Efficient Implicit Parallel Patterns for Geographic Information System

Kevin Bourgeois, Sophie Robert, Sébastien Limet, Victor Essayan

Efficient Implicit Parallel Patterns for Geographic Information System

Kevin Bourgeois[1,2], Sophie Robert[1], Sébastien Limet[1], and Victor Essayan[2]

[1] Univ.Orléans, INSA Centre Val de Loire, LIFO EA4022, Orléans, France
(kevin.bourgeois, sophie.robert, sebastien.limet)@univ-orleans.fr
[2] Géo-Hyd (Antea Group), Olivet, France
(kevin.bourgeois, victor.essayan)@anteagroup.com

Abstract

With the data growth, the need to parallelize treatments become crucial in numerous domains. But for non-specialists it is still difficult to tackle parallelism technicalities as data distribution, communications or load balancing. For the geoscience domain we propose a solution based on implicit parallel patterns. These patterns are abstract models for a class of algorithms which can be customized and automatically transformed in a parallel execution. In this paper, we describe a pattern for stencil computation and a novel pattern dealing with computation following a pre-defined order. They are particularly used in geosciences and we illustrate them with the flow direction and the flow accumulation computations.

Keywords: Parallel programming, Implicit parallelism, Performance, GIS

1 Introduction

During the last decades, fast progress in capturing and measuring methods in every scientific domains made the amount of available data grows in a dramatic way. Such a revolution allows the scientists to refine their models and simulations. However computing on such amounts of data requires the use of powerful computers, such as clusters, to get reasonable run-times. One of the main locks in many sciences is to gather in a single man or at least in a single team, high skills in both the specific scientific domain and in high performance computing. One solution to address this issue consists in providing to scientists, who are able to program in sequential way, tools which help them to (semi-)automatically derive efficient parallel programs from a program that hides, more or less, technicalities due to parallelism. Currently, great efforts are done in computer science to define efficient tools to help non-HPC experts to program efficiently parallel computers. There are mainly two families of solutions. On one hand are dedicated libraries that provide turnkey implementations of classical and domain-specific functions. On the other hand, skeleton programming or pattern programming provide an abstract model of an algorithm that can be customized and transformed into a parallel program. Often, these

approaches are associated with a Domain Specific Language (DSL) to describe, in a familiar way for the scientist, the skeleton and its inputs.

Geosciences are no exception and amount of data is a major concern even for SMEs or public labs working on geographic information systems (GIS) to conduct environmental studies like soil erosion or analysis of the water cycle. The work presented in this paper addresses the implicit parallelism for geoscientists. It focuses on two parallel patterns widely used when computing on large GIS represented by 2D rasters. However this work takes place in a more general project that aims at providing an implicit parallel computing framework dedicated to geoscientists. This framework illustrated Figure 1 relies on implicit parallelism associated to a DSL. It is split into three main layers from a high level user interface to program the algorithm to a low level data system to efficiently manage data on disks. These layers are connected by an implicit parallelism pattern layer. This framework targets computer architectures such as clusters of small or middle size (i.e. from 10 to 100 nodes) as those frequently found in SMEs or scientific labs.

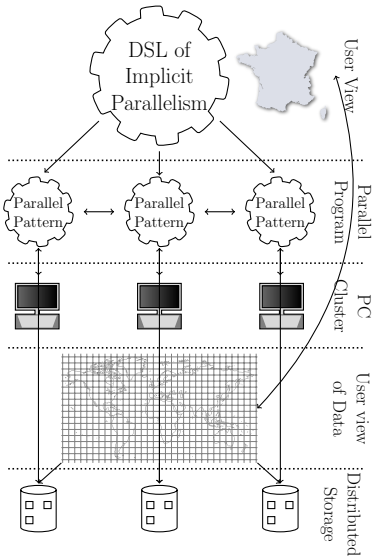


Figure 1: Implicit Parallel Framework for GIS

At the top level, the geoscientist has a classical view of his/her GIS and programs in a classical way using a DSL embedded in Python. This DSL is based on libraries to compute on raster and/or vector datasets using some predefined patterns. At the low level, the datasets are stored in a distributed file system and the framework provides a layer that allows the user to efficiently select data he/she requires as well as to store intermediate results. At the middle, a layer is responsible to make the connection between the two ends of the framework and to produce efficient parallel programs from the ones written by the user. This middle layer relies on the implicit parallel pattern programming interface written in C++ and using MPI parallel library. This middle layer uses the data layer library to access and store data. Different patterns have been implemented corresponding to different classes of algorithms frequently used in geosciences. They are optimized to efficiently parallelize programs according to their class. From the program written by the user with the DSL, it is possible to determine the patterns to apply in the parallel program and a C++ program is automatically derived using them.

