Implementation and Performance Analysis of SkelGIS for Network Mesh-Based Simulations

Hélène Coullon[1,2] and Sébastien Limet[1]

[1] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, F-45067 Orléans Cedex 2
{helene.coullon,sebastien.limet}@univ-orleans.fr
[2] Géo-Hyd Antea Group, 101 rue Jacques Charles, 45160 Olivet, France

Abstract. The implicit parallelism is an active domain of computer-science to hide intricate details of parallelization from the end-user. Some solutions are specific to a precise domain while others are more generic, however, the purpose is always to find the adapted level of abstraction to ease the high performance and parallel programming. We present SkelGIS, a *header-only* implicit parallelism C++ library to solve mesh-based scientific simulations. In this paper is detailed the implementation of SkelGIS for the specific case of *network simulations*, where the space domain can be represented as a *directed acyclic graph* (DAG). This implementation is based on a modified, optimized and parallelized version of the *Compressed Sparse Row* format, which is completely described in this paper. Finally, experiments on different kinds of clusters and different sizes of DAGs are evaluated.

1 Introduction

Taking advantage of the full potential of emerging parallel high performance systems becomes increasingly difficult. In novel processor architectures such as GPUs and many-cores processors, the memory hierarchy becomes complex, and CPU clusters are growing toward a capability of millions and billions of cores. Even if those extremes are not common nowadays, it becomes difficult and time consuming to write efficient parallel codes, especially for non-computer scientists. This is why *implicit parallelism* is one of the most active domain of computer research. It consists in providing programming tools to improve the level of productivity and performance of parallel codes, while hiding the intricate details of new architectures and low level parallel libraries.

Most scientific simulations are based on partial differential equations (PDEs). Some of those equations can be solved analytically, some of them cannot. This technique is difficult and most resolutions are proceeded for specific boundary or initial conditions. Thus, most PDEs are commonly solved using numerical methods such as finite difference, finite volume or finite element methods. Those methods produce mesh-based simulations where the time domain is discretized by a set of iterations and the space domain is discretized by a set of points (finite difference) or a set of cells to form a mesh (finite volume and finite element). One or more numerical schemes are obtained from numerical methods. A scheme

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 439–450, 2014.

represents a computation to apply at each time iteration on the mesh and is of the form:

$$\{U_{t-1}(x), U_{t-1}(y); y \in N(x)\} \longmapsto U_t(x), \tag{1}$$

where x represents an element of the mesh, $U_t(x)$ is the set of quantities to compute for element x at the time iteration t, $N(x)$ is the neighborhood of x required to compute $U_t(x)$. In other words, quantities of the element x at a time iteration t is a function of quantities for this element and its neighborhood at time iteration $t-1$. In computer science, this kind of computation is called a *stencil* and is intensively studied.

Some simulations have the particularity to identify two kinds of places in the space domain, for which the behavior is different and where different numerical schemes are applied to simulate the real phenomena. For example, in a blood flow simulation in the human arterial network, the behavior is different in an artery and at a conjunction node where arteries join. In this case, the domain is first discretized as a *network*, before each kind of element is again discretized to a mesh. A network is represented as a graph and contains two kinds of elements: nodes and edges. For the same blood flow example, the arteries are represented by the edges of the graph and the conjunction nodes by the nodes of the graph. A node could have more or less incoming and outgoing edges (arteries), thus a network is an irregular structure. Networks are used a lot in different kind of simulations as, for example, arterial or vein simulations, road or rail traffic simulations, water-flow or pollutant transfer simulations etc. Thus, it is very important to offer implicit parallelism solutions to write parallel network-simulations. However, existing implicit parallelism solutions, to solve mesh-based PDEs, do not propose an easy way to write network-simulations. SkelGIS is a header-only C++ library to write parallel mesh-based simulations on distributed memory architectures, using MPI. The parallelization of codes is totally hidden from the user through four concepts: *distributed data structures* (DDS), *data mappings*, *appliers* and *interfaces*. Those concepts have already been implemented for Cartesian two-dimensional regular mesh [5,6]. In this paper is presented the implementation of the implicit parallelism library SkelGIS, for the specific case of network simulations, and more precisely for networks which can be represented by directed acyclic graphs (DAGs). The implementation of SkelGIS for network simulations, and its efficiency, are based on an adaptation, a re-indexation and a parallelization of the *compressed sparse row* (CSR) format, which is described in this paper. Moreover, performances of the solution have been evaluated until 8000 cores on a blood flow simulation.

The rest of this paper is organized as follows. Section 2 explains the CSR format for sparse matrices and graphs. Then, Section 3 describes the adaptation of the CSR format for networks. Section 4 details the implementation of the whole SkelGIS solution for network simulations. Performance of SkelGIS is evaluated in Section 5 and related work are discussed in Section 6. Finally, Section 7 concludes this work.

