:delvxeta;
  const ℝ delvxzetat= (delvxzeta>0.0)?0.0:delvxzeta;
  const ℝ qlin = -qlc_monoq*rho*(delvxxit*(1.0-phixi)+
                                 delvxetat*(1.0-phieta)+
                                 delvxzetat*(1.0-phizeta));
  const ℝ qquad = qqc_monoq*rho*(delvxxit*delvxxit*(1.0-phixi*phixi) +
                                 delvxetat*delvxetat*(1.0-phieta*phieta) +
                                 delvxzetat*delvxzetat*(1.0-phizeta*phizeta));
  const ℝ qlint  = (vdov>0.0)?0.0:qlin;  // Remove length scale
  const ℝ qquadt = (vdov>0.0)?0.0:qquad;
  qq = qquadt ;
  ql = qlint  ;
}
```

Listing 48: LULESH Compute Loop +4.7 ⇒ +4.72

```
// ****************************************************************************
// * The routine ApplyMaterialPropertiesForElems() updates the pressure and
// * internal energy variables to their values at the new time, p_n+1 and e_n+1.
// * The routine first initializes a temporary array with the values of Vn+1 for
// * each element that was computed earlier. Then, upper and lower cut-off
// * values are applied to each array value to keep the relative volumes
// * within a prescribed range (not too close to zero and not too large).
// * Next, the routine EvalEOSForElems() is called and the array of modified
// * relative element volumes is passed to it.
// ****************************************************************************
cells void applyMaterialPropertiesForElems(void)
  in (cell vnew) out(cell vnewc) @ 5.0{
  vnewc = vnew ;
  vnewc = (vnewc < option_eosvmin)?option_eosvmin;
  vnewc = (vnewc > option_eosvmax)?option_eosvmax;
}

// ****************************************************************************
// * The routine EvalEOSForElems() calculates updated values for pressure p_n+1
// * and internal energy e_n+1.
// * The computation involves several loops over elements to pack various mesh
// * element arrays (e.g., p, e, q, etc.) into local temporary arrays.
// * Several other quantities are computed and stored in element length
// * temporary arrays also.
// * The temporary arrays are needed because the routine CalcEnergyForElems()
// * calculates p_n+1 and e_n+1 in each element in an iterative process that
// * requires knowledge of those variables at time tn while it computes the
// * new time values.
// ****************************************************************************
cells void evalEOSForElems0(void)
  in (cell e, cell delv, cell p, cell q, cell vnewc)
  out(cell e_old, cell delvc, cell p_old, cell q_old,
      cell compression, cell compHalfStep,cell work) @ 6.0{
  const ℝ vchalf = vnewc - ( ½*delv);
  work = 0.0;
  e_old = e;
  delvc = delv;
  p_old = p;
  q_old = q ;
  compression = (1.0/vnewc) - 1.0;
  compHalfStep = (1.0/vchalf)-1.0;
}

cells void evalEOSForElems1(void)
  in (cell vnewc, cell compression)
  out(cell compHalfStep) @ 6.1 {
  compHalfStep = (vnewc <= option_eosvmin)?compression;
}

cells void evalEOSForElems6(void)
  in (cell vnewc, cell compHalfStep)
  out(cell p_old, cell compression) @ 6.6 {
  p_old = (vnewc < option_eosvmax)?p_old:0.0;
  compression =(vnewc < option_eosvmax)?compression:0.0;
  compHalfStep = (vnewc < option_eosvmax)?compHalfStep:0.0;
}

// ****************************************************************************
// * The routine CalcEnergyForElems() calls CalcPressureForElems() repeatedly.
// * The function CalcPressureForElems() is the gamma Equation of State model
// * The value c1s passed to the routine is defined to be γ - 1.
// * The Equation of State calculation is a core part of any hydrocode.
// * In a production code, one of any number of Equation of State functions
// * may be called to generate a pressure that is needed to close the system
// * of equations and generate a unique solution.
// ****************************************************************************
cells void calcEnergyForElems1(void)
  in (cell e_old, cell delvc, cell p_old, cell q_old, cell work)
  inout (cell e_new) @ 7.1{
  e_new = e_old - ½*delvc*(p_old + q_old) + ½*work;
  e_new = (e_new < option_emin)?option_emin;
}
```

