POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Parallelization techniques for domain decomposition methods for the wave equation

Francesco Songia, Enrico Zardi

Project in Advanced Programming for Scientific Computing
Academic year 2022-23

FINIRE MAKEFILE CON EIGEN PATH e togliere moretest

Contents

1 Introduction

The focus of our work is in applying a new strategy for computing a numerical solution of the wave equation in a 1D bounded domain. The new idea is to consider time as a spatial variable and therefore moving from 1D to a 2D domain. On the resulting space-time grid we apply Discontinuos Galerkin finite element method.
The goal of all finite element methods is to construct a linear system $Au = f$, usually of large dimension, which solution u is an approximation of the solution of the original differential problem. In the NAPDE (Numerical Analysis Project for Partial Differential Equations) project we focused our attention on the construction of the linear system and subsequently in the application of various domain decomposition methods. The mathematical formulation of the problem and the linear system construction involve formal technicalities that we leave in Appendix A for the reader. From now on we assume the matrix A and the right hand side f, have been constructed and given to us to be solved.
Once the finite element method produces a linear system it is important to select good solvers that exploit the physics of the problem in question, in our case the wave equation hyperbolic nature. We tackle this large system with different iterative solvers in a domain decomposition setting.
From the implementation point of view we have created from scratch a C++ code trying to be more general as possible in order to allow future development of new solving policies for this kind of problem. We are in a domain decomposition framework and we exploit the possibility of parallelize the code. It comes naturally the need to define different policies to describe how subdomains are assigned to different cores. This will be the starting template parameter used to define different solvers and parallelization strategies.

1.1 Domain decomposition

Domain decomposition methods partition a problem's large domain into multiple smaller domain on which the problem is solved.
This strategy helps us in two ways. Firstly, we iteratively solve multiple smaller linear system instead of a single large linear system. Secondly, the linear systems are independent from one another, hence, ideal for parallelization. Moreover, in a domain decomposition setting it is necessary that the method is iterative, since each subdomain has to transmit its numerical information to the neighbours at each iteration.

For a generic linear system (1) the iterative Richardson method take the form of (2) where P is the preconditioner chosen.

$$A\boldsymbol{u} = \boldsymbol{f}. \tag{1}$$

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + P^{-1}(\boldsymbol{f} - A\boldsymbol{u}^n). \tag{2}$$

Domain decomposition methods can be seen as instances of the preconditioner P. In our scenario we propose two preconditioners, which rise from two domain decomposition approaches: Restricted Additive Schwarz (RAS) and Pipelined Restricted Additive Schwarz (RAS Pipelined) methods.

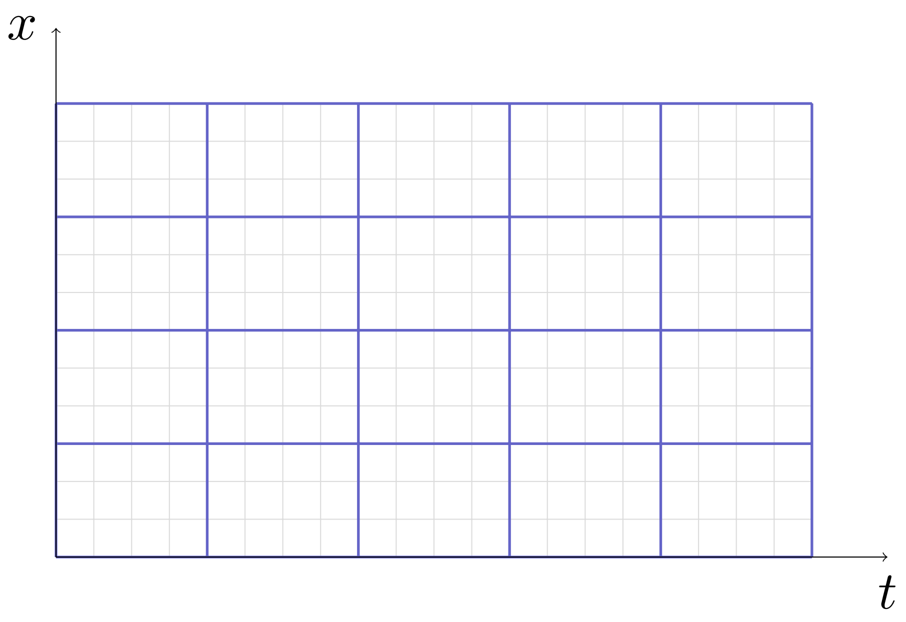


Figure 1: An example of domain subdivision without overlap.

1.2 RAS

Given a differential problem in a big domain Ω, we can subdivide Ω into M subdomains. To each of these subdomains we associate a local matrix A_j that is the restriction of matrix A on subdomain $j \in \{1, .., M\}$. The local matrix can be computed as $A_j = R_j A R_j^T$, where R_j and R_j^T are the restriction and prolongation rectangular matrix. In domain decomposition scenario if the subdomains are not a partition of Ω but there is overlap between the elements the solution in the overlapping region is a convex combination of the solutions of the subdomains concurring in the overlap. The coefficients of the convex combination are in the matrix $\tilde{R_j}^T$. Finally, the RAS preconditioner is:

$$P_{ras}^{-1} = \sum_{j=1}^{M} \tilde{R_j}^T A_j^{-1} R_j. \tag{3}$$

1.3 RAS Pipelined

RAS Pipelined is a variation of RAS method that exploits the time factor in the problem. Subdomains that have not reached a good approximation of the solution cannot transmit useful numerical information to subdomains further in time. The first subdomains in time will be the first to be solved, and the latest in time at last. Hence, by not solving all subdomains at each iteration we avoid useless computations as in RAS. Moreover, we avoid a transfer of inexact solutions and information coming from the late in time subdomains that cannot be solved during the first iterations. We consider only a subdomains' subset changing at each iteration, this subset is the sliding window S. The sliding window will first consider the first subdomains in time and then once convergence is reached it advances in time.

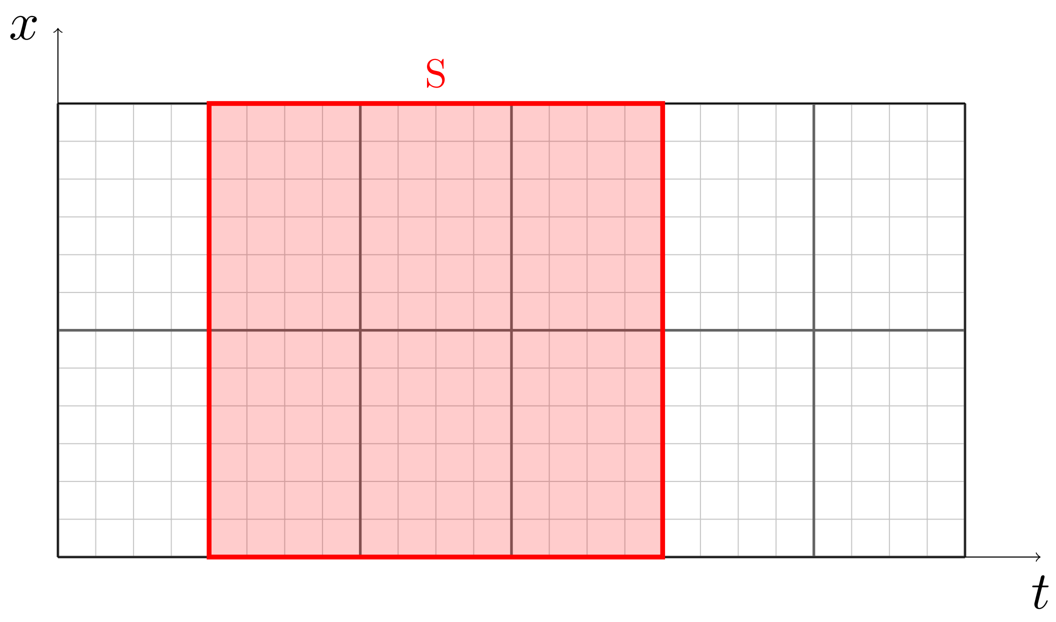


Figure 2: Sliding window in RAS Pipeline.

At first S is set at the the initial time, and it contains not just a vertical stride of the same time interval but some subdomains in time and all the space ones. Then we run the iterations (solving the subdomains inside S) until the residual in the left part of the zone S reaches a defined tolerance and then we move S in time. One could opt to just move to the right the window when the left part of S converges. This is a very conservative way to proceed since we can be blocked in one window for many iterations. To face this problem we implemented a more 'aggressive' version of Pipelined RAS: we do not wait until all the subdomains in the left part of S are solved but we update the right boundary of S after a defined number of iterations. However, we continue to check the left part and we shift it to the right when we achieved a fixed tolerance. In this way the zone of highlighted subdomains S is freer to move, we are not blocked in only one zone and the right side can evolve faster exploring all the subdomains. Then if it moves too fast it will be necessary to wait for all the previous subdomains to be exactly solved but certainly the method will solve less subdomains with respect to the original RAS. The use of this approach with a moving pipeline brings benefits to the parallel resolution. Compared to the previous method we assign at each thread much less subdomains to solve without lose quality in the convergence.

In RAS pipeline the preconditioner P is similar to RAS but the summation is only for the subdomains included in the sliding window.

$$P_{pipe}^{-1} = \sum_{j \in S} \tilde{R_j}^T A_j^{-1} R_j. \tag{4}$$

2 Parallelization

For both RAS and RAS Pipelined approach we propose different parallelization techniques. Dealing with domain decomposition methods together with parallelization determines the choice of the domain subdivision to be dependent on the number of cores available. Indeed, it is fundamental to know in advance the number of cores at disposal such that an ad hoc subdivision is established together with an ad hoc subdomains assignment policy. Therefore we will explain the parallelization techniques we propose in terms of subdomain divisions and number of cores available.

2.1 AloneOnStride

This parallelization technique prescribes a subdomain division in space equal to number of cores available and no restriction on the number of time subdomains. Each core is associated to a particular spatial interval and the whole time interval, one could call it a time stride. In our opinion it is a natural way to combine the subdivision strategy with cores availability.

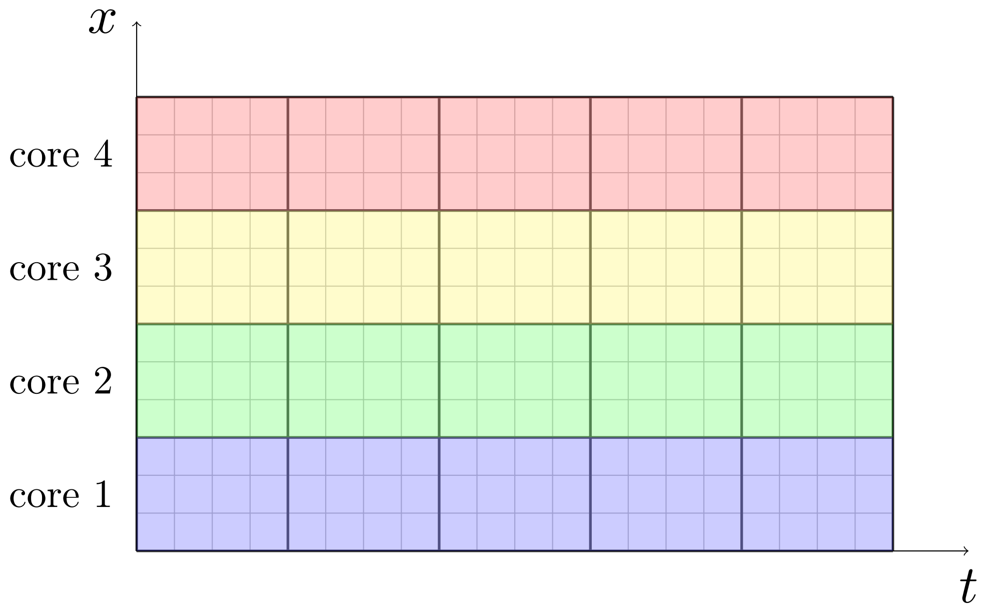


Figure 3: AloneOnStride parallelization.

Considering four cores available we present an instance of AloneOnStride parallelization. In Figure 3 a space-time rectangular domain with square elements has 4 subdivision in space, 5 in time and no overlapping. We assign to each core a set of subdomains, graphically grouped by having the same color.
In a RAS framework, at each iteration each core solves all and only the subdomains in its stride. Finally, the stride solution computed at each core is shared between the cores and summed to obtain $\boldsymbol{u}^{n+1}$. In a Pipeline framework every core has assigned a time stride but at each iteration only the subdomains in the sliding window are considered. Since we have to update the sliding window by looking at the convergence in the left side of S, each core checks the residual of its left-most current subdomain and communicates to the other cores; if convergence is reached the sliding window is shifted.
For a sequential algorithm, that does not exploit a multi core parallelization the `AloneOnStride` policy is used. That means that one single core solves all subdomains local problems at each iteration. We underline that, with this policy, we are able to treat together a sequential and parallel version of the code.

2.2 CooperationOnStride

This parallelization technique requires that the number of space subdivision be a divisor of the number of cores such that each stride is assigned to the same number of cores. For each stride one core is elected to be a principal core, responsible for computing the local solutions in each subdomain and the remaining ones in the same stride,referred as helpers, are used to perform linear algebra matrix multiplication in parallel.