In this paper we present two patterns which represent very large classes of programs usually used in geosciences. The first pattern is known as *stencil computing* and consists in computing the value of a cell in a raster according to a bounded neighborhood around the dataset. Such a pattern is widely used for computing simulations in an iterative way (i.e. the computation is repeatedly applied at each time step). Computations on GIS we focus on, are less iterative but are applied on very large dataset. The usual way to help the programmer in implementing such a calculation is to provide a pattern where he/she describes functions (also called kernels) that must be applied on the cell and the neighborhood shape. From these information it is possible to generate efficient parallel programs that do the computation.

The second pattern is more specific to geosciences and consists in computing cell values in an order defined by the topology of the terrain on which the calculation is performed. One of the typical computations is the flow accumulation that consists in computing for each cell of a terrain how many cells it drains according to the flow direction. Such computation is not easy to parallelize efficiently since the value of a cell depends on an arbitrary number of cells which could be located anywhere in the terrain. Our pattern aims at hiding the parallelism to programmers by providing a way to describe such a computation by means of a function and a pre-order describing the dependencies between cells. As for the stencil pattern, it is possible to derive an efficient parallel program from these information.

The rest of the paper is organized as follows. Section 2 presents some related work. The two implicit parallelism patterns are described Section 3 and the experimental results are given Section 4. Finally, Section 5 concludes the paper and draw some perspectives.

2 Related Work

Many problems in the geocomputation domain use raster datatype, which is a regular mesh with one or many layers and vector datatype which is a geometric shape representing various geographic elements such as rivers. The steady increasing size of the rasters and the computational complexity of the problems require the use of parallel computers which are difficult to program.

The GIS specialists are used to code their algorithm in a sequential way, but the growing amount of data requires to parallelize their codes. Most of the parallel tools focus on raster datatype. For example, the PaRGO library [14] tries to hide most of the parallel aspects like the communications and the load-balancing by encapsulating the parallel details, it supports local and global operations and targets all architectures. It allows the user to implement operators that can be applied in parallel over the mesh. pRPL [10] is another raster parallel library implemented in C++ and template based. It supports local and global operators, and allows various decomposition (like the quad tree decomposition). These two libraries are implemented in C++ and make use of templates which can be very difficult for a GIS specialists to understand as he/she is more familiar with higher level language like Python used in ArcGIS. Hence, alternatives can be found like the RasDaMan database [2] that is an array database system to make terrain analyses with a query language close to SQL called *rasql*. Moreover, most of the GIS software, like ArcGIS, SAGA GIS or Grass, propose built-in parallel implementations of terrain analysis problems, like TerraFlow [1] in Grass to compute flow accumulation on massive raster, or the IDW interpolation [12] in ArcGIS. Other tools can be added to GIS software or used alone like TauDEM [16] to compute watershed and flow accumulation on large rasters.

A way to hide most of the parallel complexity is to use skeleton programming. The skeleton programming [3, 15] allows to manipulate high order function that can be interpreted as a program template that can be parametrized in sequential problem-specific code, and which can

be optimized for specific platforms. Those solutions hide most of the parallelization as they usually hide the navigation over the data structures and allow very good performances. The skeletons proposed by Murray Cole are very recurrent parallelization schema, like *map* which applies a function on each element of an array. Skeletons can be nested in order to create more complexes skeletons to address more complexes problems. The MapReduce library of Google [7] has resurfaced the skeleton with very good performances for large data. There is no real formal description of what a skeleton is and several implementations of skeleton libraries can be found, for example FastFlow [5], still in development that aims to support highly efficient stream parallel computation for the shared-memory architectures. SkelGIS [4] is a pattern based library for efficient scientific simulations on mesh initially design for GIS purposes.