```
// ****************************************************************************
// * calcPressureForElems
// * p_new => pHalfStep
// * compression => compHalfStep
// * e_old => e_new
// ****************************************************************************
cells void calcPressureForElemspHalfStepcompHalfStep(void)
  in (cell compHalfStep, cell e_new)
  inout(cell bvc, cell pHalfStep)
  out (cell vnewc, cell pbvc) @ 7.2{
  const ℝ c1s = 2.0/3.0;
  bvc = c1s*(compHalfStep+1.0);
  pbvc = c1s;
  pHalfStep = bvc*e_new ;
  pHalfStep=(rabs(pHalfStep)<option_p_cut)?0.0;
  pHalfStep = (vnewc >= option_eosvmax )?0.0; // impossible condition here?
  pHalfStep = (pHalfStep < option_pmin)?option_pmin;
}

inline ℝ computeSoundSpeed(const ℝ c,//pbvc
                           const ℝ energy,
                           const ℝ volume,
                           const ℝ b,//bvc
                           const ℝ pressure,
                           const ℝ rho,
                           const ℝ _ql,
                           const ℝ _qq){
  const ℝ pe = c*energy;
  const ℝ vvbp=volume*volume*b*pressure;
  const ℝ ssc = (pe + vvbp)/rho;
  const ℝ ssct = (ssc <= 0.111111e-36)?0.333333e-18:√(ssc);
  const ℝ sscq = ssct*_ql;
  const ℝ sscqt = sscq+_qq;
  return sscqt;
}
inline ℝ computeSoundSpeed(const ℝ c,//pbvc
                           const ℝ energy,
                           const ℝ volume,
                           const ℝ b,//bvc
                           const ℝ pressure,
                           const ℝ rho){
  const ℝ pe = c*energy;
  const ℝ vvbp=volume*volume*b*pressure;
  const ℝ ssc = (pe + vvbp)/rho;
  const ℝ ssct = (ssc <= 0.111111e-36)?0.333333e-18:√(ssc);
  return ssct;
}

cells void calcEnergyForElems3(void)
  in (cell compHalfStep, cell delvc, cell pbvc, cell ql, cell qq,
      cell bvc, cell pHalfStep, cell p_old, cell q_old)
  out (cell q_new)
  inout (cell e_new) @ 7.3 {
  const ℝ vhalf = 1.0/(1.0+compHalfStep);
  const ℝ ssc = computeSoundSpeed(pbvc,e_new,vhalf,bvc,pHalfStep,option_refdens,ql,qq);
  q_new = (delvc>0.0)?0.0:ssc;
  e_new = e_new + ½*delvc*(3.0*(p_old+q_old)-4.0*(pHalfStep+q_new)) ;
}

cells void calcEnergyForElems4(void) in (cell work)
  inout (cell e_new) @ 7.4{
  e_new += ½*work;
  e_new = (rabs(e_new) < option_e_cut)?0.0;
  e_new = (e_new<option_emin)?option_emin;
}

cells void calcPressureForElemsPNewCompression(void)
  in (cell compression,
      cell bvc,
      cell e_new, cell vnewc)
      inout (cell pbvc, cell p_new) @ 7.5,7.7{
  const ℝ c1s = 2.0/3.0;
  bvc = c1s*(compression + 1.0);
```

```
  pbvc = c1s;
  p_new = bvc*e_new ;
  p_new = (rabs(p_new) < option_p_cut)?0.0;
  p_new = (vnewc >= option_eosvmax )?0.0;
  p_new = (p_new < option_pmin)?option_pmin;
}

cells void calcEnergyForElems6(void)
  in (cell delvc, cell bvc,cell pbvc, cell vnewc, cell p_new, cell ql,
      cell qq, cell p_old, cell q_old, cell pHalfStep, cell q_new)
  inout(cell e_new) @ 7.6{
  const ℝ sixth = 1.0/6.0;
  const ℝ ssc = computeSoundSpeed(pbvc,e_new,vnewc,bvc,p_new,option_refdens,ql,qq);
  const ℝ q_tilde = (delvc > 0.)?0.0:ssc;
  e_new = e_new - (7.0*(p_old + q_old)
                   - (8.0)*(pHalfStep + q_new)
                   + (p_new + q_tilde)) * delvc*sixth ;
  e_new = (rabs(e_new) < option_e_cut)?0.0;
  e_new = (e_new < option_emin)?option_emin;
}