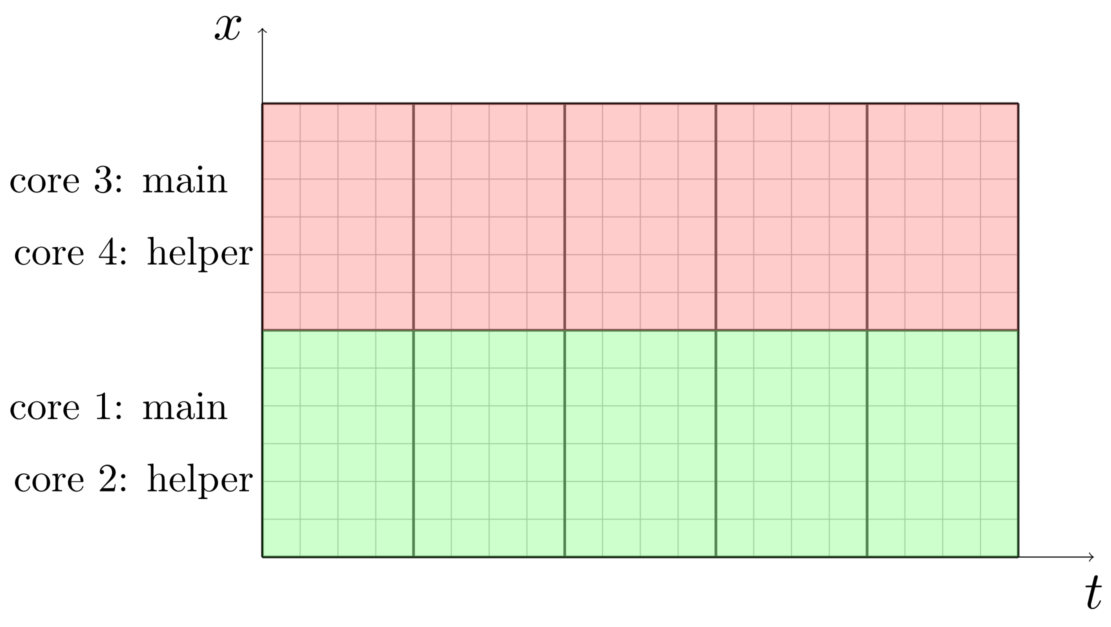


Figure 4: CooperationOnStride parallelization.

Considering four cores available we present an instance of CooperationOnStride parallelization. In Figure 4 a space-time rectangular domain with square elements has 2 subdivision in space, 5 in time and no overlapping. We assign to each stride two cores, one principal and one helper. In a RAS framework as in AloneOnStride parallelization the principal core solves all the subdomains in the stride at each iteration and the other cores help him in parallelization of linear algebra matrix multiplication. In a Pipeline framework at each iteration the principal cores only computes the solution in the subdomains of the sliding window and of course checks the leftmost subdomain residual as in AloneOnStride.

2.3 CooperationSplitTime

This parallelization technique requires that the number of time subdivisions be a multiple of the number of cores and that the total number of cores is a multiple of the number of space subdivisions. In a RAS framework each core is associated to a subset of subdomains which is still a stride but not a full one in time. In Ras Pipelined first we identify the sliding window and then we divide the subdomains inside S within each core. In this way we assign subdomains to different cores not considering the whole domain but only the sliding window.

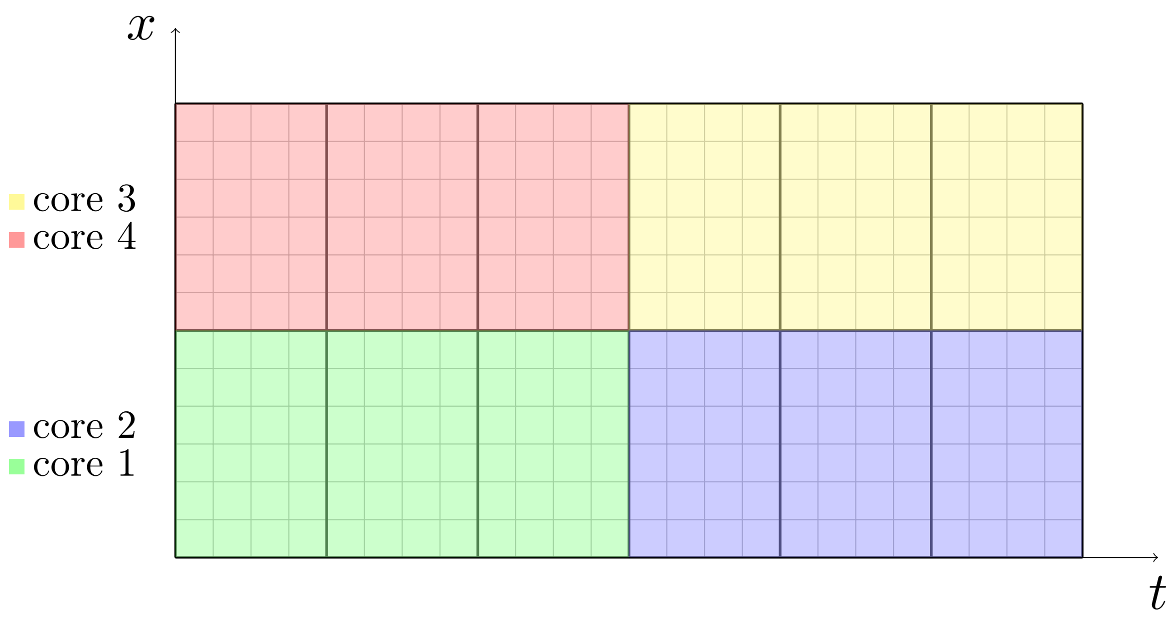


Figure 5: CooperationSplitTime parallelization in RAS.

Considering four cores available we present an instance of CooperationSplitTime parallelization. In Figure 5 a space-time rectangular domain with square elements has 2 subdivision in space, 6 in time and no overlapping. In a RAS framework each core is associated to the cores of the same color group. Differently from the other parallelization techiniques there is also a time subdomain subdivision. In a pipelined setting, Figure 6 the subdomains considered are only inside the sliding window.

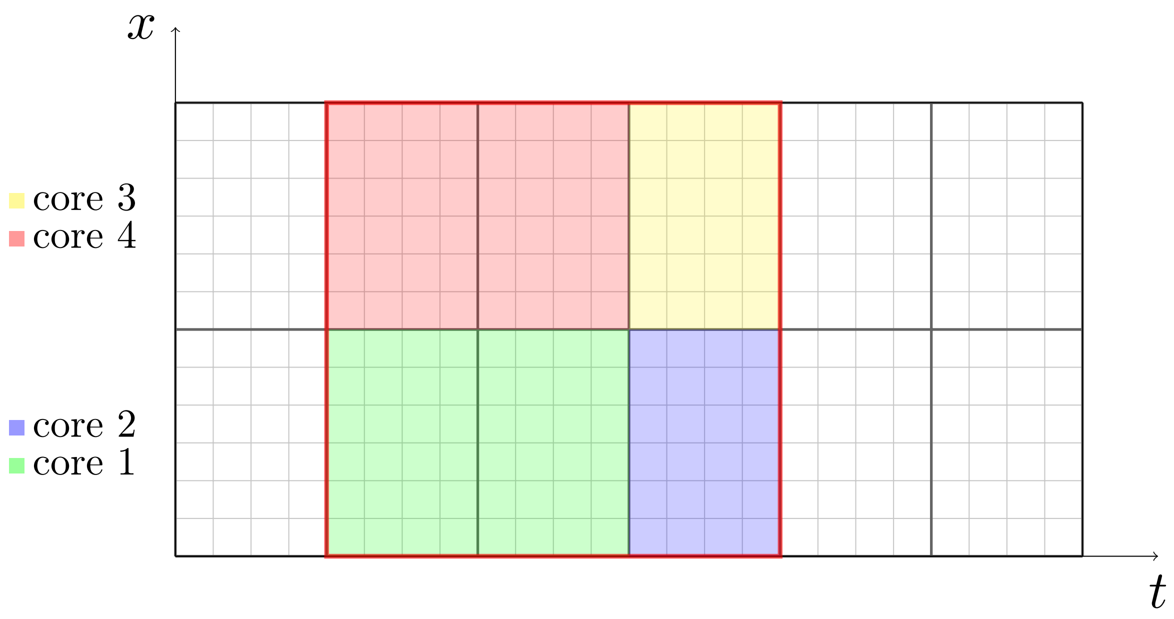


Figure 6: CooperationSplitTime parallelization in RAS Pipelined.

2.4 Comparing the parallelization techniques

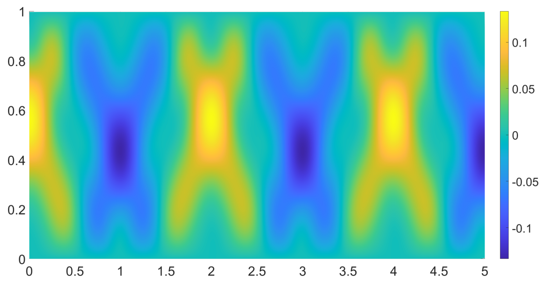
Before comparing the three parallelization strategies we recall some features of the decomposition methods. Decomposition methods, as we said, allow to split the problem into smaller problem to be solved in parallel and faster. However, the subdomains need to transmit numerical information to the neighbours and, if one split too much the domain, the numerical information needs more and more iterations to reach the whole domain. Hence, a very refined decomposition leads to slower convergence. Therefore there is a need to balance the number of subdomains and their size in order to have feasible linear systems and few subdomain edges through which the information needs to travel.
With our code's policies we can explore different possibilities to identify the optimal strategy in terms of decomposition choice and parallelization policy given a number of cores available. Given the same decomposition is it better to have more cores working on the same stride or to further assign less subdomains to each working-alone core? Can the parallelization speed-up beat the numerical drawback when dealing with a very refined decomposition?
Consider four cores available. `AloneOnStride` requires four subdivisions in space to assign at each core a temporal stride (with an arbitrary number of time subdivisions). `CooperationSplitTime` and `CooperationOnStride` requires only two space subdivisions and consequently two temporal stride. In `CooperationOnStride` each entire temporal stride is assigned to two cores, instead in `CooperationSplitTime` there will be a further time division between the two pairs of cores.
We underline that `AloneOnStride` and `CooperationSplitTime` work in the same way, each core is always working alone but the second policy not only succeed in assigning to each core few subdomains, as done in `AloneOnStride`, but succeed also in less decomposing the domain avoiding the numerical drawback.
`CooperationOnStride` has the advantage to requires less space subdivision but keeping two cores working togheter on the same stride could not be optimal with a small problem. Indeed, in this framework, `CooperationSplitTime` performs better. Instead, with a bigger problem it could be useful that more cores works on the same subdomain since the dimension increases and `CooperationOnStride` could be a valid policy. Anyway, with bigger problems, local problem sizes could increase and in this case a further decomposition could be exploit, `AloneOnStride` could be considered as well.
There are many balances to control and all the policies could be useful depending on the problem. From a theoretical point of view, we expect that the more robust one will be `CooperationSplitTime` that assigns few subdomains to each core but at the same time permit to consider a coarse decomposition avoiding numerical drawbacks due to a slow information transfer.

We have implemented those three policies and the baseline sequential version for both domain decomposition methods, RAS and RAS Pipelined.
The pipelined version of Restricted Additive Schwarz already embed in its nature a special treatment of time. The evolution of the moving window in the space time domain permit to consider only few subdomains at each iteration, following the flow of the correct solution started from the initial condition. In terms of computing time, number of subdomains solved and, consequentially, resources required RAS Pipelined is better than RAS.

3 Tests

We have created two tests cases in a rectangular space-time domain with a cartesian grid. The problem is the same but we change the domain and the number of finite elements used.
`Test 1` consider a domain $\Omega = (0,1) \times (0,1)$, with a mesh of 20 and 20 elements in space and time respectively. We decompose the domain in 2 or 4 space subdivisions and 10 time subdivisions.
`Test 2` consider a domain $\Omega = (0,1) \times (0,5)$, with a mesh of 20 and 100 elements in space and time respectively. We decompose the domain in 2 or 4 space subdivisions and 20 time subdivisions.

$$
\begin{cases}
u_{tt} - u_{xx} = 0 & \text{in } \Omega, \\
u(x,0) = x^2(1-x)\sin(\pi x)^2 & x \in [0,1], \\
u_t(x,0) = 0 & x \in [0,1], \\
u(a,t) = u(b,t) = 0 & t \in [0,5].
\end{cases} \tag{5}
$$


(a) displacement u.

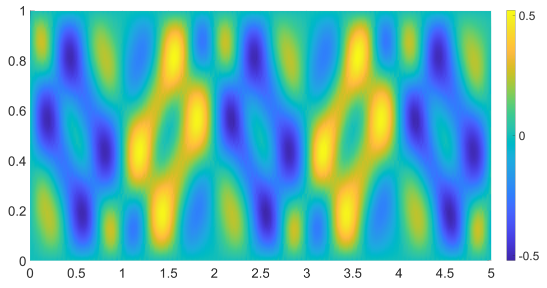
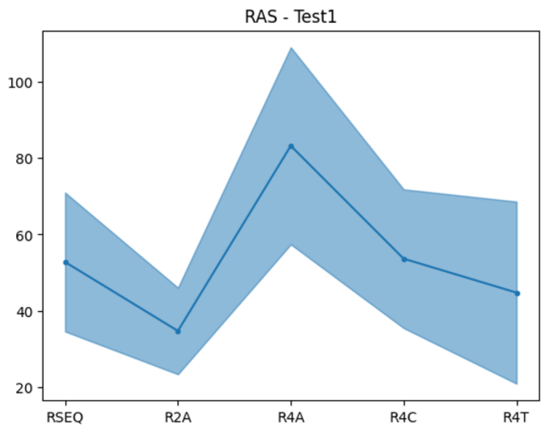

(b) velocity w.

Figure 7: Test 2 solution.

We ran many times the two test cases and we collected the elapsed time to solve the problem. Here we report the results for all the policies with the mean and the standard deviation among the different runs for a single policy.
As notation we refer to RAS method with two cores and `AloneOnStride` policy with `R2A`. In general, we use 2 or 4 to underline the number of cores used (`SEQ` is the sequential version); `A` stands for `AloneOnStride`, `C` for `CooperationOnStride`, `T` for `CooperationSplitTime`. Finally, `R` stands for RAS and `P` for RAS Pipelined method.



