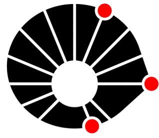

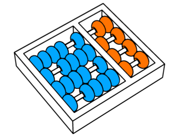

Extending a finite difference domain-specific language to a distributed runtime system

J. Ribeiro *H. Yviquel*

Relatório Técnico - IC-PFG-21-30
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Estendendo uma linguagem de domínio específico de diferenças finitas para um runtime de tarefas distribuídas

Extending a finite difference domain-specific language to a distributed runtime system

José Ribeiro* Hervé Yviquel*

Resumo

A computação científica está levando a indústria a resolver problemas cada vez mais difíceis. No entanto, isto representa grandes desafios para cientistas e desenvolvedores, tornando difícil implementar / manter esses problemas e ainda entregar resultados de alta performance aos usuários finais. Para lidar com essas questões, Devito surgiu nos últimos anos. Este framework combina inteligentemente tecnologia de compiladores com um motor de matemática simbólica baseado em Python para entregar uma execução otimizada e transparente para os usuários finais. Apesar das questões relacionadas à solução e à manutenção serem consideravelmente amenizadas com Devito, os aspectos de desempenho ainda são uma preocupação que invariavelmente leva o sistema a usar execuções distribuídas. Ao usar o padrão MPI, o Devito fornece efetivamente uma solução para este problema. No entanto, o MPI é historicamente difícil de manter e apresenta várias responsabilidades aos desenvolvedores do Devito. Para contornar os problemas do MPI, o OmpCluster se apresenta como uma ótima solução, fornecendo ao programador uma sintaxe de código mais simples, que se estende a partir da conhecida API OpenMP, porém tão confiável e escalável quanto o MPI. Portanto, neste relatório o framework Devito foi estendido por meio de sua especialização de backend para que pudesse usar as boas propriedades fornecidas pelo runtime de tarefas distribúidas OmpCluster. Além disso, resultados numéricos mostraram o potencial escalonável do backend criado. Em conclusão, algumas restrições necessárias são apresentadas para que o usuário possa estar ciente das limitações existentes na implementação atual.

Abstract

Scientific Computing is pushing the industry to increasingly solve harder problems. However, this is posing considerable challenges to scientists and developers, making it difficult to implement/maintain those problems and still deliver high-performance results to final users. To deal with those struggles, Devito has arisen in the past few years. This framework cleverly combines compiler technology with a Python-based symbolic mathematical engine to deliver transparent and optimized execution to end-users. Despite the issues related to solving and maintaining are considerably alleviated with Devito, the performance aspects are still a concern which invariably leads the system toward using distributed executions. By using the MPI standard, Devito effectively provides a solution to that problem. However, MPI is historically difficult to maintain and poses multiple responsibilities to Devito's developers. To overcome the MPI issues, OmpCluster presents itself as a great solution, providing to the programmer a simpler code syntax, which is extended from the very known OpenMP API, yet as reliable and scalable as MPI. So in this report, the Devito framework was extended through its backend specialization so that it could use the good properties provided by the OmpCluster runtime system. On top of that, numerical results have shown the scalable potential of the created backend. In conclusion, some necessary restrictions are presented so that the user can be conscious of the existing limitations of the current implementation.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Contents

1 Introduction **3**

2 Theoretical background **3**
2.1 Devito . . . 3
2.1.1 Specifying a Finite - Difference Scheme . . . 4
2.1.2 Point regions . . . 7
2.1.3 Compiler . . . 7
2.1.4 Specializing operators through backends . . . 9
2.2 OpenMP . . . 9
2.3 MPI . . . 9
2.4 Ompcluster . . . 10

3 Backend Specialization **11**
3.1 OpenMP Target Code . . . 11
3.2 OmpCluster code based on OpenMP Target Code . . . 13
3.3 Domain Decomposition . . . 13
3.4 Pointer Cast Substitutions . . . 14

4 Discussions and Limitations **15**
4.1 Numerical Experiments . . . 15
4.2 No direct execution . . . 16
4.3 Pointer Substitution . . . 16
4.4 Data Exchange . . . 17

5 Conclusion **18**

1 Introduction

This report extends the Devito framework to the distributed runtime system called OmpCluster. The idea behind that is to provide Devito with a simpler code syntax, yet as reliable and scalable as MPI. Thus, also easing the process of code generation and maintainability.

To achieve that, a simplification without considering data exchange between nodes was first considered. Given the limited time, the step considering this exchange was left as future work. Also, due to the OpenMP syntax, some major restrictions in the backend specialization had to be considered, which for instance deviated the data pointer from its common representation. Because of LLVM, the backend engine used to compile the generated code, another important restriction was introduced: the impossibility of executing the specialized generated code directly inside the python environment. Though a major attraction of Devito, this last restriction was left aside, and code generation, not direct execution, became the main focus of this report.

To validate the correctness and performance of the implementation, several tests were performed. Numerical results show the interesting scalable potential of the implementation. Therefore, the specialization scheme depicted in this report has proven to achieve its purpose by providing Devito with the first approach toward a specialized OmpCluster code, which is scalable, reliable, and easier to maintain.