Our framework uses something similar to the skeleton, which we call patterns, but they do not hide completely the parallelization and cannot be nested as the aim is to provide a higher level tool to hide the the complexity of parallelism and unlike FastFlow we target the distributed-memory architecture

One can also restrict the range of problems to parallelize by identifying a specific domain of application. The domain specific language (DSL) is a programming language design to address the specific problems of a domain. Because they address a restricted range of problems, a lot of optimizations can be done. Liszt [9] is a DSL for solving partial-differential equations on general mesh in parallel, it implements a subset of Scala [13] running thought the JVM. Few DSLs exist in geosciences [8, 6, 11] but they do not aim at parallel architectures.

Our framework tries to take advantages of each solutions presented above. By defining a DSL on top of the layers, we make sure that our framework is easy to use and fairly general to meet a lot of problems. The patterns layer allows us to take advantages of skeleton programming and ensure good performance by taking care of the communications and the load-balancing.

3 Parallel Patterns

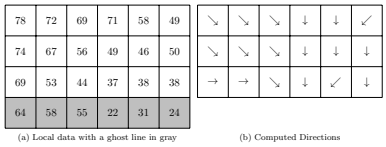
One way to smooth out the difficulties of designing a parallel program, especially considering the complexity of current architectures, is to use of patterns. By separating the data distribution, the communications and the synchronizations of parallel solutions, common parallel patterns can be isolated and applied on various problems. This idea is similar to well-known *design patterns* [17] in the sequential world. Thus, the pattern is an abstract model for a class of algorithms. It can be specialized from specific functions brought by the scientist. Then, the pattern is automatically transformed into a parallel program executable on a parallel computer.

For GIS, we identified two very important patterns and implemented them. The first one is the *stencil pattern* which corresponds to a very common computation schema. It is widely used in many domains like image processing or numerical simulations. In Geosciences, it is used for computations like the flow direction algorithm or simulations based on cellular automata. Stencil based computation has been intensively studied for several decades. The second pattern has not really been identified as such in the literature but many algorithms rely on it. We called it the *Pre-Defined Dependency Pattern* (PDD) as the cell computation follows an order based on dependencies between cells. It leads to a more complex parallel schema. This pattern can be found in the flow accumulation algorithm or in the watershed algorithm for example.

We need some preliminary notations and definitions to describe our patterns. In GIS, the initial data is a terrain considered as a regular mesh. Several values can be assigned to each point of this terrain (elevation, direction, ...). In this paper, we consider that all values are stored in different meshes that have the same structure. Thus, we call $\mathcal{D}^{\mathcal{T}}$ the domain associated to a terrain (i.e. the mesh). A *cell* c is a point of $\mathcal{D}^{\mathcal{T}}$ and for a mesh m, $m(c)$ denotes the value

attributed to c. Classically, c is a pair (i, j) that defines a point of the terrain $\mathcal{D}^{\mathcal{T}}$.

As, we focus on distributed architecture, the different meshes are distributed on the different processes. Each process manages a sub domain $\mathcal{D} \subseteq \mathcal{D}^{\mathcal{T}}$. To optimize parallel computation, each mesh is divided with overlapping parts. Each process owns additional cells called *ghost cells* and denoted by $\mathcal{G}$ illustrated in the figure 2 with the cells in grey. Finally, we denote $\mathcal{O} \subset \mathcal{D}$ the cells which are shared with another process.



(a) Local data with a ghost line in gray (b) Computed Directions

Figure 2: Ghost cells on the flow direction example

```
1  Function STENCIL(m_in:mesh, N_size:int,
     f_update:func, f_border:func, nb_iter:int):mesh is
2    mesh m_out;
3    for i=1 to nb_iter do
4      Exchange_Ghosts(m_in);
5      foreach Cell c at the borders of D\G do
6        m_out(c) = f_border(c, m_in);
7      foreach Cell c ∈ D\G not at the border do
8        m_out(c) = f_update(c, m_in);
9      m_in = m_out;
10   return m_out;
```

Algorithm 1: Stencil Pattern

3.1 Stencil Pattern

The stencil pattern consists in computing for each point of a matrix, a value depending on the values of its neighbors. Thus, a kernel convolution matrix or a mask is applied on the matrix. The mask size indicates the size of the neighborhood needed to update a point.