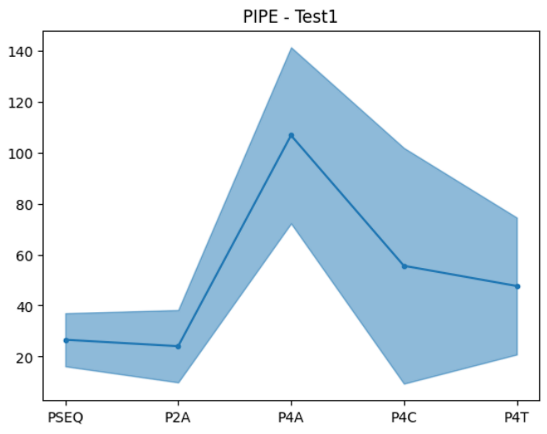
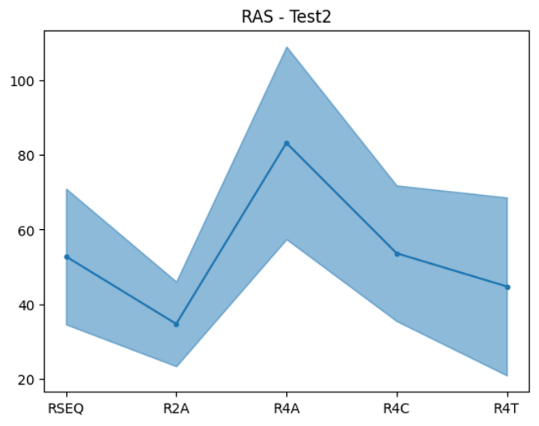


Figure 8: `Test1`, time is in milliseconds.



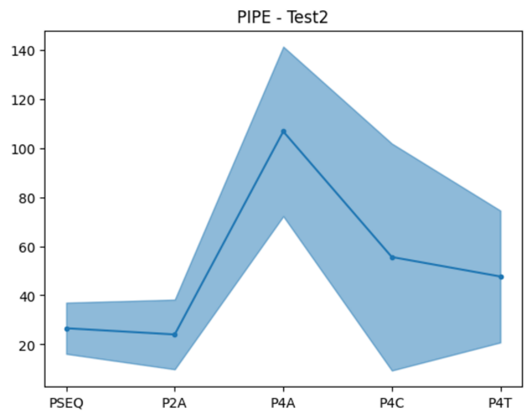


Figure 9: `Test2`, time is in milliseconds.

The parallelized version with two cores beat the sequential one but for this kind of small problems we do not obtain a scalable result, indeed with four cores the performance is worse.
With two cores we are able only to use `AloneOnStride` with two space subdivisions (`R2A`), then when moving to four cores with same policy (`R4A`) there is a huge drop in the performance. This is not due only to parallelization problems but, mainly, to the numerical drawback and the fact that we must use four space subdivisions.
Considering always four cores we can exploit `CooperationOnStride` (`R4C`) and `CooperationSplitTime` (`R4T`) that need two space subdivision as `R2A`. The performance improves, when using four cores it is better to consider a cooperation on the same time-stride in order to have less space subdivisions. Moreover, it is better to have a cooperation in a `CooperationSplitTime` fashion instead of `CooperationOnStride`. This is because for small problems we cannot highlight any advantage when parallelizing linear algebra computations.
As we expect, RAS Pipelined performs better than RAS and this is due to the sliding window that permit to solve less subdomains at each iteration. When comparing these two methods it is more important to focus on the total number of subdomains solved (*solves*) rather than the iterations used because this is an index of how many computational resources are employed. For `Test 2` the *solves* for RAS are 2400, instead for RAS Pipelined they are 540.
Our results show that there is a slow communication between cores and in fact the performance is not scalable between two and four cores. This can be seen in `R2A` and `R4T` that consider the same decomposition and simply assign the subdomains to cores differently. Although in `R4T` there are less subdomains assigned to each core we have a worse performance in elapsed time.

It is reasonable to expect that with bigger problems there will be advantages of a parallelization with four cores. Unfortunately we face some problems when dealing with bigger domains and we cannot provide any test for larger problems. This is due to convergence and stability issues in DG discretization from the NAPDE side of this project.
Nevertheless, the idea remains and with those results we showed that the best way to use four cores is to employ a `CooperationSplitTime` policy.

As a final remark, we reached a huge speed up in the implementation using C++ with respect to the one in Matlab implemented for the NAPDE project. Matlab takes around 2 seconds and `RSEQ` takes about 50ms for `Test 2` solved by RAS in a domain decompose in 2 and 20 space and time subdivisions respectively.

4 Code

The implementation of the solver is in C++ and it is built upon `Eigen` and `MPI` libraries. The `Eigen` package offers to us easy tools for LU factorization for the solution of linear systems. It is also used for manipulate sparse matrices. The `MPI` library is required for the algorithm's parallelization and, finally, we use of `Gnuplot` library for postprocessing plots.

The folders and files structure is the following:

- `doc`: this folder contains the report in a `pdf` file with all the documentation.
- `scr`: this folder contains .cpp files with methods' definitions of the non-template header files in the include folder.
- `include`: this folder contains all the header files of classes and some helper functions used in the program.
- `results`: here we collect the text files output solution of the program, which are the u, w solutions (displacement and velocity, respectively) and the version of these ready for a `Gnuplot` postprocessing.
- `tests`: this is the folder in which problem instances are saved. It contains sub-folders including the matrix A, the right-hand side vector b, the data parameter file and the physical coordinates of the d.o.f. used in the postprocessing plot. Multiple instances of the problem can be saved and selected to be launched.
- main files: `main_seq` runs a sequential solver with no parallelization technique, `main_parallel` runs a parallelization solver. `main_custom_matrix` runs a parallelization version with subdomains specifically assigned to cores.
- Makefile: in **Running instructions and main files** section there is a complete documentation for its use.

We describe the code structure starting from simple classes and moving to advance structures.

First,we note that we make vastly use of the Curiously Recurring Template Pattern (CRTP) technique to define different policies and methods. This allows us to write generic code: if a new solving strategy needs to be implemented we just add a specializiation of the Base class exploiting the inheritance features of C++. We did not make use of static polymorphism althought it is possible to implement it.

```
// The Curiously Recurring Template Pattern (CRTP)
template <class D>
class Base
{
    // methods within Base can use template to access members of Derived
};
class Derived : public Base<Derived>
{
    // ...
};
```

Another general remark for our implementation is that the sparse matrices that we use are stored in a format defined in the `Eigen` library. We rely on a column storage scheme and we defined the type `SpMat`:

```
typedef Eigen::SparseMatrix<double> SpMat;
```

The class SparseMatrix offers high performance and low memory usage and it has implemented a variant of the widely-used Compressed Column Storage scheme. It consists of four compact arrays:

- Values: stores the coefficient values of the non-zeros.
- InnerIndices: stores the row indices of the non-zeros.
- OuterStarts: stores for each column the index of the first non-zero in the previous two arrays.
- InnerNNZs: stores the number of non-zeros of each column (resp. row). The word inner refers to an inner vector that is a column for a column-major matrix, or a row for a row-major matrix. The word outer refers to the other direction.

We suggest to read the Eigen webpage documentation for further information.

4.1 Domain decomposition classes

`include\Domain.hpp` is a self-contained header file of the class `Domain` which includes information of the problem rectangular domain. It contains simple data: `nx` and `nt` are the number of elements in space and time respectively, `X` and `T` the values of the $[0, X] \times [0, T]$ rectangular domain. `nln` represents the number of degrees of freedom for each element, which is crucial for determining the appropriate memory allocation for the matrices. `d` is the spatial dimension, we introduced it in the perspective of generalize the code in 2D but it has no use at the moment. As methods there are only the getters of the members we just described.

`include\decomposition.hpp` is the header file of the class `Decomposition` which goal is to construct the domain decomposition. It has two constructor: one takes `nsub_x` and `nsub_t` the number of subdomains in space and time in which the user wants to decompose the domain; the other constructor takes also the specification of `n` and `m`, number of finite elements in space and time each subdomain the user wants to have, as an example, in Figure 1 `n` is 3 and `m` is 4. If they are not specified, as in the first constructor, they are default-constructed to fit the subdomain choice with minimal or without overlap. Some checks on the input are made to ensure the parameters permit a reasonable domain subdivision, if not an assert is launched.

The methods of the class are defined in `src\decomposition.cpp`. The construction of the decomposition is achieved by the method `void CreateDec(double,double)` which populates `overlap_back`, `overlap_forw` and `start_elem` that are private members of the class `Decomposition`. The members `overlap_back`, `overlap_forw` are matrices that have two rows (one for space and one for time) and a number of columns equal to the total number of subdomains. Each subdomain can have overlap in all the four directions and the sizes of these four possibly overlaps are stored in these matrices. `start_elem` is a vector of length equal to the number of subdomains, its values permit us to link a subdomain with the smallest finite element in time and in space it has.
The total enumeration of elements is progressive in time and starts from zero in space. In Figure 1 the enumeration starts in the origin and goes from the element smaller in time to the element further in time and then moves in space with the last element having maximum time and space coordinates. The construction of these enumeration is a bit convoluted and technical, but the data is computed only once at the preprocessing stage. The method `tupla<5 unsigned int> get_info_sub_k(unsigned int k)`, given an index of the subdomain, returns its information about the four overlaps and its identification number. The method `vector<int> basic_info_decomposition()` returns a vector containing information about the domain decomposition which are the member of Decomposition class.

In `include\sub_assignment.hpp` there is a self-contained header file of the virtual class `SubAssignment`, which is a single parameter template class. The template parameter can be either `AloneOnStride`, `CooperationOnStride` or `CooperationSplitTime` and those are themselves classes derived from the virtual class `SubAssignment` using templates. Here we use the CRTP.

```
template<class LA>
class SubAssignment
{
public:
  //constructor {...}
  //getters {...}
  virtual void createSubDivision() = 0;
  unsigned int idxSub_to_LocalNumbering(unsigned int k,int current_rank) const {...};
protected:
  //members
};

class AloneOnStride : public SubAssignment<AloneOnStride>
{
  public:
  //constructor {...}
  void createSubDivision() override{...};
};

class CooperationOnStride : public SubAssignment<CooperationOnStride>
{
  public:
    //constructor {...}
    void createSubDivision() override{...};
};

class CooperationSplitTime : public SubAssignment<CooperationSplitTime>
{   public:
    //constructor {...}
    void createSubDivision() override{...};
};
```

The goal of the class is to construct the `sub_division_vec` a vector whose elements are vectors of subdomain indices. It associates to each processor the subdomains it has to solve. The method responsible for this is `void createSubDivision()` which is virtual in `SubAssignment` and it is overridden in the derived classes. The method `unsigned int idxSub_to_LocalNumbering(unsigned int,int)`, implemented in the virtual class, takes as input the rank of the processor and the subdomain index and it returns the index of `sub_division_vec` that correspond to that subdomain.

In `include\local_matrices.hpp` there is a self-contained header file of the class `LocalMatrices`, a single parameter template class. The template parameter refers to the `SubAssignment` policy employed. This class is responsible to generate and store the local matrices A_j and $\tilde{R}_j, R_j$. Each processor construct this class and each of them saves different matrices according to the subdomains assigned to it. Each core creates a vector for storing those matrices with the number of subdomains assigned as vector size. The class has member we already described: the classes `Domain`, `Decomposition` and `SubAssignment` which needs the template parameter of `LocalMatrices`. The other members are `current_rank`, that is the rank of the processor and `R` , `R_tilde` and `localA` which are the vectors of Sparse matrices. The methods `std::pair<SpMat, SpMat> createRK(int)`, `void createRMatrices()`, `void createAlocal(const SpMat& A)` construct and save the local matrices. Finally, the getters are `std::pair<SpMat, SpMat> getRk(int)` and `SpMat getAk(int)` that return the local matrices given the subdomain index, for this is used the method `uint idxSub_to_LocalNumbering(uint,int)` of the class `SubAssignment`.

`include\solver_traits.hpp` is a self-contained header file for the class `SolverTraits` which includes traits about the solvers and backup variables for RAS Pipelined. Some members of this class are setted by the user in the test data files. The user can choose the `max_it` maximum number of total iterations to perform before stopping the computations; `tol` and `tol_pipe_sx` are the tolerances of RAS and RAS Pipelined residuals and `it_waited` is the number of iteration to wait before enlarginging the sliding window in RAS Pipelined. The members `subt_sx`, `zone` and `it_waited` are needed only for RAS Pipelined. At each iteration of the method they refer to the smallest subdomain in time and space of the sliding window, the size of the sliding window and the iteration occured after its last update. The class has the getters and setters for all members.

`include\solver_results.hpp` is the header file of the class `SolverResults` that stores the final solutions and performances of the method plus the information of solver traits and domain decomposition. The final solution is `Eigen::VectorXd uw` and the performances are the variables `solves` and `time` which are the total number of times a linear system associated to a subdomain has been solved and the time in milliseconds of the total computation. The methods `Eigen::VectorXd getUW() Eigen::VectorXd getU()`, `Eigen::VectorXd getW()` are the getters of the whole solution vector `uw` or only for the displacement u or the velocity w. An additional method defined in `src\solver_results.cpp` is `void formatGNU(int,string, unsigned int, unsigned int)` that generates `u_gnuplot.txt` and `w_gnuplot.txt`. Those are text files of the solutions in `results` folder in a suitable format to be plotted with `Gnuplot`.

`include\exchange_txt.hpp` is a self contained header file which contains some helper functions. There are the method `SpMat readMat_fromtxt(const std::string&, uint ,uint)` and the method `void saveVec_totxt(const std::string, const Eigen::VectorXd)`. The first method is used to read the text file of the matrix A or b and convert them in a `SpMat` type object. The second one does the opposite, it saves the solution in a txt file.