2 Theoretical background

2.1 Devito

Given the complexity of scientific problems, the issues that arise with them (such as scarce memory and underused resources), and the request to use distributed systems as the only option to viably solve them, make high-Performance Computing code difficult to implement, maintain, or port. By using a Domain Specific Language (DSL) [5] instead of a lower-level language, the abstraction level can be increased and, therefore, those issues can be alleviated. DSLs are languages that can be used to express numerical methods through a syntax that highly resembles their mathematical expressions. Therefore, while the user focuses on what really matters, which is expressing the problem, the compiler and library stack behind the DSL take the control of the lower-level code generation and optimization. And this is the precise context where Devito [8] appears. More specifically, Devito focuses on transforming symbolic Python code based on a Finite - Difference (FD) scheme [6], into C / C++ executable code. The biggest advantages in using such a framework are:

1. Capable of solving large partial differential equations (PDEs) at scale.

2. Useful as a framework for other stencil computations.

3. It can generate, when compared with Python, lower-level nested for loops (possibly irregular ones).

4. Given its Python compatibility, it allows high productivity and integration with scientific libraries already well established (such as Matplotlib, Numpy, Scipy, TensorFlow, etc.).

5. Explores the potential of On-The-Fly and Just-In-Time (JIT) code generation.

6. Also uses compiler technology to apply multi-step code optimization.

7. At last, can reduce the floating-point (FLOP) operations at a higher level by applying symbolic transformations.

Devito is highly based on FEniCS [7] and Firedrake [10]. FEniCS allows the solution of differential equations through Finite Element (FE), using for that the UFL language [2], which is a domain-specific language for weak formulation of partial differential equations. Firedrake also allows the solution of differential equations through the Finite Element (FE) method, also using UFL. Firedrake follows the principle of graceful degradation by providing a simple low-level API that allows the user to overcome the abstraction level whenever more complex operations are necessary. But different from those two frameworks, Devito does not have a formal specification language, such as UFL. However, it does support the graceful degradation principle, uses Python, incorporates a symbolic engine, and encompasses compiler technology for code generation. Despite all that, Devito also provides multiple level code optimization, such as FLOP reduction (through common sub-expression elimination, factorization, and code motion), data locality exploration (by searching for reductions through consecutive loop iterations), and parallelism (such as loop blocking and vectorization).

2.1.1 Specifying a Finite - Difference Scheme

Devito allows the user to express finite difference and generic stencil operations by using mathematical notation. To support such expression, the framework relies upon SymPy [9], which is a flexible Python library used to represent symbolic mathematics. Below we have the general steps followed by almost any Python script specified by Devito:

1. Defines the computational Grid (the domain)

2. Defines the symbolic functions (which is the real place where computations happen)

3. Defines the symbolic equations, which implies the data update scheme

4. Creates the Operator, responsible for converting the reference code into C code for the data update scheme specified by the previous symbolic equations

5. Finally compiles and executes the generated code, using for that the JIT procedure, which generates a shared library. Then loads this library into the Python environment so that it can call the kernel function later

To illustrate this general procedure, let's take the acoustic wave equation [11] from Eq. 1 below:

$$m(x, y, z) * \frac{\partial^2 u(x, y, z, t)}{\partial t^2} - \nabla^2 u(x, y, z, t) = q_s \quad (1)$$

In order to solve this equation, the following general steps must be specified:

1. The first step consists in defining the computational grid:

$$g = Grid(shape = (nx,), origin = (ox,), extent = (sx,))$$

   (a) shape: is the number of grid points in each one of the spatial dimensions (only one dimension is specified, but up to three are allowed).
   (b) origin: the origin of the coordinated axis
   (c) extent: defines the grid extension in physical units

2. The second step for the solution consists of defining the square slowness of the velocity (m), the wavefield (u), and the source terms (q_s):

$$m = Function(name =' m', grid = g)$$
$$u = TimeFunction(name =' u', grid = g, space_order = 2, time_order = 2)$$
$$q = SparseTimeFunction(name =' g', grid = g, nt = nt, coordinates = coord)$$

   (a) Function: represents a discreet function that changes only spatially, such as the wave velocity.
   (b) TimeFunction: represents a discreet function that changes both spatially and in time, such as the wavefields.
   (c) SparseTimeFunction: represents a discreet function that is defined only in a sub-set of the grid points, such as the seismic source terms.

3. The third step for the solution involves the symbolic equations definition:

$$wave_eq = m * u.dt2 - u.laplace$$
$$injection = q.inject(field = u.forward, expr = dt * *2 * q/m)$$
$$stencil = Eq(u.forward, solve(wave_eq, u.forward), region = Interior)$$

4. The fourth step comprises the operator creation: $op = Operator([stencil, injection])$

5. The last step, which executes the wave operator: $op.apply(dt = 1e - 3)$

In summary, the Python code for that wave solution is depicted in the Listing 1 below. Then the C code generated by Devito is also shown in the Listing 2 below. It is possible to see that Devito transformed a simple set of 26 lines of code into lower-level code with 40 more lines, thus showing that the framework considerably facilitates the user job.

```python
ox = 0 # grid origin
nx = 100 # number of grid points in 'X' direction
sx = 1 # extent in physical units of the 'X' axis
dt = 1e-3 # time discretization step
nt = 10 # total number of time steps

coordinates = [[0]] # source coordinates

# first step
g = Grid(shape=(nx, ), origin=(ox, ), extent=(sx,)) #computational grid

# second step
m = Function(name='m', grid=g) # physical parameter
u = TimeFunction(name='u', grid=g, space_order=2, time_order=2, save=nt) # wavefield
q = SparseTimeFunction(name='g', grid=g, nt=nt, npoint=1, coordinates=coordinates) # source

# third step
wave_eq = m * u.dt2 - u.laplace # acoustic wave equation
stencil = Eq(u.forward, solve(wave_eq, u.forward), region='interior') # stencil
injection = q.inject(field=u.forward, expr=dt**2 * q / m) # injected energy equation into wavefield

# fourth step
op = Operator([stencil, injection]) # operator

# fifth step: compile and execute
op.apply(dt=dt)
```

