Improving Performance Portability and Exascale Software Productivity with the ∇ Numerical Programming Language

Jean-Sylvain Camier
CEA, DAM, DIF, F-91297 Arpajon, France.
Jean-Sylvain.Camier@cea.fr

1. ABSTRACT

Addressing the major challenges of software productivity and performance portability becomes necessary to take advantage of emerging extreme-scale computing architectures. As software development costs will continuously increase to deal with exascale hardware issues, higher-level programming abstractions will facilitate the path to go. There is a growing demand for new programming environments in order to improve scientific productivity, to ease design and implementation, and to optimize large production codes.

We introduce the numerical analysis specific language Nabla (∇) which improves applied mathematicians productivity, and enables new algorithmic developments for the construction of hierarchical composable high-performance scientific applications. One of the key concept is the introduction of the hierarchical logical time within the high-performance computing scientific community. It represents an innovation that addresses 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. Control and data concurrencies are combined consistently to achieve statically analyzable transformations and efficient code generation. Shifting the complexity to the compiler offers an ease of programming and a more intuitive approach, while reaching the ability to target new hardware and leading to performance portability.

In this paper, we present the three main parts of the ∇ toolchain: the frontend raises the level of abstraction with its grammar; the backends hold the effective generation stages, and the middle-end provides agile software engineering practices transparently to the application developer, such as: instrumentation (performance analysis, V&V, debugging at scale), data or resource optimization techniques (layout, locality, prefetching, caches awareness, vectorization, loop fusion) and the management of the hierarchical logical time, which produces the graphs of all parallel tasks. The refactoring of existing legacy scientific applications is also possible by the incremental compositional approach of the method.

2. INTRODUCTION

Nabla (∇) is an open-source [4] Domain Specific Language (DSL) introduced in [6] 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, even at the end of long development processes. The self-evident truth is that it is too late for these optimizations to be significantly effective with standard approaches. 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.

The remainder of the paper is organized as follow. Section 3 gives an overview of the ∇ domain specific language. Section 4 introduces the hierarchical logical time concept. The different elements of the toolchain are described in section 5. Finally, section 6 provides some evaluations and experimental results for Livermore's Unstructured Lagrange Explicit Shock Hydrodynamics (Lulesh) [15] proxy application on different target architectures.

2.1 Related Work

Because application codes have become so large and the exploration of new concepts too difficult, domain specific languages are becoming even more attractive by offering the possibility to explore a range of optimizations.

Loci[18] is an automatic parallelizing framework that has been supporting the development of production simulation codes for more than twenty years. The framework provides a way to describe the computational kernels using a relational rule-based programming model. The data-flow is extracted, transformations are applied to optimize the scheduling of the computations; data locality can also be enhanced, improving the overall scalability of a Loci application.

SpatialOps which provides the **Nebo**[11] EDSL is an embedded C++ domain specific language for platform-agnostic PDE solvers [21]. It provides an expressive syntax, allowing application programmers to focus primarily on physics model formulation rather than on details of discretization and hardware. Nebo facilitates portable operations on structured mesh calculations and is currently used in a number of multiphysics applications including multiphase, turbulent reacting flows.

Liszt[10] is a domain-specific language for solving PDE on meshes for a variety of platforms, using efficient different parallel models: MPI, pthreads and CUDA. The design of computational kernels is facilitated: the data dependencies being taken care of by the compiler.

Terra[9] is a low-level language, designed for high performance computing, interoperable with Lua [20]. It is a statically typed, compiled language with manual memory management and a shared lexical environment.

Scout[19] is a compiled domain-specific language, targeting CUDA, OpenCL and the Legion[5] runtime. It does not provides a source-to-source approach, but supports mesh-based applications, in-situ visualization and task parallelism. It also includes a domain-aware debugging tool.

RAJA[14, 16] is a thin abstraction layer consisting of a parallel-loop construct for managing work and an IndexSet construct for managing data. The composition of these components allows architecture specific details of programming models, vendor compilers, and runtime implementations to be encapsulated at a single code site, isolating software applications from portability-related disruption, while enhancing readability and maintainability.