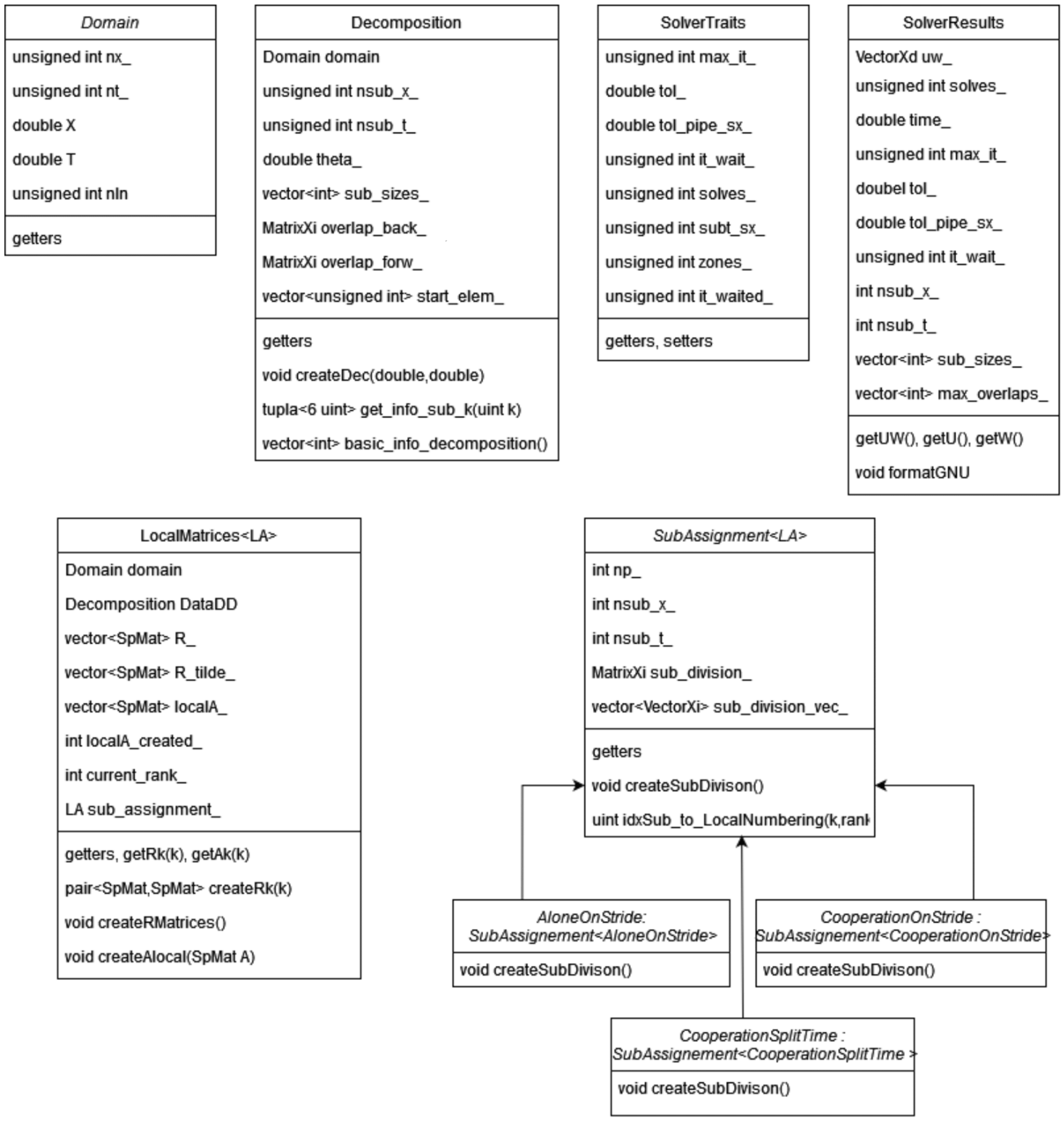


Figure 10: Domain decomposition classes structure. It is a scheme, the declarations could be simplified due to graphical reasons.

4.2 Solvers

`include\domaindec_solver_factory.hpp` is a self-contained header file of the class `DomainDecSolverFactory`. It is a object factory class which creates an instance of a solver specified through the template parameters. The constructor takes `Domain`, `Decomposition`, `LocalMatrices` and `SolverTraits` type objects and pass them to the solver that will be generated. The template method `std::unique_ptr<DomainDecSolverBase<P,LA>> createSolver(std::string const,Args&&)` generates the unique pointer to a derived class from `DomainDecSolverBase` type class using the perfect forwarding technique common in factory classes. The valid template parameters are RAS or RasPipelined for P and for LA the parallelization policies that we already described. Moreover there is an operator definition `SolverResults operator()(const std::string&, const SpMat& ,const SpMat&)`. At high level this is the one that is called in the main file and by calling it a solver following the template parameter specification will be generated and it will solve the problem returning a `SolverResults` object containing the solution.

`include\domaindec_solver_base.hpp` is a self-contained header file of the virtual class `DomainDecSolverBase`. It is the Base class for the solvers and it has two template parameters: `P` that represent the solver and `LA` that represent the subdomains assignment policy. The members are the same as in `DomainDecSolverFactory` with in addition `std::vector<Eigen::SparseLU<SpMat>> localLU`. It stores the LU factorization of the local matrices A_k, since the linear systems need to be solved multiple times. The construction of the LU factorization is not performed by a specific method but rather explicitly in the constructor. The assumption is that in all the solvers the LU factorization is the standard way to solve the local linear system. The method `const Eigen::SparseLU<SpMat>& get_LU_k(uint)` returns the LU matrix factorization associated to a subdomain given its index. The virtual method is `SolverResults solve(const SpMat&, const SpMat&)` which is overridden by the derived classes.

`include\ras.hpp` and `include\ras_pipelined.hpp` are self-contained header files of the class `RAS` and `RasPipelined` respectively. They are derived class of `DomainDecSolverBase`. Each of them have the constructor and a method `SolverResults solve(const SpMat&, const SpMat&)` which is the implementation of the virtual method in `DomainDecSolverBase`. Here we defined all solvers with all the possible combinations with subdomains assignment policies. We underline that each solver class `P` is derived from `Ras` or `RasPipelined` classes with `P` as first template parameter specification.

```cpp
//CRTP use for solvers in ras.hpp

class Parallel_AloneOnStride : public Ras<Parallel_AloneOnStride,AloneOnStride>
{
public:
    //constructor {...}
    Eigen::VectorXd precondAction(const SpMat& x){...};
    SolverResults solve(const SpMat& A, const SpMat& b){};
};

class Sequential : public Ras<Sequential,AloneOnStride>
{
public:
    //constructor {...}
    Eigen::VectorXd precondAction(const SpMat& x){...};
    SolverResults solve(const SpMat& A, const SpMat& b){...};
};

class Parallel_CooperationOnStride :
public Ras<Parallel_CooperationOnStride,CooperationOnStride>
{
    private:
    //member
    public:
    // constructor {...}
    Eigen::VectorXd precondAction(const SpMat& x{...};
    SolverResults solve(const SpMat& A, const SpMat& b){...};
};

class Parallel_CooperationSplitTime :
public Ras<Parallel_CooperationSplitTime,CooperationSplitTime>
{
    public:
    //constructor {...}
    Eigen::VectorXd precondAction(const SpMat& x){...};
    SolverResults solve(const SpMat& A, const SpMat& b){...};
};
```

Each of the solver has the **`Eigen::VectorXd precondAction(const SpMat&)`** method and **`SolverResults solve(const SpMat&, const SpMat&)`** method. They are used to perform the linear system computation and they are specific for each parallelization technique.

```

//CRTP use for solvers in ras_pipelined.hpp


template<class P,class LA>
class RasPipelined : public DomainDecSolverBase<P,LA> {
public:
    //constructor {...}
    SolverResults solve(const SpMat& A, const SpMat& b) override{...};
protected:
    //member
};

class PipeParallel_AloneOnStride :
public RasPipelined<PipeParallel_AloneOnStride, AloneOnStride>
{
public:
    // constructor {...}
    SolverResults solve(const SpMat& A, const SpMat& b) override {...};
    Eigen::VectorXd precondAction(const SpMat& x){...};
    unsigned int check_sx(const Eigen::VectorXd& v){...};
};


class PipeParallel_CooperationOnStride :
public RasPipelined< PipeParallel_CooperationOnStride, CooperationOnStride>
{
private:
    // members {...}
public:
    //constructor {...}
    SolverResults solve(const SpMat& A, const SpMat& b) override{...};
    Eigen::VectorXd precondAction(const SpMat& x)
    {...};
    unsigned int check_sx(const Eigen::VectorXd& v){...};

};


class PipeSequential : public RasPipelined<PipeSequential,AloneOnStride>
{
public:
    //constructor {...}
    SolverResults solve(const SpMat& A, const SpMat& b) override {...};
    Eigen::VectorXd precondAction(const SpMat& x) {...};
    unsigned int check_sx(const Eigen::VectorXd& v) {...};
};


class PipeParallel_CooperationSplitTime :
public RasPipelined< PipeParallel_CooperationSplitTime, CooperationOnStride>
{
public:
    //constructor {...}
    SolverResults solve(const SpMat& A, const SpMat& b) override {...};
    Eigen::VectorXd precondAction(const SpMat& x){...};
    unsigned int check_sx(const Eigen::VectorXd& v){...};
};
```

Each of the solver has the `Eigen::VectorXd precondAction(const SpMat&) SolverResults solve(const SpMat&, const SpMat&)` method and the `unsigned int check_sx(const Eigen::VectorXd& )` method. They differ from the RAS solver since the algorithm is slightly more complex. The main difference is that at each iteration the information about the sliding window and the algorithm are stored into the `traits` member. The initialization of these parameters are in the `RasPipeline` class.

In the `RAS` and `RasPipelined` class the `solve` method generate an object `P` that represents a solver among the ones defined in `ras.hpp` and `ras_pipelind.hpp`. Successively, the `solve` method of `P` is called. We opted to add the intermedieate class of `RAS` and `RasPipelined` between `DomainDecSolverBase` and the real solvers. This permits us to be cleaner and tidier in the division of the two method since they differ in the parameters and in the methods. Otherwise, every solver would have been derived directly from `DomainDecSolverBase` with all the methods declared as virtual. This is unpleasant in the view of an additional solver to be coded which might require to modify the `DomainDecSolverBase` by adding its method as virtual. Instead it would be better, in our opinion, to just have the `solve` method as virtual and create an intermediate derived class to group solver of the same type as in our case and specialize them further if needed.

```
// RAS solve method
 SolverResults solve(const SpMat& A, const SpMat& b) override
  {
    // It creates an object P that represent a Policy and then call the solve method
    // implemented in that Policy P
    P func_wrapper(this->domain,this->DataDD,this->local_mat,this->traits_);
    return func_wrapper.solve(A,b);
  };
```

```
// RasPipeline solve method
SolverResults solve(const SpMat& A, const SpMat& b) override
{
    // initialize parameters for zone evolution
    this->traits_.setZone(2);
    this->traits_.setSubtSx(1);
    this->traits_.setItWaited(0);
    this->traits_.setSolves(0);

    // It creates an object P that represent a Policy and then call the solve method
    //implemented in that Policy
    P func_wrapper(this->domain,this->DataDD,this->local_mat,this->traits_);
    SolverResults res_obj = func_wrapper.solve(A,b);

    return res_obj;
};
```

Overall there are eight solvers to choose from:

- Ras
  `-Sequential :  public Ras<Sequential, AloneOnStride>`
  `-Parallel_AloneOnStride :  public Ras<Parallel_AloneOnStride,AloneOnStride>`
  `-Parallel_CooperationOnStride :  public Ras<Parallel_CooperationOnStride, CooperationOnStride>`
  `-Parallel_CooperationSplitTime :  public Ras<Parallel_CooperationSplitTime, CooperationSplitTime>`

- Ras Pipelined
  `-PipeSequential :  public RasPipelined< PipeSequential,AloneOnStride>`
  `-PipeParallel_AloneOnStride :  public RasPipelined<PipeParallel_AloneOnStride, AloneOnStride`
  `-PipeParallel_CooperationOnStride :  public RasPipelined<PipeParallel_CooperationOnStride, CooperationOnStride>`
  `-PipeParallel_CooperationSplitTime :  public RasPipelined< PipeParallel_CooperationSplitTime, CooperationOnStride>`

In the solver `PipeParallel_CooperationSplitTime` the subdomains assignment policy is `CooperationOnStride` instead of `CooperationSplitTime` because the cores associated to a time stride need to access to all subdomain's information in that time stride. This is because the sliding window will capture all time zones and only inside S there will be the subdomains assignment between cores. For example, focusing in a single time stride, if we apply `CooperationSplitTime` we will have on a time stride two cores working, the first one on the early in time subdomains and the second one on the later ones. However, the sliding window will not always cover subdomains coming from both the two cores and in this case there will be no parallelization. To solve this issue, we assign to both cores all the subdomains inside that time stride (`CooperationOnStride`), then as the sliding window is updated we equally assign subdomains to different cores (Figure 6). We do not split subdomains among cores based on the whole domain but we base the assignment on the sliding window. We apply an ad-hoc subdomains assignment at each iteration due to the dynamic evolution of the sliding window.
`CooperationSplitTime` is only used in the `RAS` framework where there is no sliding window.