The stencil computation is defined by means of the neighborhood size N_{size} of a cell, two functions f_{update} to be applied on each cell of the mesh and $f_{boundary}$ to be applied on the cells at the border of the mesh (i.e. those whose neighborhood is not complete). The two functions are implemented assuming that all the data are local to the process. The last parameter nb_{iter} is the number of time the stencil must be applied. Algorithm 1 shows how these parameters are used in the pattern.

The stencil pattern can be used to implement the flow direction algorithm. In this case, the size of the stencil is 1 (i.e. the neighbors are the 8 cells surrounding a cell c), $f_{update}(c)$ finds the lowest cell among the 8 that surround c. The f_{border} function is a special case of f_{update} when the neighborhood contains less than 8 cells. In input m_{in} is a Digital Elevation Model (DEM) and as result m_{out} is a mesh of directions such that $m_{out}(c)$ indicates where c flows.

Concrete parallel implementation. Stencil computation only needs a bounded neighborhood to compute the value of a cell. Its parallelization is quite simple and efficient. Indeed, it is sufficient to divide the data set into overlapping parts such that each processor gets all the data it needs to perform its computations on $\mathcal{D}$ independently as illustrated Fig. 2. The pattern computes the result mesh only on its domain $\mathcal{D}$. Thanks to the ghost cells $\mathcal{G}$ it is able to access to the complete neighborhood of the $\mathcal{D}$ boundary cells.

The concrete stencil model requires the data including ghosts to be already distributed. Then the constructed program from the user's functions and neighborhood description is applied on the $\mathcal{D}$ data as described Algorithm 1. The parallelization consists in exchanging the ghosts when the iterative version is necessary. The line 4 is automatically transformed into a communication phase to send to neighbor processes m_{out} of the cells in $\mathcal{O}$ and to receive in $\mathcal{G}$ the

m_{out} of the $\mathcal{O}$ cells of these neighbor processes. This phase depends on the initial distribution and on the neighborhood size to update each cell. We implemented the three classical strategies to distribute a mesh, per bands of lines, per bands of columns and per blocks. In addition, to take into account the crucial load balancing when data can be partially sparse because of no-data, we implemented also a round-robin distribution to assign bands or blocks in a circular order to the processes.

3.2 The PDD Pattern

The Pre-Defined Dependency Pattern addresses a class of algorithms were the value of a cell depends on the input data and the value of several other cells. For example, the computation of the flow accumulation of a cell c in a terrain, requires the knowledge of the direction flow and the value of the accumulation flow of each cell which flows to c. The computation consists in evaluating the cells without dependencies first and then iteratively in evaluating the cells which depend only on already computed cells. Such computations are very common in geosciences but their parallelization is more complicated than stencil computations since some cells may need the value of other cells located on different processes which induces communications. The PDD pattern relies on a pre-defined dependency relation between cells that is non-cyclic. Moreover, for each cell, the cells it depends on, must be included in a bounded neighborhood. Thanks to these two properties, we are able to provide an efficient implicit parallelism pattern for these kind of algorithms.

Let m_{in} denote the input mesh and m_{out} denote the result mesh. The dependencies is given by a boolean function $r(c, c', m_{in})$ which tells if the cell c is dependent of c'. From r it is possible to compute a dependency coefficient n_d to each cell c of $\mathcal{D}^{\mathcal{T}}$ which is the number of c neighboring cells which have to be computed before c. The neighborhood definition is the same as the stencil pattern one and is defined from the N_{size} parameter. The algorithm consists then in computing the value of the cells whose $n_d = 0$ and in decreasing n_d of the cells c' such that $r(c, c', m_{in}) = true$. The algorithm stops when all the n_d coefficients are zero. Therefore, the user needs only to define two functions. The boolean function r defining the dependency between two cells. The updating function $f_{update}(c, m_{in}, m_{out})$ that computes $out(c)$ from the cells on which c depends and returns $S_d(c)$ the set of cells which depend on c. Then the computations are as follows where $N(c)$ designed the c neighborhood