Thanks to the source-to-source approach and its exclusive logical time model, ∇ is able to target some of the above listed languages. It allows developers to think in terms of more parallelism, letting the compilation process tools perform appropriate optimizations.

Listing 1: Libraries and Options Declaration in ∇

```
with ℵ, slurm;

options{
  Real option_dtfixed          = -1.0e-7;
  Real option_δt_initial       = 1.0e-7;
  Real option_δt_courant       = 1.0e+20;
  Real option_δt_hydro         = 1.0e+20;
  Real option_ini_energy       = 3.948746e+7;
  Real option_stoptime         = 1.0e-2;
  Bool option_rdq              = false;
  Real option_rdq_α            = 0.3;
  Integer option_max_iterations = 8;
  Bool option_only_one_iteration = false;
};
```

3. OVERVIEW OF THE NABLA DSL

This section introduces the ∇ language, which allows the conception of multi-physics applications, according to a logical time-triggered approach. Nabla is a domain specific language which embeds the C language. It 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.

3.1 Lexical & Grammatical Elements

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.

Listing 2: Variables Declaration in ∇

```
nodes{
  Real3 ∂tx;
  Real3 ∂t2x;
  Real3 nForce;
  Real nMass;
};
cells{
  Real p;
  Real3 ϵ;
  Real3 cForce[nodes];
  Real delv_xi;
};
```

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. Libraries are introduced by the `with` token: additional keywords will then be 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. 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. Listing 1 provides an example of libraries and options declaration in ∇.

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*. Listing 2 shows two declarations of variables living on *nodes* and *cells*. Velocity (∂tx), acceleration ($\partial t2x$) 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*.

Different data types are also available, such as `Integer`, `Bool`, `Real` or three-dimension vector types `Real3`, allowing the insertion of specific directives during the second compilation stage. Unicode letter and some additional mathematical operators are also provided: the Listing 3 gives some operators that are actually supported and particularly used in reduction statements, assignment, conditional, primary and multiplicative expressions.

Listing 3: Additional ∇ Expressions

```
<?= >?= ?= @ ∀ ℵ ∧ ∨ ∞ ² ³ √ ∛ ½ ⅓ ¼ ⅛ ⋆ · × ⊗
```

3.2 Functions and Jobs Declaration

In order to schedule standard *functions* and *jobs*, the language provides some new syntactic elements, allowing the design of logical time-triggered multi-tasking applications. Data-parallelism is implicitly expressed by the declaration of *jobs*, which are functions with additional attributes. The first attribute is the *item* on which the function is going to iterate: it can be a for-loop on *cells* or *nodes* for example. *Input* and *output* variables the *job* will work with are also to be provided. Finally an attribute telling *when* the *job* shall be triggered can also be given: this introduces the logical-time triggered model of parallelism that is presented in section 4.

Listing 4: ∇ Job Declaration, a *for-loop* on *nodes*

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

Listing 4 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.

Listing 5: ∇ Job Declaration, another on *cells*

```
cells void temporalComputeStdFluxesSum(void)
     in (cell reconstructed_u, node u,
         cell reconstructed_p,
         cell CQs, cell AQs)
    out (cell momentum_fluxes_Σ,
         cell total_energy_fluxes_Σ) @ 16.0 {
    foreach node{
      const Real3 Δu = reconstructed_u-u;
      Real3 FQs = AQs ⋆ Δu;
      FQs += reconstructed_p*CQs;
      momentum_fluxes_Σ -= FQs;
      total_energy_fluxes_Σ -= FQs·u;
    }
}
```

The job in Listing 5 illustrates an explicit scheme [8]. It is a for-loop on *cells*, triggered at the logical time '`+16.0`', and with an inner connectivity loop on each of the cell's nodes. Two mathematical operators are used in this job: the vector dot product '·' and the matrix vector product '⋆'.

Listing 6: ∇ Job Declaration Statement

```
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;
}
```

Listing 6 comes from the proxy application Lulesh [17], during the equation of state phase. It is a *cell* job, working on some of its variables and set to be launched at logical time '`7.1`'. It shows the use of the '`?`' binary operator, which changes the ternary standard C '`?:`', by allowing to omit the '*else*' ('`:`') statements, meaning here '*else unchanged*'.