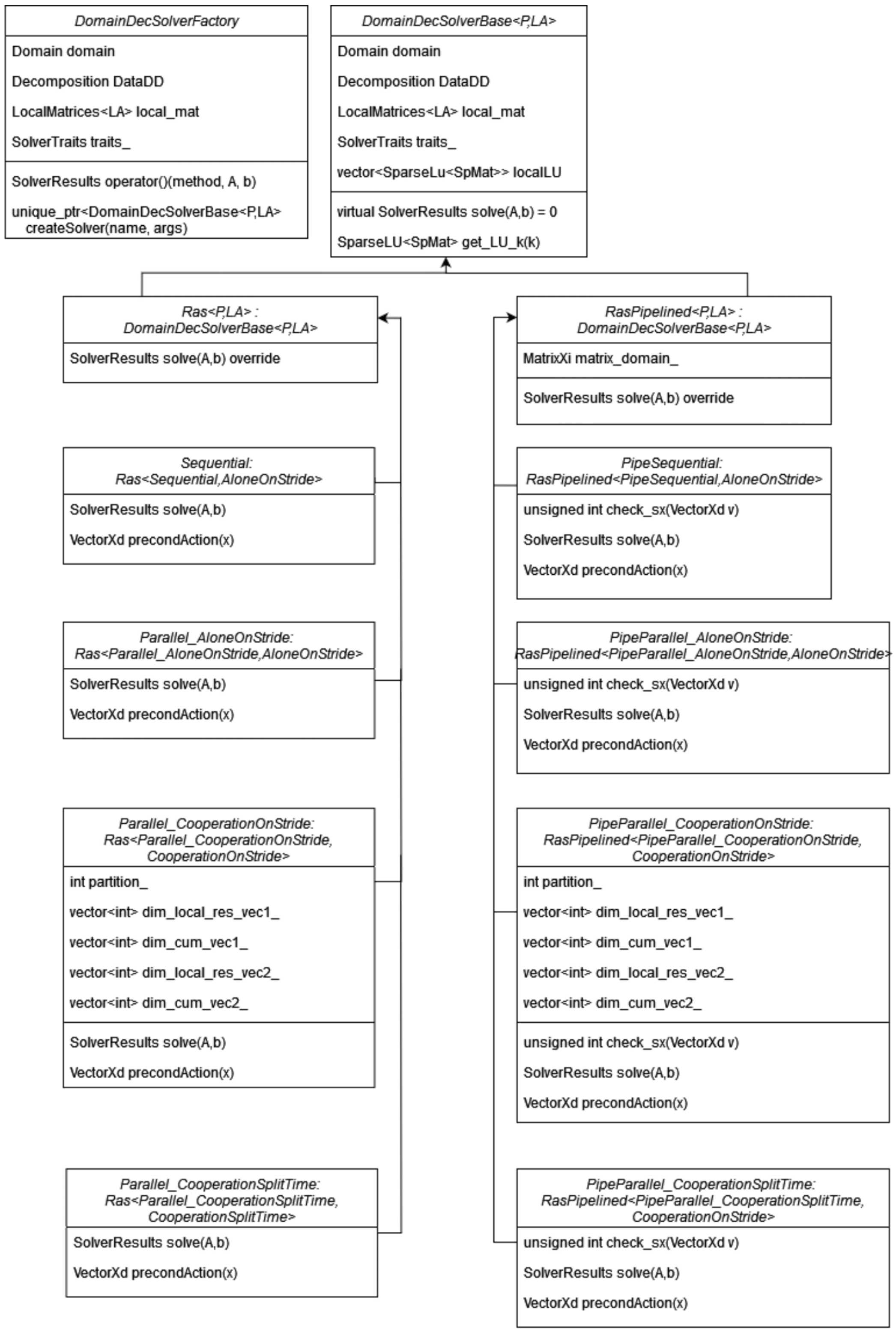

DomainDecSolverFactory
Domain domain
Decomposition DataDD
LocalMatrices<LA> local_mat
SolverTraits traits_
SolverResults operator()(method, A, b)
unique_ptr<DomainDecSolverBase<P,LA> createSolver(name, args)
DomainDecSolverBase<P,LA>
Domain domain
Decomposition DataDD
LocalMatrices<LA> local_mat
SolverTraits traits_
vector<SparseLu<SpMat>> localLU
virtual SolverResults solve(A,b) = 0
SparseLU<SpMat> get_LU_k(k)
Ras<P,LA> : DomainDecSolverBase<P,LA>
SolverResults solve(A,b) override
RasPipelined<P,LA> : DomainDecSolverBase<P,LA>
MatrixXi matrix_domain_
SolverResults solve(A,b) override
Sequential: Ras<Sequential,AloneOnStride>
SolverResults solve(A,b)
VectorXd precondAction(x)
PipeSequential: RasPipelined<PipeSequential,AloneOnStride>
unsigned int check_sx(VectorXd v)
SolverResults solve(A,b)
VectorXd precondAction(x)
Parallel_AloneOnStride: Ras<Parallel_AloneOnStride,AloneOnStride>
SolverResults solve(A,b)
VectorXd precondAction(x)
PipeParallel_AloneOnStride: RasPipelined<PipeParallel_AloneOnStride,AloneOnStride>
unsigned int check_sx(VectorXd v)
SolverResults solve(A,b)
VectorXd precondAction(x)
Parallel_CooperationOnStride: Ras<Parallel_CooperationOnStride, CooperationOnStride>
int partition_
vector<int> dim_local_res_vec1_
vector<int> dim_cum_vec1_
vector<int> dim_local_res_vec2_
vector<int> dim_cum_vec2_
SolverResults solve(A,b)
VectorXd precondAction(x)
PipeParallel_CooperationOnStride: RasPipelined<PipeParallel_CooperationOnStride, CooperationOnStride>
int partition_
vector<int> dim_local_res_vec1_
vector<int> dim_cum_vec1_
vector<int> dim_local_res_vec2_
vector<int> dim_cum_vec2_
unsigned int check_sx(VectorXd v)
SolverResults solve(A,b)
VectorXd precondAction(x)
Parallel_CooperationSplitTime: Ras<Parallel_CooperationSplitTime, CooperationSplitTime>
SolverResults solve(A,b)
VectorXd precondAction(x)
PipeParallel_CooperationSplitTime: RasPipelined<PipeParallel_CooperationSplitTime, CooperationOnStride>
unsigned int check_sx(VectorXd v)
SolverResults solve(A,b)
VectorXd precondAction(x)

Figure 11: Solver classes structure.

4.3 MPI code

Once we have decided how to assign subdomains to different cores following the different policies described above, we need to implement a strategy to make the cores working together in parallel with the least possible communication.
We recall that all the cores have at their disposal the global matrix A, then a generic core knows the local matrices A_k and the restriction matrices R_k only of the subdomains assigned to it by `SubAssignment` class.

Mainly we can identify two areas of our code that could be parallelized:

- local solution. We divide the summation inside the definition of P in RAS and RAS Pipelined methods, therefore each core compute the local solution only for a subset of the total subdomains. This can be seen in `Eigen::VectorXd precondAction(const SpMat&)` ;
- matrix multiplications. We implement parallel linear algebra for computing the residual in `SolverResults solve(const SpMat&, const SpMat&)` method and for compute the local solution for a subdomain in `Eigen::VectorXd precondAction(const SpMat&)`. To be more clear, in the latter, we need to compute firstly the restriction over the subdomain k of the residual x, $R_k x$; then, after we obtain the local solution u_k, we compute the prolongation in the whole domain: $\tilde{R_j}^T u_k$. The restriction and prolongation multiplication are computed in parallel.

Secondly, also the check for convergence of the left side of the sliding window S in Ras Pipelined method must be implemented in parallel since a single cores can verify only the convergence inside the subdomains to which it is assigned.

We are going to report only the details of the implementation for RAS method for this two tasks. For RAS Pipelined these procedures are the same but there are embedded more steps due to the definition and the evolution of the sliding window S.

4.3.1 Local solution

In `AloneOnStride` and `CooperationSplitTime` only one core work on a particular subset of subdomains. In `Eigen::VectorXd precondAction(const SpMat&)` we can see how the parallelization is implemented:

```
Eigen::VectorXd
    precondAction(const SpMat& x)
    {
      // solve the local problems in the subdomains
      Eigen::VectorXd z=Eigen::VectorXd::Zero(domain.nln()*domain.nt()*domain.nx()*2);
      Eigen::VectorXd uk(domain.nln()*DataDD.sub_sizes()[0]*DataDD.sub_sizes()[1]*2);
      int rank{0};

      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      // Get the subdomains assigned to the current rank
      auto sub_division_vec =
            local_mat.sub_assignment().sub_division_vec()[local_mat.rank()];

      for(unsigned int k : sub_division_vec){
           auto temp = local_mat.getRk(k); // temp contains R_k and Rtilde_k
           uk = this->get_LU_k(k).solve(temp.first*x);
           z=z+(temp.second.transpose())*uk;
           }

      // Each rank compute the solution over the subdomains assigned and then collect
      // and sum all the results with Allreduce
      MPI_Allreduce(MPI_IN_PLACE, z.data(), domain.nln()*domain.nt()*domain.nx()*2,
                MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
      return z;
    }
```

It is simply a matter to define for which subdomains we have to loop and this task is completed by `SubAssignment` class.
We underline that we must compute an `MPI_Allreduce` since every core must return the global `z` to the `solve` method to correctly compute the new residual.
In `CooperationOnStride` more cores works on the same stride but this procedure is the same, we divide the summation in the same way. It changes what we do inside the for loop since we exploit parallel linear algebra as we are going to show in the next section.

4.3.2 Parallel Linear Algebra

Linear algebra for matrix multiplication is implemented in `CooperationOnStride`. We report here the implementation for RAS, both the `solve` and the `precondAction` method.

```
    SolverResults solve(const SpMat& A, const SpMat& b)
    {
      int rank{0},size{0};
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      //[...] some parameters initilization not reported here

      int dim_res{
        static_cast<int>(this->domain.nln()*this->domain.nt()*this->domain.nx()*2)};

      std::vector<int> dim_local_res_vec(size,0);
      std::vector<int> dim_cum_vec(size,0);
      // compute the local dimension assigned to each process
      for(int i = 0; i<size; ++i){
        dim_local_res_vec[i] = dim_res/size + (i < (dim_res % size));
        if(i!=0)
          dim_cum_vec[i] = dim_local_res_vec[i-1] + dim_cum_vec[i-1];
      }


      Eigen::VectorXd uw0=Eigen::VectorXd::Zero(dim_res);
      Eigen::VectorXd uw1(dim_res);

      Eigen::VectorXd prod(dim_res);
      Eigen::VectorXd local_prod(dim_local_res_vec[rank]);

      Eigen::VectorXd z = precondAction(b);
      while(res>tol and niter<max_it){
          uw1=uw0+z;

          // parallelize A*uw1
          //compute local_prod
          local_prod =  A.middleRows(dim_cum_vec[rank], dim_local_res_vec[rank])* uw1;

          //gatherv di prod
          MPI_Allgatherv (local_prod.data(), dim_local_res_vec[rank], MPI_DOUBLE,
                prod.data(), dim_local_res_vec.data(), dim_cum_vec.data(),
                MPI_DOUBLE, MPI_COMM_WORLD);

          z = precondAction(b-prod);
          res = (z/Pb2).norm();
          niter++;
          uw0 = uw1;
      }
```

```
  //[...] some postprocessing features not reported here

  SolverResults res_obj(uw1,solves,time, this->traits_, DataDD);

  return res_obj;
};
```

In **solve** we parallelize the residual computation. Consider, for example, four cores and two subdivision in space to obtain two temporal stride where two cores works on each of them. First we compute the local dimension of the result and the cumulative vector that is used by `MPI_Allgatherv` to understand how to gather the results for all cores.
To select only few rows of A we exploit `middleRows` method of `Eigen` for sparse matrices.

In `precondAction` we parallelize for each subdomain the restriction and prolungation operation to compute the local solution.
Unlike what happens before we previously compute the local dimensions of the results for the two operations, this is because this function is called at each iteration of the solver and those dimensions do not change. The constructor of the solver of `CooperationOnStride` does that.

```
Eigen::VectorXd
precondAction(const SpMat& x)
{
  // solve the local problems in the subdomains
  int rank{0},size{0};
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int dim_res{
    static_cast<int>(this->domain.nln()*this->domain.nt()*this->domain.nx()*2)};
  int dim_k{
    static_cast<int>(domain.nln()*DataDD.sub_sizes()[0]*DataDD.sub_sizes()[1]*2)};

  Eigen::VectorXd z=Eigen::VectorXd::Zero(dim_res);
  Eigen::VectorXd uk(dim_k);

  // prod1: restriction over subk:            Rk * x
  // prod2: prolungation in the whole domain: Rtilde_k' * uk

  Eigen::VectorXd prod1(dim_k);
  Eigen::VectorXd local_prod1(dim_local_res_vec1_[rank]);
  Eigen::VectorXd prod2(dim_res);
  Eigen::VectorXd local_prod2(dim_local_res_vec2_[rank]);

  // Get the subdomains assigned to the current rank
  auto sub_division_vec =
    local_mat.sub_assignment().sub_division_vec()[local_mat.rank()];
  for(unsigned int k : sub_division_vec){
      auto temp = local_mat.getRk(k);

      // prod1: temp.first*x
      local_prod1 =
        temp.first.middleRows(dim_cum_vec1_[rank], dim_local_res_vec1_[rank])* x;

      // gathering the result in the main core of the time stride
      MPI_Gatherv (local_prod1.data(), dim_local_res_vec1_[rank], MPI_DOUBLE,
        prod1.data(), dim_local_res_vec1_.data(),
        dim_cum_vec1_.data() , MPI_DOUBLE, rank % partition_ , MPI_COMM_WORLD);

      // send the result also to the other processes of the time stride
      if(rank < partition_)
        for(int dest=rank+partition_; dest<size; dest+=partition_)
          MPI_Send (prod1.data(), dim_k, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
      else
```

```
             MPI_Recv (prod1.data(), dim_k, MPI_DOUBLE, rank%partition_, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

          uk = this->get_LU_k(k).solve(prod1);

          // ---------------------
          //prod2:  (temp.second.transpose())*uk
          local_prod2 = temp.second.transpose().middleRows(
            dim_cum_vec2_[rank], dim_local_res_vec2_[rank])* uk;

          // gathering the result in the main core of the time stride
          MPI_Gatherv (local_prod2.data(), dim_local_res_vec2_[rank], MPI_DOUBLE,
                prod2.data(), dim_local_res_vec2_.data(),
                dim_cum_vec2_.data() , MPI_DOUBLE, rank % partition_ , MPI_COMM_WORLD);

          // send the result also to the other processes of the time stride
          if(rank < partition_)
            for(int dest=rank+partition_; dest<size; dest+=partition_)
              MPI_Send (prod2.data(), dim_res, MPI_DOUBLE, dest, 1, MPI_COMM_WORLD);
          else
             MPI_Recv (prod2.data(), dim_res, MPI_DOUBLE, rank%partition_, 1,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

          z = z + prod2;

          }

      // The main processes of each time stride collect and sum the
      //results with Allreduce

      MPI_Comm master_comm; // communicator with one member for each temporal stride
      MPI_Comm workers_comm; // communicator with all helpers cores

      int color,color2;
      color = (rank < DataDD.nsub_x()) ? 0 : 1;
      color2 = (rank >= DataDD.nsub_x() || rank == 0) ? 0 : 1;

      MPI_Comm_split(MPI_COMM_WORLD, color, rank, &master_comm);
      MPI_Comm_split(MPI_COMM_WORLD, color2, rank, &workers_comm);
      // master ranks sums the result between them
      if(color==0)
        MPI_Allreduce(MPI_IN_PLACE, z.data(), domain.nln()*domain.nt()*domain.nx()*2,
            MPI_DOUBLE, MPI_SUM, master_comm);
      // then rank0 communicate the final result to all others ranks
      if (color2 == 0)
        MPI_Bcast(z.data(), domain.nln()*domain.nt()*domain.nx()*2, MPI_DOUBLE,
            0, workers_comm);

      MPI_Comm_free(&master_comm);
      MPI_Comm_free(&workers_comm);

      return z;
    }
```

The main steps to parallelize the computations are the same as the ones in `solve` but there are some differences to underline. Inside the loop we cannot perform a `MPI_Allgatherv` since the helpers cores do not have new and useful information. A `MPI_Gatherv` is used to gather the results of only the main cores of each stride, then a `MPI_Send` and `MPI_Recv` procedure is used to let also the helpers code know the current solution in order to go on with the iterations.
Finally, once the loop over the subdomains of each stride is ended, it is needed to exchange the results between all strides. Once again, we cannot perform a reduction over all cores since the helpers cores have the same result of their master. To face this problem we define new communicators with `MPI_Comm_split` to compute the reduction operations with `MPI_Allreduce` only between the main

cores of each stride. Then a `MPI_Bcast` is required for sharing the result also to the helpers core in order to continue with the iterations with the correctly computed residual.