cells void calcEnergyForElems8(void)
  in (cell delvc, cell bvc, cell pbvc, cell e_new,
      cell vnewc, cell p_new, cell ql, cell qq)
  inout(cell q_new) @ 7.8{
  const ℝ qnw = computeSoundSpeed(pbvc,e_new,vnewc,bvc,p_new,option_refdens,ql,qq);
  const ℝ qnwt = (rabs(qnw) < option_q_cut)?0.0:qnw;
  q_new = (delvc <= 0.)?qnwt;
}

cells void evalEOSForElems8(void)
  in (cell p_new, cell e_new, cell q_new)
  out(cell p, cell e, cell q) @ 8.0{
  p = p_new;
  e = e_new;
  q = q_new;
}

// ****************************************************************************
// * Lastly, the routine CalcSoundSpeedForElems() calculates the sound speed
// * sound_speed in each element using p_n+1 and e_n+1.
// * The maximum value of sound_speed is used to calculate constraints on t_n+1
// * which will be used for the next time advance step.
// ****************************************************************************
cells void calcSoundSpeedForElems(void)
  in (cell bvc, cell pbvc, cell e_new, cell vnewc, cell p_new)
  out (cell sound_speed) @ 9.0{
  foreach material{
    const ℝ ssTmpt = computeSoundSpeed(pbvc,e_new,vnewc,bvc,p_new,option_refdens);
    sound_speed = ssTmpt;
  }
}

// ****************************************************************************
// * The routine UpdateVolumesForElems() updates the relative volume to V_n+1.
// * This routine basically resets the current volume V_n in each element to
// * the new volume V_n+1 so the simulation can continue to the next time
// * increment.
// * Note that this routine applies a cut-off to the relative volume V in
// * each element. Specifically, if V is sufficiently close to one (a
// * prescribed tolerance), then V is set to one.
// * The reason for this cutoff is to prevent spurious deviations of volume
// * from their initial values which may arise due to floating point roundoff
// * error.
// ****************************************************************************
cells void updateVolumesForElems(void)
  in (cell vnew) out (cell v) @ 10.0{
  const ℝ v = vnew;
  const ℝ vt = (rabs(v-1.0)<option_v_cut)?1.0:v;
  v = vt;
}
```

Listing 49: LULESH EOS +5.0 ⇒ +10.0

```
// ****************************************************************************
// * The routine CalcCourantConstraintForElems() calculates the Courant timestep
// * constraint δt_Courant. This constraint is calculated only in elements
// * whose volumes are changing that is, dV/V!=0.
// ****************************************************************************
cells void calcCourantConstraintForElems(void)
  in (cell sound_speed, cell arealg, cell vdov)
  out (cell δt_cell_courant) @ 12.1{
  const ℝ arg_max_courant=1.0e+20;
  δt_cell_courant=arg_max_courant;
  foreach material{
    const ℝ qqc2 = 64.0 * option_qqc * option_qqc ;
    const ℝ δf = sound_speed[m] * sound_speed[m];
    const ℝ δft=(vdov[m]<0.0)?qqc2*arealg[m]*arealg[m]*vdov[m]*vdov[m]:0.0;
    const ℝ δfpp = δf+δft;
    const ℝ δfp = √(δfpp);
    const ℝ aδfp = arealg[m]/δfp;
    δt_cell_courant=(vdov!=0.0)?min(arg_max_courant,aδfp);
   }
}

// ****************************************************************************
// * The routine CalcHydroConstraintForElems() calculates the hydro timestep
// * constraint. Similar to δt_Courant, δt_hydro is calculated only in elements
// * whose volumes are changing. When an element is undergoing volume change,
// * δt_hydro for the element is some maximum allowable element volume change
// * (prescribed) divided by dV/V in the element.
// ****************************************************************************
cells void calcHydroConstraintForElems(void)
  in (cell vdov) out (cell δt_cell_hydro) @ 12.2{
  const ℝ arg_max_hydro=1.0e+20;
  δt_cell_hydro=arg_max_hydro;
  foreach material{
    const ℝ δdv = rabs(vdov[m]);
    const ℝ δdve = δdv+1.e−20;
    const ℝ δdvov = option_dvovmax/δdve;
    const ℝ δhdr = min(arg_max_hydro,δdvov);
    δt_cell_hydro=(vdov!=0.0)?δhdr;
  }
}