Listing 7: ∇ Implicit Job: filling a *matrix*

```
nodes void δNodes(void)
  in (face δ,
      node θ,
      node node_area,
      node node_is_an_edge,
      face Cosθ, face sdivs) @ 3.4 {
  Real δn,Σδ=0.0;
  if (node_is_an_edge) continue;
  foreach face {
    Node other=(node[0]==this)?node[1]:node[0];
    δn=δ/node_area;
    Σδ+=1.0/(Cosθ*sdivs);
    ℵ matrix addValue(θ,this,θ,other,-δn);
  }
  Σδ*=δt/node_area;
  ℵ matrix addValue(θ,this,θ,this,1.0+Σδ);
}
```

The job in Listing 7 is a for-loop on each node of the domain. It is set to be triggered at '`+3.4`'. The aleph (ℵ) library token and its programming interface is used to fill the matrix. The degrees of freedom are deduced by the use of pairs of the form: *(item,variable)*. Two degrees of freedom are used here: *(θ,this)* and *(θ,other)*.

More simple jobs can be expressed, like the one presented in Listing 8. It is one of the two reductions required at the end of each iteration in the compute loop of Lulesh. *δt_hydro* is the global variable and the *δt_cell_hydro* is the one attached to each cell. It is here a `minimum` reduction over all the *cells*.

Listing 8: ∇ Reduction Statement

```
∀ cells δt_hydro <?= δt_cell_hydro @ 12.22;
```

The different '`@`' attributes are then gathered and combined hierarchically by the toolchain, in order to create the logical time triggered execution graph, used for the scheduling of all *functions* and *jobs* of the application. Next section introduces the composition of such logical time statements.

4. HIERARCHICAL LOGICAL TIME

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.

Table 1: ∇ Logical Time Diagrams: [a] is the totally-ordered time-diagram from a typical mini-application ported to ∇ with consecutive *for-loops*; [b] is the diagram of a better partially-ordered numerical scheme. Colors stand for the *job items*.

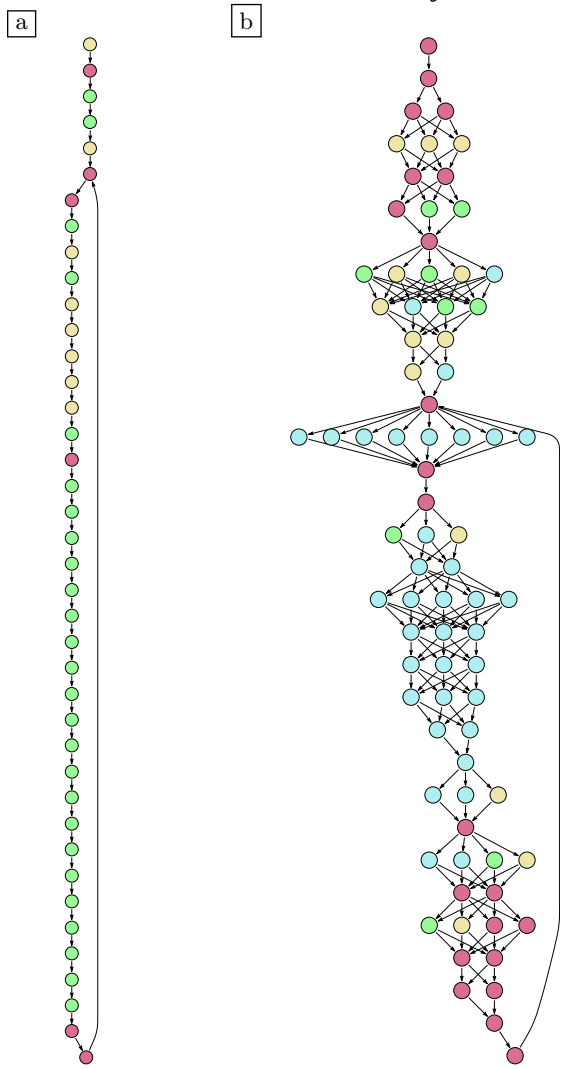

Table 1 presents two kinds of execution graphs: the first one (a) is taken from a typical proxy application, as the second one (b) comes from a new implicit numerical scheme [7], designed and written in ∇ entirely. No additional parallelism lies in the first totally-ordered diagram, whereas the second one exposes a new dimension that can be exploited for scheduling.
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. This composition is actually done by the compiler with command line options; the need of frontend tools will rapidly be crucial as applications grow bigger.