4.4 Custom matrix for subdomains assignment

We have seen that through the `SubAssignment` class it is possible to define a generic policy that define how subdomains are assigned to different cores. To underline the generality of our code, we have implemented the possibility to pass as input a generic and potentially random matrix that describe the subdomains assignment step. Imagine to represent the 2D rectangular domain and its decomposition with a matrix C. Each element (i, j) of the matrix represents a subdomain and C_{ij} is the core to which a subdomain is assigned, $C_{ij} = 0, 1, ...,$ n.cores available - 1.
With this custom matrix the user can arbitrally decide a parallelization tecnique simply by assigning subdomains to different cores, it could be a fully space parallelization or he can decide to assign the 80% of the subdomains to a powerful core and the remaining 20% to a smaller and busier core.
For this policy we have implemented only the parallelization of the summation in `precondAction`, the possibility to parallelize linear algebra is not implemented. This custom matrix is only a feature to underline the generality of this code from the subdomains assignment point of view.

5 Running instructions and main files

In this section we provide complete examples of building and running the program; some info are already reported in README.md.
First of all the program can be obtained at GitHub. It's suggested to clone the repository, for example if using https executing in a terminal window:
`git clone https://github.com/francescosongia/waveXT_ParallelDomDec`

The program needs the text files `A.txt`, `b.txt` to describe the DG discretization of the problem. Moreover, some parameters are needed to describe the solution policies: they are provided to the program using the `GetPot` library through a `data` file. All of them are organized and stored inside a test case folder such as `tests\test1`.
To briefly describe the main files we report the main steps: at first, the parameter are read through `GetPot`, some assertion with `assert.h` are made to check if the user has passed compatible parameters. Once those checks are passed, all the cores read the problem matrices from `.txt` files and the `Decomposition` object is initialized. Through the `GetPot` parameters also the specification of template parameters are provided and finally a `LocalMatrices` object and a solver object could be created. Finally a `SolverResult` object is returned and the solution is plotted through `Gnuplot`.
We have provided three different main files: `main_parallel.cpp`, `main_seq.cpp` and `main_custom_matrix.cpp`. Those are for the parallel, sequential and with custom matrix version of the program.

Here we report a description of parameters that must be passed with `GetPot` through the `data` file.

```
[parameters]
  [./problem]
    nx     = 20              // finite space elements number
    nt     = 20              // finite time elements number
    x      = 1               // space domain border
    t      = 1               // time domain border
    nln    = 6               // d.o.f. for each finite element
  [../]

  [./decomposition]
    nsubx = 2                // number of space subdomains
    nsubt = 10               // number of time subdomains
    size_subx = 0            // sizes (in finite elements) of space (time) subdomains,
    size_subt = 0            //       if 0 they are automatically computed
  [../]

  [./traits]
    method          = PIPE                          // dom. dec. method (RAS, PIPE)
    ParPolicy       = AloneOnStride                 // policy used for subs assignment
                                                    // (CooperationSplitTime, AloneOnStride,
                                                    //       CooperationOnStride)
    max_iter        = 100                           // iteration for the iterative solver
    tol             = 1e-10                         // tollerance for the iterative solver
    tol_pipe_sx     = 1e-10                         // tol for sliding window evolution
    it_wait_pipe    = 3                             // iterations to wait before sliding
                                                    //  window evolution in the right side

  [../]

[../]

[file_matrices]
  test = test1                                       // test folder
```

In the custom matrix version for subdomains assignment there is also an option to define it the custom matrix is provided as input with a `custom_matrix.txt` file or if the user want to randomly generates it.

```
    [custom_matrix]
            random = 1   // 1 for a random matrix, 0 for a txt file with an
                            already defined matrix
```

To run the program we have created a `Makefile` that contains also a brief description of the options. It can be called with `make help` with the following output:

```
Options:
- make all to build sequential and parallel
- make main, mainseq, maincustom to build parallel, sequential and custom matrix version, respectively
- make distclean to remove objects and executables
- make run <num_processes> <testname> to built and run in parallel with data and parameters specified in tests/<testname>/data
- make runseq <testname> to built and run in sequential with data and parameters specified in tests/<testname>/data
- make runcustom <num_processes> to built and run in parallel with data and parameters specified in tests/custom/data
```

To generates a plot with `Gnuplot` of the solution (both displacement and velocity) it is needed to run the following in the terminal:
`gnuplot plot.gnu`
A `Gnuplot` code written in `plot.gnu` is run and it takes the solution from the `result\` folder. An ad-hoc result file was previously created with the `void formatGNU()` method defined in `SolverResults` class.

Here we link the GitHub repo of the NAPDE project:
`https://github.com/NNrico/waveDG_XT_RAS`
There is a Matlab script `create_matrices_PACS_project.m` to generate the problem matrices `A.txt`, `b.txt` and the physical coordinates of the d.o.f `coords.txt`.

Finally, we report a practical example that uses the parameters of the previously reported example for `GetPot`. Hence, it will be a `Test 1` example in a domain $\Omega = (0,1) \times (0,1)$, with a mesh of 20 and 20 elements in space and time respectively. It will be solved by RAS Pipelined method with an `AloneOnStride` policy, two cores are available in this example and so two space subdivision are required. To run this test:

```
make run 2 test1
```

If four cores are available it is possible to run the same test with `AloneOnStride` if four space subdivision (*nsubx*) are setted. Otherwise, with `CooperationOnStride` and `CooperationSplitTest` only two space subdivisions (*nsubx*) are required, the user must set them in the `data` file. To run those tests: `make run 4 test1`.

6 Conclusion

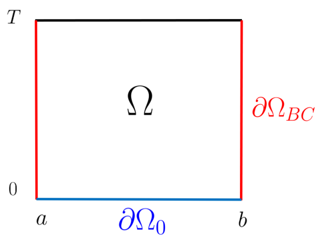
In this project for the course Advanced Programming for Scientific Computing (APSC) we have continued our work started with the project in Numerical Analysis for Partial Differential Equations. Since we are working in a domain decomposition framework it comes naturally to employ parallelization techniques in C++ and this has been the main goal of this project for APSC course.
Our program exploit different parallelization strategies and we design it in a general way in order to easily implement new policies. One of the main step is to assign in a smart way the subdomains in which we decompose the domain to the different available cores. Moreover, our space-time domains gives more flexibility when dealing with subdomains assignment, there are no difference between space and time variables and so it is not required to talk about time parallelization or space parallelization, indeed they could be mixed.
There is a crucial balance to keep in mind between the possibility to potentially decompose many times the domain if there are many available cores and the fact that from a numerical point of view a very refined decomposition leads to a slow information transfer from the initial condition for this hyperbolic problem. The policies that we have report here wants to face this balance from different angles and `CooperationSplitTime` could be a robust and suitable choice.
From the implementation point of view we relied heavily on `Eigen` library to deal with linear algebra and with sparse matrices. Then a massive use of templates is needed to well organized the code and to keep it general. Finally, `MPI` library help us in all parallel computations.
A possible future development could be an extension to a two dimensional space problem ending with a 3D space-time domain. It will be a matter to redefined the prolungation and restriction matrices but, once these structures are defined, all the other methods could be easily extended with a different subdomains assignment.

7 Appendix A - DG Discretization

7.1 Analytical and weak formulation

Let us consider the 1D wave equation in a rectangular space-time domain $\Omega = (a,b) \times (0,T)$. Assume that the wave speed is constant and equal to c. We assign initial conditions on $\partial\Omega_0$ and Dirichlet boundary condition on $\partial\Omega_{BC}$ to $u = u(x,t)$.

$$\begin{cases} u_{tt} - c^2 u_{xx} = f & \text{in } \Omega, \\ u(x,0) = u_0(x) & x \in [a,b], \\ u_t(x,0) = w_0(x) & x \in [a,b], \\ u(a,t) = g(t) & t \in [0,T], \\ u(b,t) = h(t) & t \in [0,T]. \end{cases} \tag{6}$$



To discretize this problem we first split the wave equation in two equations adding the variable $w = w(x,t)$:

$$\begin{aligned} w_t - c^2 u_{xx} &= f, \\ w - u_t &= 0. \end{aligned} \tag{7}$$

This technique allows us to deal with the initial condition on u_t.
In order to write the weak formulation we start subdividing our square domain Ω in cartesian finite elements $\Omega_k \in \mathcal{T}_h$, where $\mathcal{T}_h$ is the triangulation on Ω. We consider space on the x-axis and time on the y-axis.

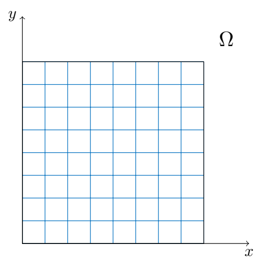


Figure 12: Example of triangulation $\mathcal{T}_h$.

We define now some sets which will help us in the DG framework. Let $\mathcal{E}^0$ be the collection of the interior edges shared between any two elements in $\mathcal{T}_h$ and by $\mathcal{E}^\partial$ the collection all the boundary edges.

$$\begin{aligned} \mathcal{E}^0 &:= \{e\colon e = \partial\Omega_i \cap \partial\Omega_j, \forall\, \Omega_i, \Omega_j \in \mathcal{T}_h\}, \\ \mathcal{E}^\partial &:= \{e\colon e = \partial\Omega_k \cap \partial\Omega, \forall\, \Omega_k \in \mathcal{T}_h\}. \end{aligned}$$

As a consequence the set of all the edges is $\mathcal{E} := \mathcal{E}^0 \cup \mathcal{E}^\partial$. We define also $\mathcal{E}^S$ as the set collecting all the vertical edge and $\mathcal{E}^T$ the set containing all the horizontal edges except the top one at the final time step.

$$\begin{aligned} \mathcal{E}^S &:= \{e \in \mathcal{E}\colon \boldsymbol{n}_e \cdot \boldsymbol{e}_1 = \pm 1\}, \\ \mathcal{E}^T &:= \{e \in \mathcal{E}\colon \boldsymbol{n}_e \cdot \boldsymbol{e}_2 = -1\}, \end{aligned}$$

where $\boldsymbol{n}_e$ is the normal associated to the edge and $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ the canonical basis vector in $\mathbb{R}^2$. We notice that we can associate two opposite normal vectors to an interior edge, which represents the edge of two adjacent element Ω_i and Ω_j. However, boundary edges have only one normal and therefore the $\boldsymbol{n}_e \cdot \boldsymbol{e}_2 = -1$ condition discards only the horizontal edge at the final time step.

Let $\mathcal{E}^{T_0}$ be the set collecting all the boundary edges at the initial time step, and $\mathcal{E}^{\partial S}$ be the set containing all the vertical boundary edges.

$$\begin{aligned}\mathcal{E}^{T_0} &:= \mathcal{E}^{\partial} \cap \mathcal{E}^{T},\\ \mathcal{E}^{\partial S} &:= \mathcal{E}^{\partial} \cap \mathcal{E}^{S}.\end{aligned}$$

Finally, we need a space V of test functions. Considering the triangulation $\mathcal{T}_h$ we take $V = V_{DG}$ as the space of test function for our problem.

$$V_{DG} := \{v \in L^2(\Omega) \colon v|_{\Omega_k} \in \mathbb{P}^r(\Omega_k), \forall \Omega_k \in \mathcal{T}_h\}.$$

We notice that a function $\psi \in V_{DG}$ can jump across $\mathcal{E}^0$ but is *single-valued* on $\mathcal{E}^{\partial}$. We also define the average and jump operator for a function $\psi \in V_{DG}$ on the edge $e := \partial\Omega_1 \cap \partial\Omega_2$. Considering $\psi_i := \psi|_{\partial\Omega_i}$ we have

$$\{\!\{\psi\}\!\} = \frac{1}{2}(\psi_1 + \psi_2), \quad [\![\psi]\!] = \psi_1 \boldsymbol{n}_1 + \psi_2 \boldsymbol{n}_2,$$

where $\boldsymbol{n}_i$ is the normal outwards of $\partial\Omega_i$. At the boundary we have $\{\!\{\psi\}\!\} = \psi$ and $[\![\psi]\!] = \psi\,\boldsymbol{n}$.
We are ready to write the weak formulation of problem (7). Let $V = V_{DG}$ and $u, w \in V$,

$$\sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} w_t v \, d\Omega - c^2 \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_{xx} v \, d\Omega = \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} f v \, d\Omega \qquad \forall v \in V, \tag{8}$$

$$\sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_t q \, d\Omega - \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} w q \, d\Omega = 0 \qquad \forall q \in V. \tag{9}$$

We now focus on the second term of (8) and perform integration by parts:

$$\sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_{xx} v \, d\Omega = - \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_x v_x \, d\Omega + \sum_{\Omega_k \in \mathcal{T}_h} \left(\int_{\Gamma_{2k}} u_x v \; n_x \, dl + \int_{\Gamma_{1k}} u_x v \; n_x \, dl \right), \tag{10}$$

where Γ_{1k} and Γ_{2k} are respectively the left and right vertical boundary associated to the element Ω_k.