1. $\forall c \in \mathcal{D}\ n_d(c) = |\{c' \in N(c) | r(c, c', m_{in})\}|$
2. $\forall c\ s.t.\ n_d(c) = 0$
   - $S_d(c) = f_{update}(c, m_{in}, m_{out})$
   - $\forall c' \in S_d(c)\ n_d(c') - = 1$

Finally, from f_{update} and r, the abstract model illustrated Algorithm 2 becomes a concrete program and it can be transformed into a parallel implementation. In this algorithm the function $f_{dep}(c, m_{in}, r)$ computes the n_d of c. This function can be automatically derived from r.

This pattern can be used to implement the flow accumulation algorithm supposing that m_{in} the flow directions are already computed. In this case, the neighborhood size is 1 and $N(c)$ of a cell c consists of the 8 surrounding cells. The two functions are defined as follows

- $r(c, c', m_{in}) = \left| \begin{array}{ll} true & \text{if } m_{in}(c') \text{ indicates } c \\ false & \text{otherwise} \end{array} \right.$

- $f_{update}(c, m_{in}, m_{out}) : \left| \begin{array}{l} m_{out}(c) = \sum_{\{c' \in N_c | r(c,c',m_{in})\}} m_{out}(c') \\ S_d(c) = \{c'' | r(c'', c, m_{in})\} \end{array} \right.$

```
1  Function PDD(m_in: mesh, f_dep:func, f_update:func) : mesh is
2      mesh m_out; // The output value
3      mesh n_d; // The dependency for each cell
4      queue O ← ∅;
5      foreach Cell c ∈ G do
6          n_d(c) = 0;
7      foreach Cell c ∈ D do
8          n_d(c) = f_dep(c, m_in, r);
9          if n_d(c) == 0 then
10             O ← O ∪ c;
11     bool over = false;
12     while (not over) do
13         while O ≠ ∅ do
14             c ← pop(O);
15             S_c = f_update(c, m_in, m_out);
16             foreach cell c' ∈ S_c do
17                 n_d(c') − = 1;
18                 if n_d(c') == 0 and c' ∈ D then
19                     O ← O ∪ c;
20         Exchange_Ghosts(m_out);
21         Send({n_d(c)|c ∈ G});
22         Receive({n_d^up(c)|c ∈ O});
23         foreach cell c ∈ O do
24             n_d(c)+ = n^up(c);
25             if n_d(c) == 0 then
26                 O ← O ∪ c;
27     return m_out;
```

Algorithm 2: Pre-Determined Dependencies Pattern

Concrete parallel implementation. As for the stencil pattern, the mesh is distributed either per bands or per blocks with the round-robin order. The initial distribution is supposed to be done with ghost data so that f_{dep} and f_{update} can be applied on each local cell without communications. At the iteration end, when all the possible cells are computed, the communications are carried out. The line 20 is similar to the ghost exchange in the stencil pattern to update m_{out} neighborhood of the border cells. The lines from 21 to 26 describe the n_d updating. Indeed, when a cell of $\mathcal{O}$ is computed, it can change the n_d value of $\mathcal{G}$ cells. This modification needs to be sent to the owner processor so that it can compute the new value n_d of the $\mathcal{O}$ cells (line 26). Therefore the concrete implementation consists in replacing the lines 20 to 22 with the code to express the corresponding communications. This automatic part depends only on the N_{size} parameter and of the initial distribution.

PDD pattern with memorized order. If the pattern is used repetitively, as the dependency does not change, the pattern can be considerably improved. Indeed, in the first pattern application, it is possible to memorized the order in which the cells were updated. It is sufficient to replace the lines 10, 19 and 26 with a way to memories which cells has been added. In

this way the new pattern is reduced to the cell updates and the ghost exchanges of m_{out}. All calculations for n_d dependency coefficients are no longer necessary.