5. THE NABLA TOOLCHAIN

The ∇ toolchain is composed of three main parts, illustrated in Figure 1. The frontend is a Flex [1] and Bison [2] parser that reads a set of ∇ input files. The middle-end provides a collection of software engineering practices, transparently to the application developer. Instrumentation, performance analysis, validation and verification steps are inserted during this process. Data layout optimizations can also take place during this phase: locality, prefetching, caches awareness and loop fusion techniques are in development and will be integrated in the middle-end.

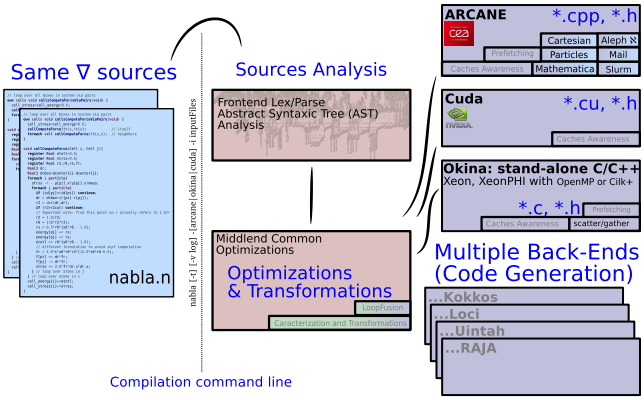

Figure 1: The three parts of the ∇ Toolchain: the Sources Analysis (Frontend), the Optimizations & Transformations (Middle-end) and the Generation Stages (Backends).

The backends hold the effective generation stages for different targets or architectures:

- **ARCANE** [13]. It is a numerical code framework for high-performance computing. It provides multiple strategies of parallelism: MPI, threads, MPI+threads and the Multi-Processor Computing framework (MPC) [24].
- **CUDA** [25, 22]. CUDA is a programming model to target NVIDIA's graphics processing unit (GPU). The ∇ compiler generates homogenous source files: all the numerical kernels run exclusively on the GPU. Initial speedups have been achieved with only minimal initial investment of time for this backend: further optimization opportunities are to be identified and deployed.
- **OKINA**. This standalone backend comes with the ∇ toolchain. C/C++ source files are generated: the code is fully-vectorized by the use of *intrinsics* classes. The choice of underlying data structures is possible for different hardwares: specific layouts with their associated *prefetch* instructions or the Kokkos [12] abstraction layer, are two examples. For now, only OpenMP 4.0 [23] and Cilk+ [3] can be used as underlying parallel execution runtimes.

As a demonstration of the potential and the efficiency of this approach, the next section presents the Lulesh benchmark, implemented in ∇. The performances are evaluated for a variety of hardware architectures.

6. ∇-LULESH EXPERIMENTAL RESULTS

The Lulesh benchmark solves one octant of the spherical Sedov blast wave problem using Lagrangian hydrodynamics for a single material. Equations are solved using a staggered mesh approximation. Thermodynamic variables are approximated as piece-wise constant functions within each element and kinematic variables are defined at the element nodes. For each figure, cells-updates-per-μs for different kind of runs are presented. On Figures 2 and 3, yellow bars show the reference performances of the downloadable version, the violet ones are obtained with the hand-optimized OpenMP version and finally, blue ones are the performances reached from the ∇-Lulesh source files and the OKINA backend.