Listing 1: Python example - Wave Solution

```c
#include "stdlib.h"
#include "math.h"
#include "sys/time.h"

struct dataobj
{
  void *restrict data;
  int * size;
  int * npsize;
  int * dsize;
  int * hsize;
  int * hofs;
  int * oofs;
} ;

int Kernel(const float dt, struct dataobj *restrict g_vec, struct dataobj *restrict g_coords_vec, const float h_x
    , struct dataobj *restrict m_vec, const float o_x, struct dataobj *restrict u_vec, const int x_M, const int
    x_m, const int p_g_M, const int p_g_m, const int time_M, const int time_m) {
  float (*restrict g)[g_vec->size[1]] __attribute__ ((aligned (64))) = (float (*)[g_vec->size[1]]) g_vec->data;
  float (*restrict g_coords)[g_coords_vec->size[1]] __attribute__ ((aligned (64))) = (float (*)[g_coords_vec->
    size[1]]) g_coords_vec->data;
  float (*restrict m) __attribute__ ((aligned (64))) = (float (*)) m_vec->data;
  float (*restrict u)[u_vec->size[1]] __attribute__ ((aligned (64))) = (float (*)[u_vec->size[1]]) u_vec->data;

  for (int time = time_m, t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3); time <= time_M; time += 1,
    t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3))
  {
    for (int x = x_m; x <= x_M; x += 1)
    {
      u[t2][x + 2] = pow(dt, 2)*(-(-2.0F*u[t0][x + 2]/pow(dt, 2) + u[t1][x + 2]/pow(dt, 2))*m[x + 1] + u[t0][x +
    1]/pow(h_x, 2) - 2.0F*u[t0][x + 2]/pow(h_x, 2) + u[t0][x + 3]/pow(h_x, 2))/m[x + 1];
    }
    for (int p_g = p_g_m; p_g <= p_g_M; p_g += 1)
    {
      float posx = -o_x + g_coords[p_g][0];
      int ii_g_0 = (int)(floor(posx/h_x));
      int ii_g_1 = (int)(floor(posx/h_x)) + 1;
      float px = (float)(-h_x*(int)(floor(posx/h_x)) + posx);
      if (ii_g_0 >= x_m - 1 && ii_g_0 <= x_M + 1)
      {
        u[t2][ii_g_0 + 2] += 1.0e-6F*(1 - px/h_x)*g[time][p_g]/m[ii_g_0 + 1];
      }
      if (ii_g_1 >= x_m - 1 && ii_g_1 <= x_M + 1)
      {
        u[t2][ii_g_1 + 2] += 1.0e-6F*px*g[time][p_g]/(h_x*m[ii_g_1 + 1]);
      }
    }
  }

  return 0;
}
```

Listing 2: C example - Wave Solution

2.1.2 Point regions

Another important issue to consider is the point regions. In the Devito framework, a function distinguishes between three type of regions:

1. **Domain:** the computational domain of the function and it is inferred from the input grid.
2. **Halo:** the subset of points around the domain region.
3. **Padding:** set of grid points around the halo region, allocated for optimization/performance purposes (ex. data alignment).

2.1.3 Compiler

In Devito, the "Operator" object previously described is one of the most important entities, being responsible for three major tasks: Lower level code generation, JIT compilation, and Code Execution. The operator input consists of one or more symbolic equations. In the generated code, those equations are implemented inside nested loops (with proper depth and extension). The "Operator" also accepts substitution rules, such that it can substitute symbols by constant values. Furthermore, it accepts optimization levels by the user, which are processed by the Devito Symbolic Engine (DSE) and by the Devito Loop Engine (DLE). This process of dynamically converting equations from high-level code (Python) to lower code (C) to then execute them consists of multiple steps, following described:

1. **Equations Lowering** This process can be split into three sub-steps:
   (a) Indexification: the set of input equations of the operator typically involves one or more indexed functions. The "indexification" consists in converting such objects into real arrays. Still, one array keeps a reference to its original function.
   (b) Substitution: The imposed substitution rules are applied here. Those substitutions (of symbols by constant values) must be applied for each literal appearing at the input equations, such as the grid spacing and the symbols. The input symbols in which no rule is provided must be given at execution time.
   (c) Domain-Alignment: applies a shift in the array accessing, where the "halo" and "padding" regions are not null. For instance, u[t][x] becomes u[t][x + 2] for two points in the "halo" region.
2. **Local Analysis:** The lowered equations are inspected in "isolation" so that important information can be obtained for the construction/execution of the operator. The metadata below is returned:
   (a) Input and output of functions
   (b) Dimensions, which are topologically ordered based on how they appear in the various indexes of arrays of functions.

   (c) Two notable spaces: the iteration space (ISpace) and the data space (SPace)

3. **Clustering:** this process groups equations by similar properties between them, such as the ones having the same ISpace and the same flow of control.

   (a) Iteration Direction: from global analysis, a new ISpace is given to each equation. Iteration directions given an "arbitrary" mark can be forced to take the forward (+) or the backward (-) direction.

   (b) Grouping: this performs clustering, checks ISpaces, as well as handles the flow of control.

4. **Symbolic Optimization:** this is a series of passes (common sub-expression and rewrite procedures) to reduce the number of operations in each cluster. The output of this step is a new ordered sequence of clusters.

5. **IET Construction:** the "Iteration / Expression Tree (IET)" is generated here, out of the previous intermediate representation. This IET is an abstract syntax tree, where iterations and expressions are the most important nodes. For instance, equations are wrapped inside Expression nodes, while loops are encapsulated within an Iteration node. At some point, the clusters are given their own nested loop (iteration node).

6. **IET Analysis:** Properties of the iteration, such as parallel, sequential, and vectorizable regions are extracted from the IET in this analysis. The extracted information is incorporated into relevant nodes of the tree, which will be used in future passes to optimize the final code.