4 Results

We used the C++ templates to automatically transform our patterns into a C++ implementation using the MPI library. For example, we designed a datastructure called *mesh*< *Type,* N_{size}*, n, m*> where the parameters respectively characterize the mesh data type, the neighborhood and the block dimensions. This structure allows to deal with the different data distribution and communication phases. The DSL will provide an easy way to manipulate the mesh without the need of all the parameters of the C++ implementation. Thus, the user will only provide the dataset associated with the mesh and we will automatically find the parameters as the distribution method, the size of the blocks...

To test the performance of the patterns, we compared them to two handmade implementations in C++ using the MPI library. The flow direction computation has been used to illustrate the stencil pattern and the flow accumulation computation for the PDD pattern.

We used the compiler GCC 6.1.0 with optimization flag set to -O2 and OpenMPI 1.10.2. All tests have been done on the Centre de Calcul Scientifique en Région Centre (CCSC). The cluster runs on the Scientific Linux Release V6.6. It is composed of 48 nodes where each node hosts 20 CPU's Intel Xeon E5-2670 2.5Ghz, 64Gb of memory. The network is an InfiniBand 40G/s. We used 4 nodes for the tests with 64 cores. We done our tests on a 60001×90001 matrix of 32-bit integer (around 20GB). The matrix represents Asia and contains a lot of no data values at the bottom that may cause load unbalancing. Every run has been done height times and averaged. The maximum deviation was less than 2%.

We chose to split the meshes into bands of lines. Because of ghost exchanges, a distribution per blocks will generate much more communications. Two splitting have been done, one with 50 lines per bands and an other with 100 lines. With our configuration, the best splitting proved to be the splitting with 100 lines per bands. Thus, we present only these results. Let notice that beyond 64 processes, the band height should be decreased in order to ensure that each process deals with at least one band of lines of our dataset.

The results obtain for the stencil pattern are shown in the figure 3. The performance is very similar to the handmade version. Thus, the pattern genericity does not generate additional costs. As we mainly focus on the PDD pattern, we do not dedicate much space for the stencil performance.

The figure 3 presents the results of the PDD pattern denoted by PDD against the handmade version denoted by H in log scale to highlight the speed-up. The algorithm tested is the flow accumulation algorithm. The overhead of PDD is about 8% compared to the H except for 32 cores when the overhead is about 17%. The pattern needs to be general, so some conditions have to be done in the patterns which can be excluded in the H version. However, the two versions are still very closed. Moreover, we can see that the speed-up is linear which was not a foregone conclusion since the computation is less regular than a stencil one. The figure 4 presents the results of the PDD patterns against the pre-computed order version. This last is decomposed into two parts, the PDD_o part, which is the order computation algorithm and PDD_{k-o} part which is the PDD pattern when the cell order is known. We can see that the PDD_o part is slower than the PDD version, while it only computes the order. This is due to data structure (a map of vector) used to store the order of the cells, the synchronizations and the frequent access to it. However, the PDD_{k-o} part, is really faster compared to the PDD version by almost 68%. Hence, the PDD version is the most useful version when the user only needs to