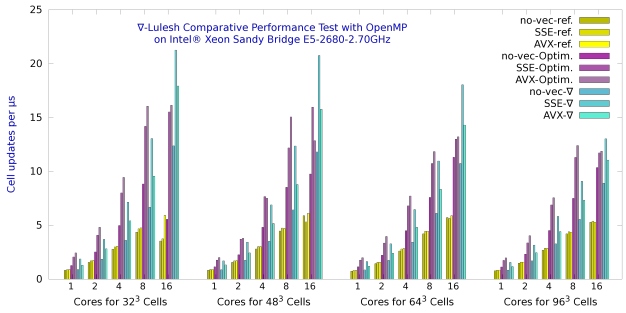

Figure 2: Reference (ref.), Optimized (Optim.) and ∇ Lulesh Performances Tests on Intel Xeon-SNB with the C/C++ Standalone OKINA+OpenMP Backend and no-vec., SSE or AVX Intrinsics. Higher is better.

Figure 2 shows the results obtained on Xeon Sandy Bridge E5-2680@2.7GHz architectures. Different kinds of vectorization are represented for each run: no-vectorization, only SSE and full AVX. The ∇-Lulesh version presents a similar level of performances as the optimised one, despite the `scatter` and `gather` instructions generated by the backend which are emulated by software on this architecture.

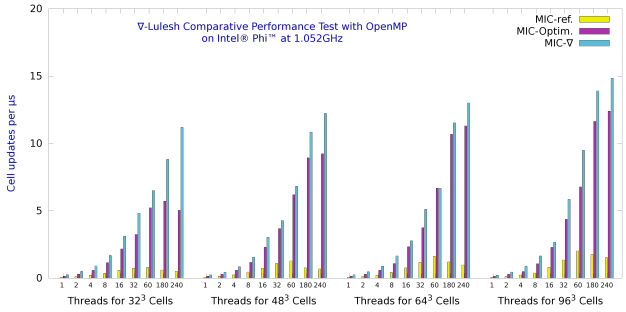

Figure 3: Reference (ref.), Optimized (Optim.) and ∇ Lulesh Performances Tests on Intel Xeon PHI with the C/C++ Standalone OKINA+OpenMP Backend and AVX512 Intrinsics

Figure 3 shows the performances obtained on Intel Xeon-PHI processors with the AVX512 vectorization intrinsics. In this example, the `scatter` and `gather` operation codes, supported by the hardware, are not emulated anymore and a higher level of performances is reached, better than the hand-tuned version.

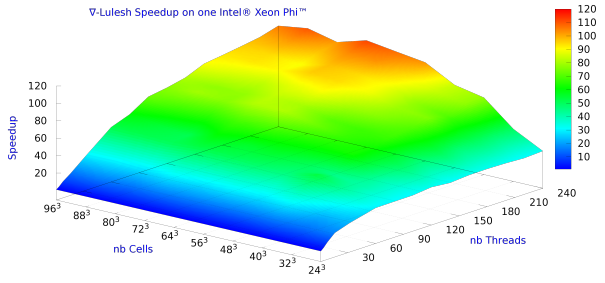

Figure 4: ∇-Lulesh Speedups on Intel Xeon PHI: A speedup of more than one hundred is reached for a mesh of 125000 Elements with 240 Threads

Figure 4 shows the speedup with the OKINA+OpenMP backend that is reached for different runs. The number of cells are presented on the X-axis, the number of threads used on the Y-axis. The 3D-surface renders in hot colors where the application starts taking advantage of hyper-threading on this architecture. A speedup of more than a hundred is reached for a mesh of more than one hundred thousands of cells on more than two hundreds of threads.