7. **IET Optimization:** This is an important step since it performs performance optimizations to transform the IET. Multiple loop optimizations are applied, such as SIMD vectorization, loop blocking, shared memory, parallelism through OpenMP, prefetching, etc.

8. **Synthesis, Dynamic Compilation, and Execution:** Here the variable declaration, as well as header lines, are injected into the IET. Also, profiling instrumentation to collect runtime and declarations assuring that they appear as close as possible to where they are used are inserted into the IET. A proper "visitor" for the IET inspects the nodes and generates a CGen Tree, which at the end is converted into a C code string and written to a file. This file is typically written to a Devito cache, which is then compiled by JIT and therefore converted into a shared object (.so). This object is at some point loaded into the Python environment. Given that the compiled code has a default entry point, a function called "Kernel", the python code can finally call the generated code.

This was a brief explanation about the Devito Framework. For more details, the reader is advised to read the interesting work of [8].

2.1.4 Specializing operators through backends

In Devito, the "backend" is a mechanism used to specialize operator passes during the code generation, while still keeping the software modularity of the system. So far, Devito provides support to multiple backends. For instance, the core backend, which is the default, uses the Domain Loop Engine (DLE) to optimize loops. Another one is the Yask backend, which uses the Yask stencil compiler to generate optimized C++ code for Intel Xeon Architectures. Finally, the OpenMP specialization is already available, allowing "for loops" to be easily parallelized on either CPUs or GPUs (target devices) under the context of shared memory systems. The Message Passing Interface (MPI) is also supported by the framework, allowing executions on distributed memory systems, such as local clusters and cloud environments.

2.2 OpenMP

The Open Multi-Processing (OpenMP) [4] is an Application Programming Interface (API) based on directives to easily provide parallel programming to the developer. Its main dependency relies upon shared memory systems, such as multithreading and multi-core executions. Though restricted to this context, the benefits in its usage are multiples. For instance, the developer can easily parallelize for loops (parallel, parallel for, and task). The developer can also specify synchronization schemes (locks, dependencies between tasks, and barriers), scheduling mechanisms (static, dynamic, taskyeld, etc.), atomic operations / critical sections to prevent data racing, and even offloading the execution to GPU devices by using target directives. With all that, the developer does not have to worry so much about the parallelization intricacies, such as load balance, data distribution between threads, and difficult programming interfaces (CUDA / OpenCL), given that OpenMP deals with all that stuff. But using only a shared memory system may be an insufficient solution to meet the execution requirements, being necessary to rely on more robust mechanisms, such as multi-node systems.

2.3 MPI

The standardized Message Passing Interface (MPI) [3] may be a possible alternative to overcome such OpenMP issues, establishing the conventions to be followed by a multi-node system so that they can exchange messages through a distributed memory scheme. By that, MPI specifies the interface, protocol, and semantics to be used to accomplish such exchange. Though quite vulnerable to fault incidents, which a single node can hold or break the whole execution, the common implementations of the MPI standardization provide scalable and reliable high-performance executions, thus justifying its huge adoption by the industry. And that good features become even more attractive under the scenarios that Devito usually deals with, especially the huge seismic problems solved by the framework, such as Full Waveform Inversion (FWI) and Reverse Time Migration (RTM).

However, directly dealing with MPI poses multiple challenges and concerns to the developer. For instance, he must not only be worried about solving the problem but also

with the parallelization and the efficiency of the data exchange. In other terms, the programmer must know about everything. As a consequence, the increase in concerns makes the overall program harder to maintain. And this invariably leads to poor usage of new software/hardware resources, which sadly prevents the code from being as much flexible as it should be. A more desirable solution certainly should be the one that delegates the explicit data exchange to a third-party library, thus allowing the user to be worried only about what matters.

2.4 Ompcluster

OmpCluster [1] is a great candidate to fill the gap left by both OpenMP and MPI. This runtime system allows the programmer to use the OpenMP syntax to automatically offload the execution to multiple cluster regions, therefore extending the OpenMP runtime to distributed architectures. To provide such inter-node communication, the system uses MPI under the hood so that the developer does not have to program a single line of MPI code. More than that, OmpCluster automatically maps data to computing regions and also schedules the generated tasks so that they can properly be executed at the correct time - assuming here a proper dependency scheme provided by the user. Thus, all the load balancing is managed by OmpCluster, and the programmer only has to worry about the specific tasks that should be generated and how to correctly map the data.

The listing 3 below shows a parallelized OmpCluster code for the canonical matrix multiplication algorithm. In this, it is possible to see that in order to generate tasks, the developer had to split the C matrix computation into row blocks, meaning that each node is responsible for computing the matrix multiplication of a subset of rows from C. This is possible since the matrix multiplication has the embarrassingly parallel property for each cell of the matrix. Therefore, the issue of most concern for the programmer is to keep the parallelization overhead as low as possible and the data locality as coherent as it can be. Given that the matrix domain is already split, thus enabling the task generation (lines 17-20), the developer only has to specify which part of the data should be mapped to some target node, so that the multiplication can happen. Interesting to note is that the correct execution relies heavily on dependencies between tasks, which secures that the mapped data will be available when needed for computation (line 21). Also, the same way that the data was offloaded (lines 11-15), it is also brought back to the host node (the one executing the single thread and where the data was initially allocated) at the end of the computation (line 23); therefore securing that the correct output will be available at the end of the whole execution.

```
1  void multiply(float *a, float *b, float *c, unsigned size) {
2    int numBlocks = 2;
3    int sizeBlock = (size + numBlocks - 1) / numBlocks;
4
5    #pragma omp parallel
6    #pragma omp single
7    {
8      #pragma omp target enter data map(to: b[: size * size]) nowait
9
10     for (int blockIdRow = 0; blockIdRow < numBlocks; ++blockIdRow) {
11       #pragma omp target enter data map(to: a[blockIdRow * sizeBlock * size : sizeBlock * size]) \
12                                      depend(out: a[blockIdRow * sizeBlock * size]) nowait
13
14       #pragma omp target enter data map(to: c[blockIdRow * sizeBlock * size : sizeBlock * size]) \
15                                      depend(out: c[blockIdRow * sizeBlock * size]) nowait
```

```
16
17      #pragma omp target map(a[blockIdRow * sizeBlock * size : sizeBlock * size]) \
18                         map(c[blockIdRow * sizeBlock * size : sizeBlock * size]) \
19                         depend(in: a[blockIdRow * sizeBlock * size]) \
20                         depend(inout: c[blockIdRow * sizeBlock * size]) nowait
21      multiplyMatrixBlock(a, b, c, size, blockIdRow, sizeBlock);
22
23      #pragma omp target exit data map(from: c[blockIdRow * sizeBlock * size : sizeBlock * size]) \
24                                   depend(in: c[blockIdRow * sizeBlock * size]) nowait
25    }
26  }
27 }
```