// ****************************************************************************
// * After all solution variables are advanced to t_n+1, the constraints δtCourant
// * and δthydro for the next time increment t_n+1 are calculated in this routine.
// * Each constraint is computed in each element and then the final constraint value
// * is the minimum over all element values.
// * The constraints are applied during the computation of δt for the next time step.
// ****************************************************************************
∀ cells δt_courant <?= δt_cell_courant @ 12.11;
∀ cells δt_hydro   <?= δt_cell_hydro   @ 12.22;
```

Listing 50: LULESH Courant Hydro Constraints +12.0 ⇒ +∞

```
// ****************************************************************************
// * calcElemShapeFunctionDerivatives
// ****************************************************************************
ℝ calcElemShapeFunctionDerivatives(const ℝ³* X, ℝ³* β){
  const ℝ³ fjxi = ⅛*((X[6]−X[0])+(X[5]−X[3])−(X[7]−X[1])−(X[4]−X[2]));
  const ℝ³ fjet = ⅛*((X[6]−X[0])−(X[5]−X[3])+(X[7]−X[1])−(X[4]−X[2]));
  const ℝ³ fjze = ⅛*((X[6]−X[0])+(X[5]−X[3])+(X[7]−X[1])+(X[4]−X[2]));
  // compute cofactors
  const ℝ³ cjxi =  (fjet×fjze);
  const ℝ³ cjet = −(fjxi×fjze);
  const ℝ³ cjze =  (fjxi×fjet);
  // calculate partials: this need only be done for 0,1,2,3
  // since, by symmetry, (6,7,4,5) = − (0,1,2,3)
  β[0] = − cjxi−cjet−cjze;
  β[1] =   cjxi−cjet−cjze;
  β[2] =   cjxi+cjet−cjze;
  β[3] = − cjxi+cjet−cjze;
  β[4] = −β[2];
  β[5] = −β[3];
  β[6] = −β[0];
  β[7] = −β[1];
  // calculate jacobian determinant (volume)
  return 8.0*(fjet⋅cjet);
}

// ****************************************************************************
// * calcElemVelocityGradient
// ****************************************************************************
ℝ³ calcElemVelocityGradient(const ℝ³* υ,
                            const ℝ³* B,
                            const ℝ detJ){
  const ℝ inv_detJ=1.0/detJ;
  const ℝ³ υ06=υ[0]−υ[6];
  const ℝ³ υ17=υ[1]−υ[7];
  const ℝ³ υ24=υ[2]−υ[4];
  const ℝ³ υ35=υ[3]−υ[5];
  return inv_detJ*(B[0]*υ06+B[1]*υ17+B[2]*υ24+B[3]*υ35);
}

// ****************************************************************************
// * computeElemVolume
// ****************************************************************************
ℝ computeElemVolume(const ℝ³* X){
  const ℝ twelveth = 1.0/12.0;
  const ℝ³ d31=X[3]−X[1];
  const ℝ³ d72=X[7]−X[2];
  const ℝ³ d63=X[6]−X[3];
  const ℝ³ d20=X[2]−X[0];
  const ℝ³ d43=X[4]−X[3];
  const ℝ³ d57=X[5]−X[7];
  const ℝ³ d64=X[6]−X[4];
  const ℝ³ d70=X[7]−X[0];

  const ℝ³ d14=X[1]−X[4];
  const ℝ³ d25=X[2]−X[5];
  const ℝ³ d61=X[6]−X[1];
  const ℝ³ d50=X[5]−X[0];

  const ℝ tp1 = (d31+d72)⋅(d63×d20);
  const ℝ tp2 = (d43+d57)⋅(d64×d70);
  const ℝ tp3 = (d14+d25)⋅(d61×d50);
  return twelveth*(tp1+tp2+tp3);
}