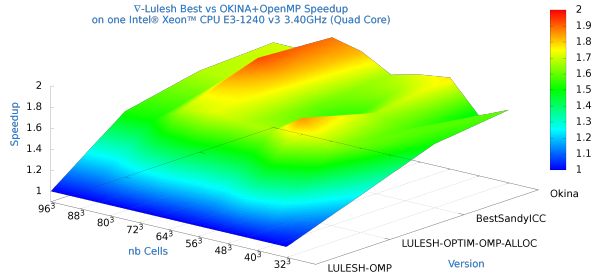

Figure 5: ∇-Lulesh Speedups on a quad core Intel Xeon Haswell: the OKINA+OpenMP backend vs other OpenMP versions

Finally, Figure 5 presents the performance results on a single Intel Xeon Haswell E3-1240v3 at 3.40GHz of different OpenMP versions of Lulesh. The `LULESH-OMP` one can be downloaded and is the reference in this test. The `LULESH-OPTIM-OMP-ALLOC` is an optimized version and the `BestSandy` is the fastest that can be found on the web site: it stands for the best candidate with OpenMP. The last one is the ∇-Lulesh version with OKINA. The 3D-surface renders again in hot colors the best speedups that are reached for different mesh sizes and for the different versions. The results of the OKINA backend are as good as the `BestSandyICC` ones: the back of the surface stays on the same level of speedup.

These results emphasize the opportunity for domain-specific languages. Doing so opens up a potential path forward for enhanced expressivity and performance. ∇ achieves both portably, while maintaining a consistent programming style and offering a solution to the productivity issues.

7. DISCUSSION AND FUTURE WORK

The numerical-analysis specific language Nabla (∇) provides a productive development way for exascale HPC technologies, flexible enough to be competitive in terms of performances. The refactoring of existing legacy scientific applications is also possible by the incremental compositional approach of the method. Raising the loop-level of abstractions allows the framework to be prepared to address growing concerns of future systems. There is no need to choose today the best programming model for tomorrow's architectures: ∇ does not require to code multiple versions of kernels for different models.

Nabla's source-to-source approach and its exclusive logical time model will facilitate future development work, focusing on new backends: other programming models, abstraction layers or numerical frameworks are already planned.

The generation stages will be improved to incorporate and exploit algorithmic or low-level resiliency methods by coordinating co-designed techniques between the software stack and the underlying runtime and operating system.

∇ is open-source, ruled by the French CeCILL license, which is a free software license, explicitly compatible with the GNU GPL.

8. REFERENCES

[1] Flex: The Fast Lexical Analyzer, flex.sourceforge.net.

[2] GNU Bison, www.gnu.org/software/bison.

[3] Intel Cilk Plus, www.cilkplus.org.

[4] ∇-Nabla, www.nabla-lang.org.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

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

[7] J.-S. Camier and F. Hermeline. A Monotone Non-Linear Finite Volume Method for Approximating Diffusion Operators on General Meshes, To Appear.

[8] G. Carré, S. Del Pino, B. Després, and E. Labourasse. A cell-centered lagrangian hydrodynamics scheme on general unstructured meshes in arbitrary dimension. *J. Comput. Phys.*, 228(14):5160–5183, Aug. 2009.

[9] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[11] C. Earl, M. Might, A. Bagusetty, and J. C. Sutherland. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations.

[12] H. C. Edwards and D. Sunderland. Kokkos Array Performance-portable Manycore Programming Model. Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), 2012.

[13] G. Grospellier and B. Lelandais. The Arcane Development Framework. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11, New York, NY, USA, 2009. ACM.

[14] R. Hornung and J. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, 2014.

[15] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, December 2012.

[16] J. Keasler and R. Hornung. Managing Portability for ASC Applications. In *SIAM Conference on Computational Science and Engineering*, 2015.

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

[18] E. A. Luke. Loci: A Deductive Framework for Graph-Based Algorithms. In *Computing in Object-Oriented Parallel Environments, Third International Symposium, ISCOPE 99, San Francisco, California, USA, December 8-10, 1999, Proceedings*, pages 142–153, 1999.

[19] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 1–10, Piscataway, NJ, USA, 2014. IEEE Press.

[20] F. P. Miller, A. F. Vandome, and J. McBrewster. *Lua (Programming Language)*. Alpha Press, 2009.

[21] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software. *ACM Trans. Math. Softw.*, Nov. 2012.

[22] NVIDIA. CUDA C Programming Guide, 2014.

[23] OpenMP Application Program Interface, version 4.0. OpenMP Architecture Review Board, 2013.

[24] M. Pérache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, Berlin, Heidelberg, 2008.

[25] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.