Listing 3: Matrix Multiplication parallelized with OmpCluster

3 Backend Specialization

To effectively specialize the backend with OmpCluster, multiple modifications had to be accomplished. First of all, Devito allows the user to specify the type of compiler to be used (gcc, clang, etc.). And for each compiler it is possible to specify the set of flags to be passed to the compilation. Given that the only compiler supported by OmpCluster is Clang, the first step followed was to modify the 'ClangCompiler' class (included at the file "devito/arch/compiler.py") so that the user could specify the correct OmpCluster backend and, thus, inform it during compilation to correctly use the flags "openmp" and "-fopenmp-targets=x86_64-pc-linux-gnu". In these step, a new platform had to be generated to allow the user to specify during execution time the correct backend. After finishing this step, the whole OpenMP Target backend (GPU backend) was clonned and, only after that, the modifications started. The most important files introduced were:

1. OmpCluster Operator File (at 'devito/core/ompcluster.py')
2. OmpCluster Pragma File (at 'devito/passes/iet/parpragmaompcluster.py')
3. OmpCluster Orchestration File (at 'devito/passes/iet/ompclusterorchestration.py')
4. OmpCluster Directive Dictionary File (at 'devito/passes/iet/languages/ompcluster.py')

3.1 OpenMP Target Code

Albeit those new files were introduced, the optimized version of the base code (Listing 4 and 5), which consists of summing one to the "u" array at every timestep, was exactly equal to the one produced by the OpenMP Target backend, depicted in the Listing 6 below. Even though this OpenMP Target Code is not focusing completely on the OmpClusters specific directives, some important elements are already introduced to this first optimized code. For instance, task creation is explicit in line 13, although is completely different than the directive expected by OmpCluster, which will rely upon domain decomposition over the X dimension to effectively produce tasks. Another aspect also introduced is the data management, depicted in lines 9, 23, and 24, which shows that the array information is correctly being mapped to the device. In this case, although changes are necessary, this highly resembles the expected directives used with OmpCluster. As a matter of fact, those

examples illustrate the two passes specialized by the Devito backend, which is: 1) Introduction of parallelism to code (through parallel for, omp target, etc.), and 2) Introduction of data management pragmas to code (omp target enter data map, omp target exit data map, etc.). Therefore, they were the main focus during the OmpCluster backend specialization.

```python
from devito import *
from devito.tools import pprint
import matplotlib.pyplot as plt

configuration['language'] = 'C'
configuration['compiler'] = 'clang'

grid = Grid(shape=(4, 4))

u = TimeFunction(name="u", grid=grid, space_order=1, time_order=1)
u.data[0, :, :] = 1.

eq = Eq(u.forward, u + 1)

op = Operator([eq,], platform='ompclusterX', opt="noop")

print(op)
```

Listing 4: Python Code Example 1

```c
int Kernel(struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m,
    const int y_M, const int y_m)
{
  float (*restrict u)[u_vec->size[1]][u_vec->size[2]] __attribute__ ((aligned (64))) = (float (*)[u_vec->size
    [1]][u_vec->size[2]]) u_vec->data;

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 =
    (time + 1)%(2))
  {
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        u[t1][x + 1][y + 1] = u[t0][x + 1][y + 1] + 1;
      }
    }
  }

  return 0;
}
```

Listing 5: Non-Optimized Generated C code for the Python Code Example 1

```c
int Kernel(struct dataobj *restrict u_vec, struct dataobj *restrict v_vec, const int time_M, const int time_m,
    const int x_M, const int x_m, const int y_M, const int y_m, const int devicerm, const int deviceid)
{
  if (deviceid != -1)
  {
    omp_set_default_device(deviceid);
  }
  float (*restrict u)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->data;

  #pragma omp target enter data map(to: u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])

  for (int time = time_m; time <= time_M; time += 1)
  {
    #pragma omp target teams distribute parallel for collapse(2)
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        u[time + 1][x + 2][y + 2] = u[time][x + 2][y + 2] + 1;
      }
    }
  }

  #pragma omp target update from(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])
  #pragma omp target exit data map(release: u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]]) if(devicerm)

  return 0;
}
```