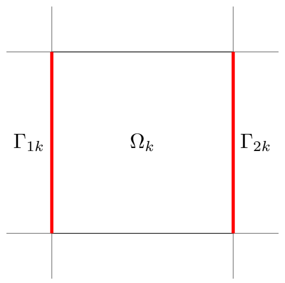


Figure 13: Γ_{2k} on the right vertical border and Γ_{1k} on the left vertical border of the element Ω_k.

The last term in (10) can be written using the jump and average operators as

$$\sum_{\Omega_k\in\mathcal{T}_h}\left(\int_{\Gamma_{2k}} u_x v\; n_x\, dl + \int_{\Gamma_{1k}} u_x v\; n_x\, dl\right) = \sum_{\Gamma_i\in\mathcal{E}^S}\int_{\Gamma_i} \{\!\{u_x\}\!\}[\![v]\!]\, dl + \sum_{\Gamma_i\in\mathcal{E}^0\cap\mathcal{E}^S}\int_{\Gamma_i} [\![u_x]\!]\{\!\{v\}\!\}\, dl. \tag{11}$$

The last term in (11) is null since for exact solution the jump on internal faces of u_x is zero.
To simplify the notation we write $\{\!\{u_x\}\!\}$ instead of $\{\!\{\nabla u\}\!\}$ since for these integrals on the vertical edge the only component remaining is u_x.

Using (11) (10) in (8) we obtain a first possible DG weak formulation

$$\sum_{\Omega_k\in\mathcal{T}_h}\int_{\Omega_k} w_t v\, d\Omega + c^2\sum_{\Omega_k\in\mathcal{T}_h}\int_{\Omega_k} u_x v_x\, d\Omega - c^2\sum_{\Gamma_i\in\mathcal{E}^S}\int_{\Gamma_i}\{\!\{u_x\}\!\}[\![v]\!]\, dl = \sum_{\Omega_k\in\mathcal{T}_h}\int_{\Omega_k} f v\, d\Omega \qquad \forall v\in V, \tag{12}$$

$$\sum_{\Omega_k\in\mathcal{T}_h}\int_{\Omega_k} u_t q\, d\Omega - \sum_{\Omega_k\in\mathcal{T}_h}\int_{\Omega_k} w q\, d\Omega = 0 \qquad \forall q\in V. \tag{13}$$

7.2 Stability terms

To this formulation we add four stability terms which don not alter the consistency of the numerical method:

- A term for symmetry in (12)

$$-c^2\sum_{\Gamma_i\in\mathcal{E}^S}\int_{\Gamma_i}\{\!\{v_x\}\!\}[\![u]\!]\, dl.$$

- A space stability term in (12) with μ large enough to be chosen. This term penalizes high jumps of the solution on the space interfaces between elements.

$$\mu\sum_{\Gamma_i\in\mathcal{E}^S}\int_{\Gamma_i}[\![u]\!]\cdot[\![v]\!]\, dl.$$

- Two stabilisation terms in time in (12) and (13) respectively. This term also penalizes high jumps of the solution on the lower time interface of each element.

$$\sum_{\Gamma_i\in\mathcal{E}^T}\int_{\Gamma_i}[\![w]\!]v\, dl, \qquad \sum_{\Gamma_i\in\mathcal{E}^T}\int_{\Gamma_i}[\![u]\!]v\, dl.$$

  by $[\![\psi]\!]v$ we indicate $(\psi^+ - \psi^-)\, v^+$ where ψ^+ and ψ^- are the bases functions in the above and bottom element sharing an horizontal edge.

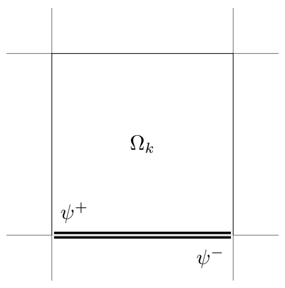


Figure 14: ψ^+ associated to Ω_k, ψ^- associated to its neighbour below.

Finally, we reach this final formulation.

$$\sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} w_t v \, d\Omega + c^2 \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_x v_x \, d\Omega - c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{u_x\}\!\} [\![v]\!] \, dl - c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{v_x\}\!\} [\![u]\!] \, dl$$

$$\sum_{\Gamma_i \in \mathcal{E}^T} \int_{\Gamma_i} [\![w]\!] v \, dl + \mu \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} [\![u]\!] \cdot [\![v]\!] \, dl = \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} f v \, d\Omega \qquad \forall v \in V, \quad (14)$$

$$\sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_t q \, d\Omega - \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} w q \, d\Omega + \sum_{\Gamma_i \in \mathcal{E}^T} \int_{\Gamma_i} [\![u]\!] q \, dl = 0 \qquad \forall q \in V. \quad (15)$$

7.3 Boundary terms

It is important to clarify how the terms with jumps and averages behave at the boundary and understand how the boundary conditions are weakly imposed. Thus we show how those terms are explicitly computed at the boundary in the following:

$$\sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} \{\!\{u_x\}\!\} [\![v]\!] \, dl = \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} u_x v \, dl, \quad (16)$$

$$\sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} \{\!\{v_x\}\!\} [\![u]\!] \, dl = \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} v_x (u - g) \, n_x \, dl = \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} v_x u \, n_x \, dl - \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} v_x g \, n_x \, dl. \quad (17)$$

The last term of (17) is a known quantity and goes to the right hand side imposing the spatial boundary condition on u.

$$\mu \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} [\![u]\!] \cdot [\![v]\!] \, dl = \mu \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} (u - g) v \, n_x \, dl = \mu \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} u v \, n_x \, dl - \mu \sum_{\Gamma_i \in \mathcal{E}^{\partial S}} \int_{\Gamma_i} g v \, n_x \, dl. \quad (18)$$

The last term of (18) is a known quantity and goes to the right hand side. Also here we have a weakly imposition on u of the spatial boundary condition.

$$\sum_{\Gamma_i \in \mathcal{E}^{T_0}} \int_{\Gamma_i} [\![u]\!] v \, dl = \sum_{\Gamma_i \in \mathcal{E}^{T_0}} \int_{\Gamma_i} (u_0 - u) v \, dl = \sum_{\Gamma_i \in \mathcal{E}^{T_0}} \int_{\Gamma_i} u_0 v \, dl - \sum_{\Gamma_i \in \mathcal{E}^{T_0}} \int_{\Gamma_i} u v \, dl. \quad (19)$$

The first term of (19) is a known quantity and goes to the right hand side and impose the initial condition weakly on u. A similar result is obtained if we compute the term associated to w, in that case we weakly impose the initial condition on u_t.

7.4 IP and IPH formulation

Some of the terms in (14) can be grouped in an Interior Penalty (IP) term corresponding to a symmetric bilinear form $a_{\mathrm{IP}} \colon V \times V \to \mathbb{R}$,

$$a_{\mathrm{IP}}(u, v) := c^2 \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_x v_x \, d\Omega - c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{u_x\}\!\} [\![v]\!] \, dl$$

$$- c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{v_x\}\!\} [\![u]\!] \, dl + \mu \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} [\![u]\!] \cdot [\![v]\!] \, dl \qquad \forall v \in V. \quad (20)$$

By adding an additional consistent term affecting the interior edges we obtain the Hybridizable Interior Penalty (IPH) formulation:

$$a_{\mathrm{IPH}}(u,v) := c^2 \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} u_x v_x \, d\Omega - c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{u_x\}\!\} [\![v]\!] \, dl - c^2 \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} \{\!\{v_x\}\!\} [\![u]\!] \, dl$$
$$+ \frac{\mu}{2} \sum_{\Gamma_i \in \mathcal{E}^S} \int_{\Gamma_i} [\![u]\!] \cdot [\![v]\!] \, dl - \frac{1}{2\mu} \sum_{\Gamma_i \in \mathcal{E}^0 \cap \mathcal{E}^S} \int_{\Gamma_i} [\![u_x]\!] \cdot [\![v_x]\!] \, dl \qquad \forall v \in V. \quad (21)$$

We consider these formulations since we will see that they are linked to different transmission conditions when applying domain decomposition methods. These formulations are studied in [4] and in [6] for reaction diffusion problems in spatial domain, and we will investigate their behaviour in a space-time framework.

7.5 Algebraic formulation

It is possible to write the remaining terms in (14) (15) compactly by defining some operators:

$$\mathrm{b}(w,v) := \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} w_t v \, d\Omega \quad \forall w, v \in V, \quad (22)$$

$$\mathrm{c}(w,v) := \sum_{\Gamma_i \in \mathcal{E}^T} \int_{\Gamma_i} [\![w]\!] v \, dl \quad \forall w, v \in V, \quad (23)$$

$$\mathrm{m}(w,v) := \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} wv \, d\Omega \quad \forall w, v \in V, \quad (24)$$

$$\mathrm{F}(v) := \sum_{\Omega_k \in \mathcal{T}_h} \int_{\Omega_k} fv \, d\Omega \qquad \forall v \in V. \quad (25)$$

Therefore (14) and (15) become

$$\mathrm{b}(w,v) + \mathrm{a}_{\mathrm{IP/IPH}}(u,v) + \mathrm{c}(w,v) = \mathrm{F}(v) \qquad \forall v \in V, \quad (26)$$
$$\mathrm{b}(u,q) - \mathrm{m}(w,q) + \mathrm{c}(u,q) = 0 \qquad \forall q \in V. \quad (27)$$

Each form written can be associated to matrix and vector multiplication compactly written as

$$\begin{bmatrix} A_{IP/IPH} & B+C \\ B+C & -M \end{bmatrix} \begin{bmatrix} U \\ W \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} + \begin{bmatrix} F_{BC1} \\ F_{BC2} \end{bmatrix}, \quad (28)$$

where F_{BC1} and F_{BC2} are the vectors arising from o the boundary condition.

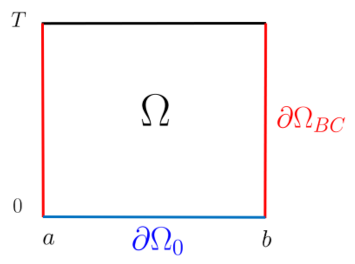
In the end the system is in the form of $Au = f$.

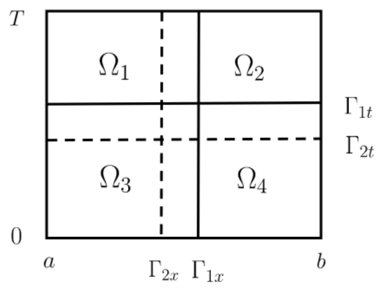
8 Appendix B - Domain Decomposition methods

We consider the wave equation in 1D discretized with a space-time approach using Discontinuos Galerkin method. This discretization leads to a solution of a linear system $A\boldsymbol{u} = \boldsymbol{f}$ for all the space-time domain $\Omega = (a,b) \times (0,T)$.
To formalize the problem, we decompose the domain Ω considering one division in time and space obtaining Ω_i with $i = 1,..,M$ where $M = 4$ in the following figure.
Let $\boldsymbol{z}_i := (u_i, w_i)$ and let $\mathcal{L}$ be the operator corresponding to (6). We consider M different subproblems

of the form:

$$\begin{cases} \mathcal{L}\boldsymbol{z}_i = \boldsymbol{f} & \Omega_i, \\ \boldsymbol{z}_i = \boldsymbol{z}_{i+2} & \Gamma_{it} \quad i = 1, 2, \\ \mathcal{B}(\boldsymbol{z}_i) = \mathcal{B}(\boldsymbol{z}_{i+1}) & \Gamma_{ix} \quad i = 1, 3, \\ \boldsymbol{z}_i(x, 0) = (u_0(x), w_0(x)) & \partial\Omega_i \cap \partial\Omega_0, \\ u_i(a, t) = g(t),\ u_i(b, t) = h(t) & \partial\Omega_i \cap \partial\Omega_{BC}. \end{cases} \tag{29}$$





8.1 Space-time decomposition methods

For the solution of the linear system $A\boldsymbol{u} = \boldsymbol{f}$ we will follow three different approaches:

- Restricted Additive Schwarz (RAS) method that changes the restriction matrices of Additive Schwarz (AS) method [5];
- GMRES method with the preconditioner identified by RAS (this is not implemented in our C++ code);
- Pipelined RAS that modify RAS based on the idea that the solution evolves in the space time domain in the time direction.

8.1.1 RAS as an iterative method and as a preconditioner for GMRES

The discretization of the partial differential equation leads to a linear system of equations (30) that can be solved with a stationary iterative method (31). Let M be the number of subdomains in which we decompose the domain. R_j and R_j^T, with $j = 1, ..., M$, are rectangular matrices that correspond to the restriction and prolongation operators, respectively. The local matrices A_j are related to A through the restriction and prolongation matrices: $A_j = R_j A R_j^T$.

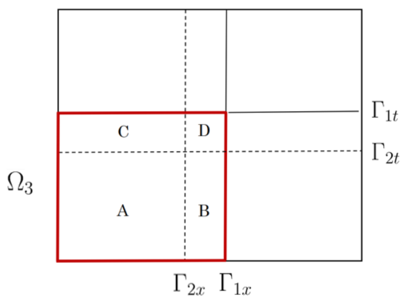
$$A\boldsymbol{u} = \boldsymbol{f}. \tag{30}$$

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + P^{-1}(\boldsymbol{f} - A\boldsymbol{u}^n). \tag{31}$$

For the Additive Schwarz method P_{as} is used as the preconditioner of (31)

$$P_{as}^{-1} = \sum_{j=1}^{M} R_j^T A_j^{-1} R_j. \tag{32}$$

Additive Schwarz methods suffers in convergence [5] since the prolongation operator considers the contribution of both the neighbouring subdomains. Therefore, in the overlapping regions the final solution will be the sum of the two solutions computed in the subdomains associated to the overlap. To fix this issue Restricted Additive Schwarz method weights the two solutions in the overlap. Those weight factors are in the prolongation matrix $\tilde{R}^T$. This weight is added both in the time and space overlapping and we fixed it to 0.5.



$$\eta = \begin{array}{c} \qquad\qquad\quad \Gamma_{2x} \\ \left[\begin{array}{cccc:cc} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1/2 & \\ & & & & & 1/2 \end{array}\right] \end{array}$$

$$\xi = \begin{array}{c} \left[\begin{array}{cccc:cc} 1/2 & & & & & \\ & 1/2 & & & & \\ & & 1/2 & & & \\ & & & 1/2 & & \\ & & & & 1/4 & \\ & & & & & 1/4 \end{array}\right] \\ \qquad\qquad\quad \Gamma_{2x} \end{array}$$

$$\tilde{R}_3 = \begin{array}{c} \qquad\qquad\quad \Gamma_{2t} \quad \Gamma_{1t} \\ \left[\begin{array}{cccc:cc|c} \eta & & & & & & \\ & \eta & & & & & \\ & & \eta & & & & \\ & & & \eta & & & \\ & & & & \xi & & \\ & & & & & \xi & \end{array}\right] \end{array}$$

Figure 15: Restriction matrix $\tilde{R}_3$. We can observe the different weights corresponding to the different overlapped zone B,C and D. For example, D is common to all subdomains and therefore its contribute is weighted by 4.