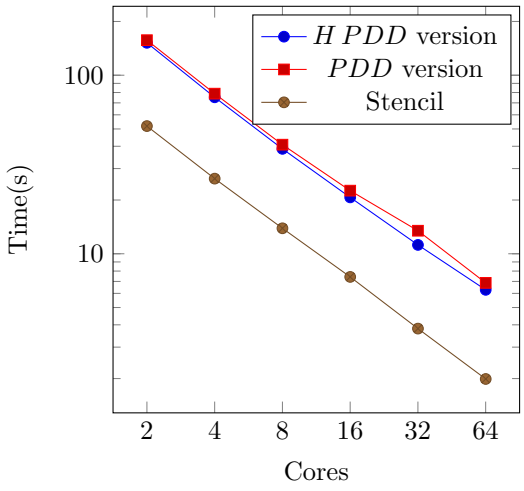


Figure 3: Time comparison between hand-made algorithm and PDD algorithm

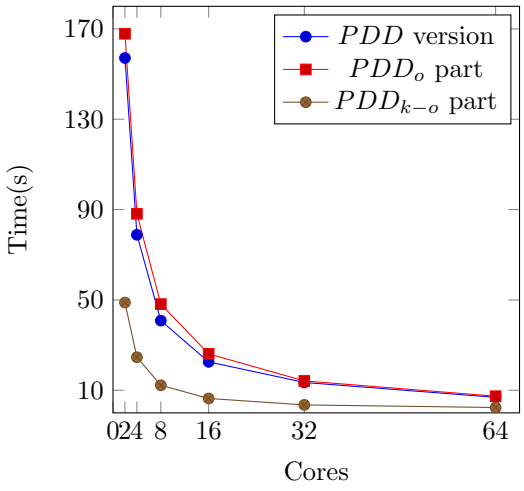


Figure 4: Time comparison between PDD algorithm and ordered PDD algorithm

run the pattern once, but, if he/she wants to re-run the pattern with various parameters or on slightly different data but with the same dependency between cells the PDD_{k-o} proved to be really interesting because the cells order is computed only once.

5 Conclusion and Perspectives

In this paper, we presented two patterns as elements of a general implicit parallel framework for GIS. If the stencil pattern is well-known, the PDD pattern is more innovative. We showed its efficiency particularly in a repetitive process when the computation order has been memorized during the first iteration. The perspectives of this work are numerous. Beyond new optimization for our patterns, new versions are under development to take into account different machine architectures (shared memory, GPGPU). Other patterns need to be added to address more problems as the wavefront pattern for example. Moreover, pattern composition need to be defined to express computations as a workflow. For example, in the PDD pattern, the n_d coefficient computation can be seen as a stencil computation. However, the cooperation between the two patterns induce an additional cost because of extra data access. More generally, the data distribution need to be specifically studied in order to help the user to choose the adapted policy. Finally, these patterns are the first step for the creation of a layer based framework to help GIS specialists to create parallel program easily. We want to develop a DSL based on Python to completely hide the parallelism and the C++ templates to users. This DSL associated to our patterns should offer a simple way for geoscientists to parallelize their data treatments.

References

[1] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *GeoInformatica*, 7(4):283–313, 2003.

[2] P. Baumann. Rasdaman: Array databases boost spatio-temporal analytics. In *Computing for Geospatial Research and Application (COM. Geo), 2014 Fifth International Conference on*, pages 54–54. IEEE, 2014.

[3] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[4] H. Coullon and S. Limet. The sipsim implicit parallelism model and the skelgis library. *Concurrency and Computation: Practice and Experience*, 28(7):2120–2144, 2016.

[5] M. Danelutto and M. Torquati. Structured parallel programming with core fastflow. In *Central European functional programming school*, pages 29–75. Springer, 2015.

[6] O. David, W. Lloyd, J. C. Ascough II, T. R. Green, K. Olson, G. Leavesley, and J.R. Carlson. *Domain specific languages for modeling and simulation: use case OMS3*. PhD thesis, International Environmental Modelling and Software Society (iEMSs), 2012.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] P. Degenne, D. L. Seen, D. Parigot, R. Forax, A. Tran, A. A. Lahcen, O. Curé, and R. Jeansoulin. Design of a domain specific language for modelling processes in landscapes. *Ecological Modelling*, 220(24):3527–3535, 2009.

[9] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[10] Q. Guan and K. C. Clarke. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. *International Journal of Geographical Information Science*, 24(5):695–722, 2010.

[11] N. Holst and G. F. Belete. Domain-specific languages for ecological modelling. *Ecological Informatics*, 27:26–38, 2015.

[12] F. Huang, D. Liu, X. Tan, J. Wang, Y. Chen, and B. He. Explorations of the implementation of a parallel idw interpolation algorithm in a linux cluster-based parallel gis. *Computers & Geosciences*, 37(4):426–434, 2011.

[13] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[14] C. Qin, L. Zhan, A. Zhu, and C. Zhou. A strategy for raster-based geocomputation under different parallel computing platforms. *International Journal of Geographical Information Science*, 28(11):2127–2144, 2014.

[15] F. Rabhi and S. Gorlatch. *Patterns and skeletons for parallel and distributed computing*. Springer Science & Business Media, 2003.

[16] D. Tarboton. Terrain analysis using digital elevation models (taudem). *Utah State University, Logan*, 2005.

[17] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.