Listing 6: Optimized OpenMP Target code based on the Non-Optimized code version

3.2 OmpCluster code based on OpenMP Target Code

Before jumping directly to the Domain Decomposition version of the code, which would be the desired one, a previous step was considered: the simple generation of OmpCluster based task. In this scheme, it is possible to see on Listing 7 (lines 13-14) that the mapping step belonging to the Data Management pass was introduced during the Parallelism step. This was necessary since OmpCluster requires tasks to specify their data space, as well as their dependencies (which are not explicitly being shown in the code). But surely, the current code is not optimal, given that only one task is being generated by time-step, while N is required (being N the number of active nodes in the system).

```
1 int Kernel(struct dataobj *restrict u_vec, struct dataobj *restrict v_vec, const int time_M, const int time_m,
     const int x_M, const int x_m, const int y_M, const int y_m)
2 {
3   #pragma omp parallel
4   {
5     #pragma omp single
6     {
7       float (*restrict u)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->
      data;
8
9       #pragma omp target enter data map(to: u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]]) nowait
10
11      for (int time = time_m; time <= time_M; time += 1)
12      {
13        #pragma omp target \
14                map(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]]) nowait
15        for (int x = x_m; x <= x_M; x += 1)
16        {
17          for (int y = y_m; y <= y_M; y += 1)
18          {
19            u[time + 1][x + 2][y + 2] = u[time][x + 2][y + 2] + 1;
20          }
21        }
22      }
23
24      #pragma omp target exit data map(from: u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])
25    }
26  }
27
28  return 0;
29 }
```

Listing 7: Optimized OmpCluster code after small changes to OpenMP Target code

3.3 Domain Decomposition

The optimized code presented on Listing 8 highly resembles the one expected to properly work with OmpCluster. For example, the domain decomposition is finally introduced, such that a new for block (line 11) is completely injected into code, responsible for partitioning the X domain (which extends from x_m to x_M) into 'numBlocks' elements. Also, the map and dependencies specified through the pragmas are basically completed; therefore allowing a proper execution of the code. However, the presented code has a simple specification that prevents it from compiling: the map's shown in Lines 22 and 23 are using the same array base, which is not allowed by the current versions of the OpenMP specifications. Because of that, this code is not possible to be used without modifications, and a new pointer cast must be introduced to overcome such an issue.

```
1 int Kernel(struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m,
     const int y_M, const int y_m, int numBlocks_M)
2 {
3   #pragma omp parallel
4   {
5     #pragma omp single
```

```
6     {
7       float (*restrict u)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->
      data;
8
9       for (int time = time_m; time <= time_M; time += 1)
10      {
11        for (int blockId = 0; blockId <= fmin(numBlocks_M - 1, x_M - x_m - 1); blockId += 1)
12        {
13          int blockSize = fmax(1, (numBlocks_M + x_M - x_m)/numBlocks_M);
14
15          int inix = blockId*blockSize + x_m + 2;
16          int stepx = fmin(x_M, blockSize*(blockId + 1) + x_m - 1) - inix + 3;
17
18          #pragma omp target enter data map(to: u[time:1][inix:stepx][0:u_vec->size[2]]) \
19                                        map(to: u[time + 1:1][inix:stepx][0:u_vec->size[2]]) \
20                                        depend(in: u[time][inix][0]) depend(out: u[time][inix][0]) depend(out: u[
      time + 1][inix][0])  nowait
21
22          #pragma omp target map(u[time:1][inix:stepx][0:u_vec->size[2]]) \
23                             map(u[time + 1:1][inix:stepx][0:u_vec->size[2]]) \
24                             depend(in: u[time][inix][0]) depend(out: u[time + 1][inix][0]) nowait
25          {
26            for (int x = blockId*blockSize + x_m; x <= fmin(x_M, blockSize*(blockId + 1) + x_m - 1); x += 1)
27            {
28              for (int y = y_m; y <= y_M; y += 1)
29              {
30                u[time + 1][x + 2][y + 2] = u[time][x + 2][y + 2] + 1;
31              }
32            }
33          }
34
35          #pragma omp target exit data map(from: u[time + 1:1][inix:stepx][0:u_vec->size[2]]) depend(out: u[time
      + 1][inix][0]) nowait
36        }
37      }
38    }
39  }
40
41  return 0;
42 }
```