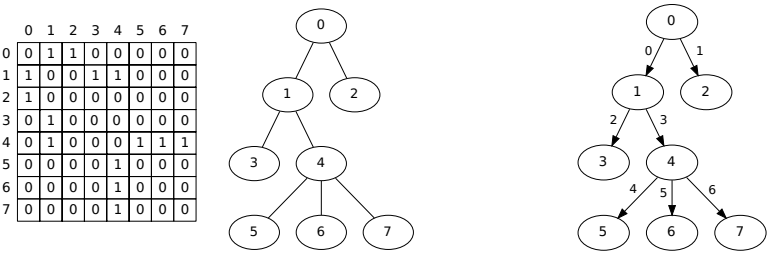
2 The Compressed Sparse Row Format

The work presented in this paper is based on an adaptation of the Compressed Sparse Row (CSR) format [3] for network simulations. Two different views of the CSR format are presented in this section, first the sparse matrix view and then the graph view.

The 3-array variation of the CSR format handles the storage of sparse matrices with three arrays. The first array, named *values*, contains the non-zero values of the matrix, stored line by line. The second array is named *columns*. The element i of the array *columns* contains the column index of the associated i^{th} element of *values*. Finally, the third array is named *rowIndex*. The element i of the array *rowIndex* contains the index, in the array *values*, of the first non-zero element of the i^{th} row of the matrix. A dummy entry, equal to zero, is added at the beginning of the *rowIndex* array. This way, the row i contains $rowIndex[i+1] - rowIndex[i]$ non-zero elements. To access a non-zero element with its row and column indexes (i, j) it is needed to find j between elements $columns[rowIndex[i]]$ and $columns[rowIndex[i+1]-1]$. The index where j is found in *columns* is the index of the searched non-zero value in *values*. CSR, as other formats, only stores non-zero values of a sparse matrix. Thus, the CSR has a light memory footprint. However, it suffers from a lack of efficiency to access a non-zero value with its row and column index (i, j). Nevertheless, CSR is very efficient to represent connectivity in a graph as shown below.

CSR can be used to store undirected graphs. An *undirected graph* is denoted by $G = (V, E)$ where V is a finite set of *vertices* or *nodes* and $E \subseteq V \times V$ is the set of *edges*. The matrix $Sp(G)$ associated to a graph G represents the adjacency matrix of the graph G (e.g. Figure 1(a)). In the case of undirected graphs, $Sp(G)$ is symmetric. In a graph $G = (V, E)$, v_i and $v_j \in V$ are said *neighbor vertices* if $(v_i, v_j) \in E$. In other words, two vertices are neighbor vertices if a non-zero value is placed at (v_i, v_j) and (v_j, v_i) in $Sp(G)$. $\forall v \in V, N(v)$ denotes the set of neighbors of the vertex v. The degree of a vertex $v \in V$, denoted by $deg(v)$, is the number of incident edges to v, i.e. $deg(v) = |N(v)|$. In the row v of the matrix $Sp(G)$, $N(v)$ represents column indexes where a non-zero value is present. In an undirected graph $G = (V, E)$ where $V = \{v_0, ..., v_{n-1}\}$, the *cumulative degree* of a vertex $v_i \in V$ is denoted $cdeg(v_i)$ and defined by $cdeg(v_i) = \sum_{j=0}^{i} deg(v_j) = \sum_{j=0}^{i} |N(v_j)|$. In the matrix $Sp(G)$, $cdeg(v_i)$ represents the number of non-zero elements in the row v_i added to the number of non-zero elements in previous rows. Thus, it is possible to represent G with two arrays. The first one of size $n+1 = |V|+1$, called *cdeg*, is defined by $cdeg[i+1] = cdeg(v_i), \forall i \in [0, n[$, where $cdeg[0] = cdeg(v_{-1}) \stackrel{def}{=} 0$. The second array, denoted N (for neighborhood), is of size $cdeg[n] = cdeg(v_{n-1})$ and $\forall v_i \in V, N(v_i) = \{v_{N[j]} | j \in [cdeg[i], cdeg[i+1][\}$. This two-arrays representation corresponds to the CSR format where arrays *cdeg* and N of G are respectively equal to the arrays *rowIndex* and *columns* of $Sp(G)$. Figure 1(a) illustrates a simple undirected graph. The node 0 of this graph has two neighbors, 1 and 2, as a result, the second cumulative degree value is 2. Node 1 has three neighbors therefore $cdeg[2] = 2 + 3 = 5$. Iterating this process on

each node of the graph leads to $cdeg = [0, 2, 5, 6, 7, 11, 12, 13, 14]$ for this graph. Neighbors of node 0 are 1 and 2 and those of node 1 are 0, 3 and 4 etc. Finally, $N = [1, 2, 0, 3, 4, 0, 1, 1, 5, 6, 7, 4, 4, 4]$. It can be noticed that the neighborhood of node 4, for example, can be easily accessed since $cdeg[4] = 7$ and $cdeg[5] - 1 = 10$ give the first and the last index of N where are stored indexes of the neighbors of node 4, as a result nodes 1, 5, 6 and 7 are the neighbors of node 4.