Now we can define the RAS preconditioner:

$$P_{ras}^{-1} = \sum_{j=1}^{M} \tilde{R_j}^T A_j^{-1} R_j. \tag{33}$$

Given our cartesian domain we have two degrees of freedom regarding the decomposition in subdomains: the number of subdomains and the overlap magnitude. The domain is divided based on the number of threads available and then each subdomain is extended in the neighbouring subdomains by few elements to create the overlapping. The overlap is often limited because with an increasing overlap the method solves a bigger zone more times. For example, for a domain $\Omega = (0, 1) \times (0, 5)$, we usually choose 2 subdomains in space and 10 in time with an overlap of 0.2 and 0.05 in space and time respectively.
The RAS method can be used as preconditioner to accelerate the convergence of Krylov methods, like GMRES. A good preconditioner must be a good approximation of A to reduce the magnitude of the condition number. In domain decomposition method with restriction and prolongation operator we give a good representation of A. We keep the idea of decomposing the domain to pass a good preconditioner to an efficient Krylov method.

It is important to observe that the preconditioning step is fully parallelizable as it involves the solution of M independent system, one for each local matrix A_j.

8.1.2 Pipelined RAS

For both the approaches presented before the preconditioner solves all subdomain problems at each iteration. We are not using the fact that in space-time domains the correct information travels in the time direction. This means that the first subdomains in time will be the first to be exactly solved. When we compute the inverse of the preconditioner times the residual, the problem is solved at each subdomain Ω_j, but at the beginning it will be impossible to have a correct solution in the part of the domain corresponding to the last time instants. Considering only few subdomains at the time permits to avoid useless computation to solve system for which is impossible to obtain a good solution. From this idea we built a Pipelined space-time RAS method, it has the same framework of RAS method but the preconditioner P_{pipe} considers only K subdomains:

$$P_{pipe}^{-1} = \sum_{j=1}^{K} \tilde{R_j}^T A_j^{-1} R_j. \tag{34}$$

We have the freedom to decide which subdomains are used at each iteration, this must be chosen according to how fast the wave propagation moves from the initial time and according to the velocity of the convergence. We will refer to the subdomains considered at each iteration as subdomains' window S.

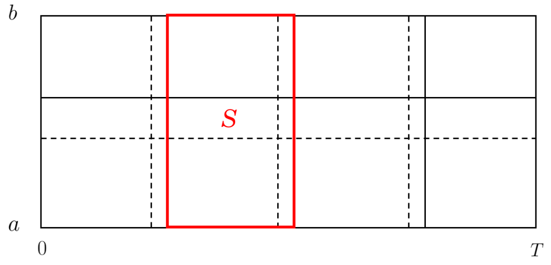


Figure 16: Subdomains' window S. Imagine dividing the entire domain in the time direction and consider only the subdomains in the red zone S. Then, using the division in space, assign the subdomains at each thread. Notice that the domain is not oriented as usual but there is time on the horizontal axis.

At first S is set at the the initial time. Then we run the iterations until the residual in the left part of the zone S reaches a defined tolerance and then move S in time. In this case we have to choose a fixed time interval that corresponds to a certain number of subdomains in time, then we checked if all the subdomains at the left boundary of S are exactly solved and, only after, shift the subdomains' window to the right. This is a very conservative way to proceed since we can be blocked in one window for many iterations and we could lose time in vain.
To face this problem we implemented a more 'aggressive' version of Pipelined RAS: we do not wait until all the subdomains in the left part of S are solved and we update the right boundary after a defined number of iterations. However, we continue to check the left part and we shift it to the right when we achieved a fixed tolerance. In this way the zone of highlighted subdomains S is freer to move, we are not blocked in only one zone and the right side can evolve faster exploring all the subdomains. Then if it moves too fast it will be necessary to wait for all the previous subdomains to be exactly solved but certainly the method will solve less subdomains with respect to the original RAS.

The use of this approach with a moving pipeline brings benefits to the parallel resolution. We can choose the number of space subdomains according to the number of threads available and then assign to each of them one space subdomain. In this way S is equally divided and with one iteration we will solve all the subdomains' window. Compared to the previous method we assign at each thread much less subdomains to solve without lose quality in the convergence.

8.2 Trasmission conditions

IP and IPH formulations are associated to different transmission conditions operators $\mathcal{B}(\cdot)$ between subdomains.
For the analysis of our problem, we have followed [6] which study the effect of IP and IPH transmission condition on convergence. In order to investigate if this effect still holds in a space-time domain, we

want to apply the same ideas and a similar proof on our problem.
Starting from (29), we apply RAS method without overlapping subdomains and we study the corresponding transmission conditions of the two formulations. We recall that this is not our typical framework since we usually consider overlap. A possible further development would be to prove the transmission conditions with overlapping subdomains. However, it is reasonable to expect the same transmission conditions in both cases, so we proceed with the proof of transmission conditions in a non-overlapping decomposition.

8.2.1 Trasmission conditions for IP and IPH

To obtain the transmission condition operator $\mathcal{B}(\cdot)$ on the space interface, we will study only the bilinear form $a(\cdot,\cdot)$ corresponding to the space derivative in (6). This form is the one which characterizes the two different formulations IP and IPH. Let Ω be a domain decomposed without overlap and let Γ be the space interface between two subdomains. The space vertical edges are denoted by $\mathcal{E}^S$ and $\mathcal{E}^{S0}$ correspond to the internal one. With $(\cdot,\cdot)_{\Omega_i}$ we refer to $\int_{\Omega_i} \; d\Omega$ and with $\langle\cdot,\cdot\rangle_{\mathcal{E}_i^S}$ we refer to $\int_{\mathcal{E}_i^S} \; dl$. Let first consider IP formulation on the subdomain Ω_i

$$\begin{aligned}\frac{1}{c^2}a_{\text{IP}}(u_i,v_i) &= (u_{ix}v_{ix})_{\Omega_i}\langle\{\!\{u_x\}\!\},[\![v]\!]\rangle_{\mathcal{E}_i^S} - \langle\{\!\{v_{ix}\}\!\},[\![u_i]\!]\rangle_{\mathcal{E}_i^S} + \mu\langle[\![u_i]\!],[\![v_i]\!]\rangle_{\mathcal{E}_i^S} \qquad \forall u_i,v_i\in V(\Omega_i)\\
&= (u_{ix}v_{ix})_{\Omega_i} - \langle\ \{\!\{u_{ix}\}\!\},[\![v_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma} - \langle\ \{\!\{v_{ix}\}\!\},[\![u_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma} + \mu\langle[\![u_i]\!],[\![v_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma}\\
&\quad + \int_\Gamma -u_i\frac{\partial v_i}{\partial n_i} - v_i\frac{\partial u_i}{\partial n_i} + \mu u_i v_i dl \qquad \forall u_i,v_i\in V(\Omega_i).\end{aligned}$$

Integrating back the volume term, we arrive at

$$\begin{aligned}\frac{1}{c^2}a_{\text{IP}}(u_i,v_i) &= (-u_{ixx},v_i)_{\Omega_i} + \langle\ \{\!\{u_{ix}\}\!\},[\![v_i]\!]\rangle_{\mathcal{E}_i^S} - \langle\ \{\!\{u_{ix}\}\!\},[\![v_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma} + \langle\ \{\!\{v_i\}\!\},[\![u_{ix}]\!]\rangle_{\mathcal{E}_i^{S0}}\\
&\quad -\langle\ \{\!\{v_{ix}\}\!\},[\![u_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma} + \langle\mu[\![u_i]\!],[\![v_i]\!]\rangle_{\mathcal{E}_i^S\setminus\Gamma} + \int_\Gamma -u_i\frac{\partial v_i}{\partial n_i} - v_i\frac{\partial u_i}{\partial n_i} + \mu u_i v_i dl \qquad \forall u_i,v_i\in V(\Omega_i)\\
&= (-u_{ixx},v_i)_{\Omega_i} + \langle\ \{\!\{v_i\}\!\},[\![u_{ix}]\!]\rangle_{\mathcal{E}_i^{S0}} + \langle[\![u_i]\!],\mu[\![u_i]\!] -\ \{\!\{v_{ix}\}\!\}\rangle_{\mathcal{E}_i^S\setminus\Gamma}\\
&\quad + \int_\Gamma -u_i\frac{\partial v_i}{\partial n_i} - v_i\frac{\partial u_i}{\partial n_i} + v_i\frac{\partial u_i}{\partial n_i} + \mu u_i v_i dl \qquad \forall u_i,v_i\in V(\Omega_i)\\
&= (-u_{ixx},v_i)_{\Omega_i} + \langle\ \{\!\{v_i\}\!\},[\![u_{ix}]\!]\rangle_{\mathcal{E}_i^{S0}} + \langle[\![u_i]\!],\mu[\![u_i]\!] -\ \{\!\{v_{ix}\}\!\}\rangle_{\mathcal{E}_i^S\setminus\Gamma}\\
&\quad + \int_\Gamma u_i(\mu v_i - \frac{\partial v_i}{\partial n_i})\ dl \qquad \forall u_i,v_i\in V(\Omega_i).\end{aligned}$$

If we however restrict u to the neighboring subdomain Ω_j, such that $\Gamma = \partial\Omega_i \setminus \partial\Omega_j$, we obtain

$$\frac{1}{c^2}a_{\text{IP}}(u_j,v_i) = \int_\Gamma u_j(\mu v_i - \frac{\partial v_i}{\partial n_i})\ dl.$$

Now consider the problem: given $u_j \in H^2(\Omega_j)$ and $\boldsymbol{f}\in[L^2(\Omega)]^2$, find $u_i\in H^2(\Omega_i)$ such that

$$(\mathcal{L}(u_i,0),(v_i,0)) + (\mathcal{L}(u_j,0),(v_i,0)) = (\boldsymbol{f},(v_i,0)) \qquad \forall v_i\in V(\Omega_i).$$

Then one can use Thm.1 in [7] and conclude that all the jumps and the averages are equal to zero since $u_i,v_i\in H^2(\Omega_i)$. Moreover, regarding the integrals over Ω_i, $\mathcal{L}u_i$ cancels out with the force term and it remains only the integral over the interface Γ

$$\int_\Gamma u_i(\mu v_i - \frac{\partial v_i}{\partial n_i}) - u_j(\mu v_i - \frac{\partial v_i}{\partial n_i})\ dl = 0 \qquad \forall v_i\in V(\Omega_i).$$

Finally, we obtain the transmission condition operator $\mathcal{B}(\cdot)$ for IP formulation

$$u_i = u_j \qquad on\ \Gamma.$$

For IPH, by following the same steps as above and following in [6] (pag 39), we obtain a Robin transmission condition

$$\mu u_i + \frac{\partial u_i}{\partial n_i} = \mu u_j + \frac{\partial u_j}{\partial n_i} \qquad on\ \Gamma.$$

The DG stability coefficient μ appears only in the IPH transmission condition as the Robin parameter, instead for IP we obtain a Dirichlet condition. It is proved in [6], by analyzing the convergence factor with Fourier series, that there exists an optimal choice of μ for convergence. We want to investigate through numerical tests how μ influences convergence also in our case.
It is reasonable to expect that for high values of μ a Robin condition behave as a Dirichlet one since the term with the normal derivative could have a lower magnitude and can be neglected with respect than the other one. This is the case in our problem that requires high values for the stability coefficient, therefore the formulations are similar and we don't expect the possible advantages for a Robin condition. However, we will further investigate differences in number of iteration and solved subdomains between the two formulations to understand if a Robin condition is more effective in terms of convergence velocity.

References

[1] *C++ Primer*, Stanley B. Lippman, Josée Lajoie, Barbara E. Moo, fifth edition, 2013

[2] *Parallel Programming in MPI and OpenMP The Art of HPC*, Victor Eijkhout, volume 2, 2022

[3] *https://learn.cineca.it/*, Cineca Academy

[4] *A space-time discontinuous Galerkin method for the elastic wave equation*, Paola F. Antonietti, Ilario Mazzieri, Francesco Migliorini. Journal of Computational Physics, Volume 419, 2020

[5] *Optimized multiplicative, additive and restricted additive Schwarz preconditioning*, A. St-Cyr, M.J. Gander and S.J. Thomas, SIAM J. Sci. Comput., Vol. 29, No. 6, pp. 2402-2425, 2007

[6] *Analysis of Schwarz methods for discontinuous Galerkin discretizations*, HAJIAN, Soheil. Thèse de doctorat : Univ. Genève, 2015, no. Sc. 4795

[7] *Stabilization mechanisms in discontinuous Galerkin finite element methods*, F. Brezzi, B. Cockburn, L.D. Marini, and E. Suli. Computer Methods in Applied Mechanics and Engineering, 195 (2006), pp. 3293 - 3310.

[8] *Iterative methods and preconditioners for systems of linear equations*, Gabriele Ciaramella, Martin J Gander. Society for Industrial and Applied Mathematics, 2022.

[9] *Unmapped tent pitching schemes by waveform relaxation*, Gabriele Ciaramella, Martin J. Gander, Ilario Mazzieri. MOX Report Collection, 2022.

[10] *hp-Version space-time discontinuous Galerkin methods for parabolic problems on prismatic meshes*, Andrea Cangiani, Zhaonan Dong, Emmanuil H. Georgoulis. arXiv, 2016.

Acknowledgements

We would like to thank Paolo Joseph Baioni for his support during this work.