Listing 8: Optimized OmpCluster code after Domain Decomposition

3.4 Pointer Cast Substitutions

To finally solve the previous problem, pointer cast substitutions must be introduced to code. Though the code becomes considerably more difficult to generate and understand, the issue in question is finally solved. The final result is presented in the Listing 9. As it can be seen, two new casts were introduced: "u0" and "u1". This was necessary because with that the map's in lines 34 and 35 would have different bases, therefore, allowing the code to correctly compile and execute. As a counterpart, the vector indexations (line 40) had to be also substituted and pointer swaps were necessary to be introduced (lines 44-46), which is an annoyance but still acceptable. With that code, the backend specialization is finally finished for the example presented.

```
1  int Kernel(struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m,
      const int y_M, const int y_m, int numBlocks_M)
2  {
3    #pragma omp parallel
4    {
5      #pragma omp single
6      {
7        float (*restrict u)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->
      data;
8        float (*restrict u0)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) &u
      [0][0][0];
9        float (*restrict u1)[u_vec->size[1]][u_vec->size[2]] = (float (*)[u_vec->size[1]][u_vec->size[2]]) &u
      [1][0][0];
10
11       for (int blockId = 0; blockId <= fmin(numBlocks_M - 1, x_M - x_m - 1); blockId += 1)
12       {
13         int blockSize = fmax(1, (numBlocks_M + x_M - x_m)/numBlocks_M);
```

```
14          int inix = blockId*blockSize + x_m + 1;
15          int endx = fmin(x_M, blockSize*(blockId + 1) + x_m - 1);
16          int stepx = endx - inix + 2;
17
18          #pragma omp target enter data map(to: u0[0:1][inix:stepx][0:u_vec->size[2]]) depend(out: u0[0][inix][0])
      nowait
19          #pragma omp target enter data map(to: u1[0:1][inix:stepx][0:u_vec->size[2]]) depend(out: u1[0][inix][0])
      nowait
20        }
21
22        for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2),
      t1 = (time + 1)%(2))
23        {
24          for (int blockId = 0; blockId <= fmin(numBlocks_M - 1, x_M - x_m - 1); blockId += 1)
25          {
26            t0 = 0;
27            t1 = 0;
28            int blockSize = fmax(1, (numBlocks_M + x_M - x_m)/numBlocks_M);
29            int inix = blockId*blockSize + x_m + 1;
30            int endx = fmin(x_M, blockSize*(blockId + 1) + x_m - 1);
31            int stepx = endx - inix + 2;
32
33            #pragma omp target  \
34                map(u0[0:1][inix:stepx][0:u_vec->size[2]]) depend(inout: u0[0][inix][0]) \
35                map(u1[0:1][inix:stepx][0:u_vec->size[2]]) depend(inout: u1[0][inix][0]) nowait
36            for (int x = inix - 1; x <= endx; x += 1)
37            {
38              for (int y = y_m; y <= y_M; y += 1)
39              {
40                u1[t1][x + 1][y + 1] = u0[t0][x + 1][y + 1] + 1;
41              }
42            }
43          }
44          u = u0;
45          u0 = u1;
46          u1 = u;
47        }
48
49        for (int blockId = 0; blockId <= fmin(numBlocks_M - 1, x_M - x_m - 1); blockId += 1)
50        {
51          int blockSize = fmax(1, (numBlocks_M + x_M - x_m)/numBlocks_M);
52          int inix = blockId*blockSize + x_m + 1;
53          int endx = fmin(x_M, blockSize*(blockId + 1) + x_m - 1);
54          int stepx = endx - inix + 2;
55
56          #pragma omp target exit data map(from: u0[0:1][inix:stepx][0:u_vec->size[2]]) depend(in: u0[0][inix][0])
      nowait
57          #pragma omp target exit data map(from: u1[0:1][inix:stepx][0:u_vec->size[2]]) depend(in: u1[0][inix][0])
      nowait
58        }
59      }
60    }
61
62    return 0;
63  }
```

Listing 9: Optimized OmpCluster code after Pointer Cast Substitution

4 Discussions and Limitations

4.1 Numerical Experiments

To validate the proposed backend, numerical experiments on four different 2D datasets were performed and their total running time was measured and shown in Table 1. Though the number of datasets is small, it is possible to see that the program is capable of almost achieving the maximum speedup, which is 2 since only two worker nodes are being used. The hardware used to execute all experiments runs Red Hat 4.8.5-39 and has an Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz coupled with 187 GB of RAM. In total, three nodes (one head node and two workers) were considered through the entire execution. The C programs were automatically generated by the specialized OmpCluster backend of Devito, using for that the base script from Listing 4. To compile the code, the flags "-fopenmp", "-fopenmp-targets=x86_64-pc-linux-gnu" were used.

Table 1: Datasets tested in this report

Dataset	**Nx**	**Ny**	**Nz**	**Nt**	Serial Exec. (s)	Parallel Exec. (s)	Speedup
1	100	100	-	1000	0.33	0.32	1.03x
2	500	500	-	1000	2.01	1.17	1.72x
3	1000	1000	-	1000	7.01	3.92	1.79x
4	5000	5000	-	1000	170.42	86.21	1.98x

Table 1 above shows a comparison between a serial execution and a parallel execution (considering 3 nodes) on four different datasets. As it can be seen, for smaller datasets the obtained speedup is negligible when comparing the serial and parallel versions. However, as the datasets increases in size, the speedup becomes more significant, almost achieving 2x for the last row. Although those results express the scalable potential of OmpCluster under the implemented backend context, more results considering more nodes surely are to be expected in future works. But for the time being, the presented results are sufficient since the main focus of this report is code generation and not the best performance.

4.2 No direct execution

Although a first scalable code was presented to Devito, some major restrictions were considered during the development of this new OmpCluster backend. First of all, the generated code depicted in Listing 9 is not possible to be loaded as a shared library and then called to execute during execution time. This happens due to LLVM restrictions which, during the execution time, do not properly differentiate the worker nodes from head nodes, therefore forcing all the MPI nodes to function as head nodes. However, this scheme is totally inviable for execution since the runtime system requires to have only one head node and multiple workers. As a workaround, code continued to be generated by Devito, however, its compilation and calling happened outside the Python environment. Though this does not solve the issue, leaving the problem as a huge drawback for the proposed backend, it is still possible to get an automatically generated code, which still facilitates the user job. However, some great efforts should be devoted in the future to more properly deal with such issues.

4.3 Pointer Substitution

Another restriction imposed by the specialization was the pointer cast substitution applied from Listing 8 to Listing 9. These small substitutions certainly made the code harder to maintain, more difficult to understand, and more susceptible to the introduction of bugs during the code generation. Although multiple concerns appeared with this choice, this was a necessary evil since the current OpenMP versions would have not allowed the code to compile in its pure version from Listing 8. However, this should be a point of great concern in the future, and more efforts should be made to find better alternatives to this pointer substitution.