// ****************************************************************************
// * AreaFace
// ****************************************************************************
ℝ AreaFace(const ℝ³ X0, const ℝ³ X1, const ℝ³ X2, const ℝ³ X3){
  const ℝ³ f=(X2−X0)−(X3−X1);
  const ℝ³ g=(X2−X0)+(X3−X1);
```

```
  return (f·f)*(g·g)-(f·g)*(f·g);
}

// ****************************************************************************
// * calcElemCharacteristicLength
// ****************************************************************************
ℝ calcElemCharacteristicLength(const ℝ³ X[8], const ℝ ν){
  ℝ χ=0.0;
  χ=max(AreaFace(X[0],X[1],X[2],X[3]),χ);
  χ=max(AreaFace(X[4],X[5],X[6],X[7]),χ);
  χ=max(AreaFace(X[0],X[1],X[5],X[4]),χ);
  χ=max(AreaFace(X[1],X[2],X[6],X[5]),χ);
  χ=max(AreaFace(X[2],X[3],X[7],X[6]),χ);
  χ=max(AreaFace(X[3],X[0],X[4],X[7]),χ);
  return 4.0*ν/√(χ);
}

// ****************************************************************************
// * Σ_FaceNormal
// ****************************************************************************
void Σ_FaceNormal(ℝ³* β,
                  const int ia, const int ib,
                  const int ic, const int id,
                  const ℝ³* X){
  const ℝ³ bisect0 = ½*(X[id]+X[ic]-X[ib]-X[ia]);
  const ℝ³ bisect1 = ½*(X[ic]+X[ib]-X[id]-X[ia]);
  const ℝ³ α = ¼*(bisect0×bisect1);
  β[ia] += α; β[ib] += α;
  β[ic] += α; β[id] += α;
}

// ****************************************************************************
// * calcElemVolumeDerivative
// * We keep the next one to allow sequential binary reproductibility
// ****************************************************************************
ℝ³ ∂Volume(const ℝ³ X0, const ℝ³ X1, const ℝ³ X2,
           const ℝ³ X3, const ℝ³ X4, const ℝ³ X5){
  const ℝ³ v01 = X0+X1;
  const ℝ³ v12 = X1+X2;
  const ℝ³ v25 = X2+X5;
  const ℝ³ v04 = X0+X4;
  const ℝ³ v34 = X3+X4;
  const ℝ³ v35 = X3+X5;
  return (1.0/12.0)*((v12×v01)+(v04×v34)-(v25×v35));
}

// ****************************************************************************
// * compute the hourglass modes
// ****************************************************************************
void cHourglassModes(const int i, const ℝ δ,
                     const ℝ³ *Δ, const ℝ γ[4][8],
                     const ℝ³ *χ,
                     ℝ* h0, ℝ* h1,
                     ℝ* h2, ℝ* h3,
                     ℝ* h4, ℝ* h5,
                     ℝ* h6, ℝ* h7){
  const ℝ υ=1.0/δ;
  const ℝ³ η =
    χ[0]*γ[i][0]+χ[1]*γ[i][1]+χ[2]*γ[i][2]+χ[3]*γ[i][3]+
    χ[4]*γ[i][4]+χ[5]*γ[i][5]+χ[6]*γ[i][6]+χ[7]*γ[i][7];
  h0[i] = γ[i][0]-υ*(Δ[0]·η);
  h1[i] = γ[i][1]-υ*(Δ[1]·η);
  h2[i] = γ[i][2]-υ*(Δ[2]·η);
  h3[i] = γ[i][3]-υ*(Δ[3]·η);
  h4[i] = γ[i][4]-υ*(Δ[4]·η);
  h5[i] = γ[i][5]-υ*(Δ[5]·η);
  h6[i] = γ[i][6]-υ*(Δ[6]·η);
  h7[i] = γ[i][7]-υ*(Δ[7]·η);
}
```

Listing 51: LULESH Geometric Functions

References

[1] ∇-Nabla, www.nabla-lang.org.

[2] JS. Camier. ∇-Nabla: A Numerical-Analysis Specific Language for Exascale Scientific Applications. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2014.

[3] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C, 1999.

[4] Lawrence Livermore National Laboratory. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254.