4.4 Data Exchange

Finally, we have the data exchange issue. To illustrate that, let's analyze the following four images. In figure 1 we have the Function domain, where computation actually happens. But in order to split computation throughout the nodes, this domain must be partitioned. For instance, in a system with two nodes, the first one would get the first half of the domain (A) to process and the second node would get the other half (B), exactly as shown by figure 2. This implies that the second node will not have access to the first sub-domain A and the first node will not have access to the second sub-domain B. However, this is a huge problem since Finite-Difference schemes rely on neighbor points to calculate numerical derivatives (see figure 4, where the derivative of grid point 0 is using neighbor points 1, 2, 3, and 4). And if those neighbor points are not residing in the node doing the computation, then either a segmentation fault will happen or incorrect values may be fetched from memory. To fix that issue, points residing near the domain division must be sent to other computing nodes in order to them correctly compute the derivatives. Figure 3 illustrates this process, where points residing in node A are sent to node B and points from B are sent to node A before computation happens. Assuming that this step is completed, the execution is safe to deliver correct results. Therefore, it is paramount this problem is addressed in future works, given that Finite-Difference is at the core of Devito.

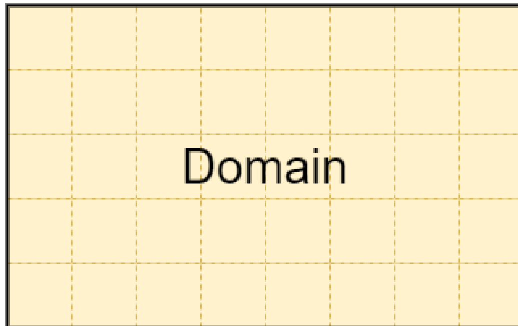

Figure 1: Domain Specification

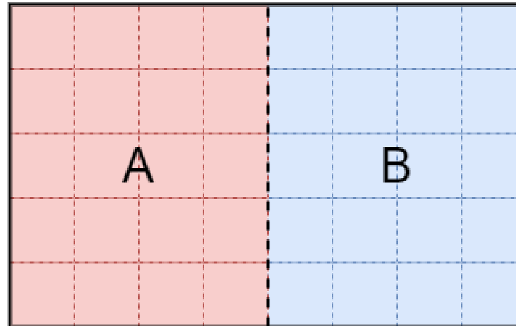

Figure 2: Domain Decomposition

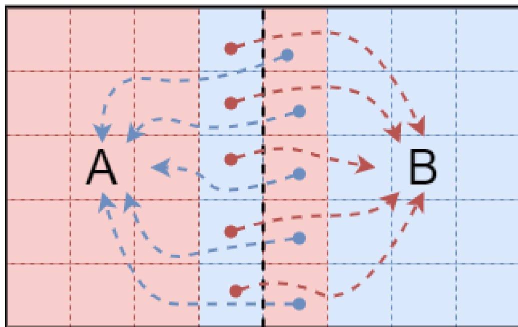

Figure 3: Data Exchange

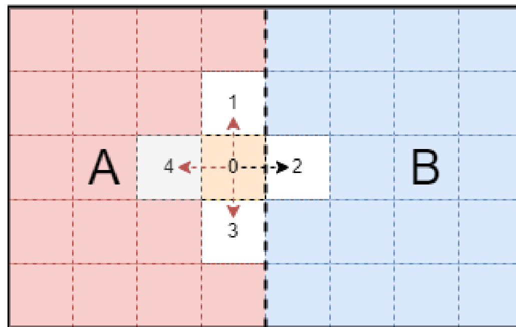

Figure 4: Finite-Difference Computation

5 Conclusion

In this work, it was presented a first approach toward specializing Devito's backend to use the runtime system OmpCluster. At first, the already existing OpenMP target backend was used as base work, given its similarity with the OmpCluster offloading mechanics. However, some major modifications were applied during the Parallelism and Data Management passes in order to adjust the code syntax with the one expected by the compiler stack. Promising results with multiple datasets have shown the backend robustness to automatically generate optimized OmpCluster code and the runtime system potential to scale the execution.

Although the satisfactory outcomes, multiple issues are left open to future works. For instance, the execution of the generated code outside of the Python environment considerably deviates from Devito's philosophy of easing the execution to final users. Though the code generation alleviates the user job, this certainly should be dealt with in the future. Another aspect is the imposed pointer substitution used to overcome the LLVM compiler restrictions. Albeit it does not introduce any harm at a first sight, this should certainly be fixed in the future since it makes code more difficult to understand and maintain, as well as it increases the possibility of introducing bugs to the array accessing mechanics already well established and tested by previous Devito developers. Finally, the last open issue is the Data Exchange, which was not approached in this work. Certainly, this is a thing to address in future work given that every Finite-Difference scheme used to solve a problem will rely on accessing data elements distributed throughout different nodes. And given that Finite-Difference schemes are at the core of Devito, it is paramount that the specialized backend offers support to an effective computation of this procedure. Only after solving those three issues, the new backend will be effectively ready to use.

Therefore, good results are obtained, but some issues are still necessary to be solved. Nevertheless, a first approach was provided to use the attractive properties of OmpCluster, which in the future will hopefully help the developer's work to produce easier and more maintainable code to that interesting framework that is Devito.

References

[1] Ompcluster. `https://ompcluster.gitlab.io/`. Accessed: 2021-12-10.

[2] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language. *ACM Transactions on Mathematical Software*, 40(2):1–37, February 2014.

[3] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218. Birkhäuser Basel, 1994.

[4] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[5] Jasper Denkers. A longitudinal field study on creation and use of domain-specific languages in industry. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, August 2019.

[6] Sanjeev Kumar. Finite difference method: A brief study. *SSRN Electronic Journal*, 2014.

[7] Anders Logg, Kent-Andre Mardal, and Garth Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*. Springer Berlin Heidelberg, 2012.

[8] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Transactions on Mathematical Software*, 46(1):1–28, April 2020.

[9] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[10] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake. *ACM Transactions on Mathematical Software*, 43(3):1–27, January 2017.

[11] Yanfei Wang and Wenquan Liang. Optimized finite difference methods for seismic acoustic wave modeling. In *Computational and Experimental Studies of Acoustic Waves*. InTech, January 2018.