(a) An undirected graph G and its associated sparse adjacency matrix Sp(G).

(b) The equivalent directed graph G'.

Fig. 1. Examples of graphs

Unlike in the sparse matrix case, where elements are accessed by $(row, column)$ indexes, in the graph case elements are accessed by row index only and then the neighborhood of the node can be accessed in $O(1)$. Supposing that the data associated to each nodes are stored in a table X such that $|X| = |V|$, then accessing neighbor values of a node v_i simply consist in accessing X from index $N[cdeg[i]]$ to $N[cdeg[i + 1] - 1]$.

3 A Distributed Data Structure for DAGs

This paper deals with the implementation of SkelGIS for the specific case of networks which can be represented with directed acyclic graphs (DAGs). This section shows how CSR can be used to implement DAGs, how it can be optimized to fit scientific simulation needs, and how it can be efficiently parallelized. A distributed version of the CSR format has already been proposed by Edmonds et Al [8]. However this distributed version is not specifically improved and optimized for scientific mesh-based simulations.

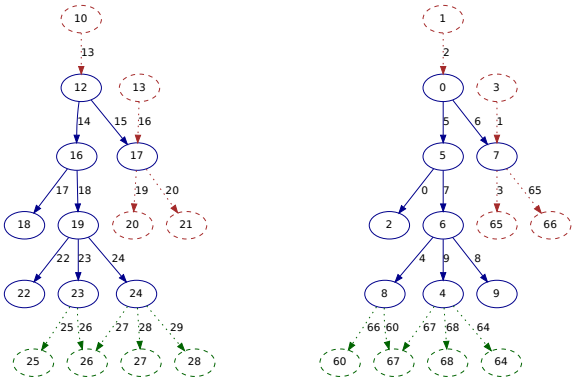
A *directed graph* $G = (V, E)$ is a graph where each edge $e = (v_1, v_2) \in E$ is directed from v_1 to v_2 and where v_1 and v_2 are respectively called the *source* and *destination* nodes of e. A *directed acyclic graph* (DAG) is a directed graph $G = (V, E)$ such that for all node $v \in V$, there is no path, following successive directed edges, from v to itself. In scientific simulations on DAGs, a node could need incoming edges and nodes, and outgoing edges and nodes, to be computed. An edge, on the other hand, needs its source and destination nodes to

be computed. First, in a DAG $G = (V, E)$, for a node $v \in V$ and an edge $e \in E$, $S(e)$ denotes the source node of e and $D(e)$ denotes the destination node of e. Two arrays represent this information, S and D of size m where $S[i] = S(e_i)$ and $D[i] = D(e_i)$ for all $e_i \in E$. $N_V^+(v)$ denotes the set of output nodes of v such that $N_V^+(v) = \{v' | (v, v') \in E\}$. $N_E^+(v)$ denotes the set of output edges of v, thus $N_E^+(v) = \{e \in E | S(e) = v\}$. Symmetrically $N_V^-(v)$ and $N_E^-(v)$ denotes the sets of input nodes and edges of a vertex v. Finally, a *root* node v of a DAG G verifies $|N_E^-(v)| = 0$, and a *leaf* node verifies $|N_E^+(v)| = 0$. Cumulative degrees are the same for edges and nodes because the number of incoming edges and nodes are the same, and the number of outgoing edges and nodes are equal too. Then, $\forall v_j \in V$, $|N_E^+(v_j)| = |N_V^+(v_j)|$ and $|N_E^-(v_j)| = |N_V^-(v_j)|$. In a DAG $G = (V, E)$, where $V = \{v_0, ..., v_{n-1}\}$), for a vertex $v_i \in V$: $cdeg^+(v_i) = \sum_{j=0}^{i} |N_E^+(v_j)|$ denotes the output cumulative degrees of a node v_i and $cdeg^-(v_i) = \sum_{j=0}^{i} |N_E^-(v_j)|$ denotes the input cumulative degrees of a node v_i. $cdeg^+$ and $cdeg^-$ denote arrays of size $n + 1$ such that $cdeg^+[i + 1] = cdeg^+(v_i)$ and $cdeg^-[i + 1] = cdeg^-(v_i)$, where $cdeg^+[0] = cdeg^-[0] = 0$. Finally, as in Section 2, it is possible to define arrays N_V^+, N_E^+, N_V^-, and N_E^- of size $cdeg^+[n]$ and $cdeg^-[n]$.

Figure 1(b) represents a directed graph which has the same structure as the graph of Figure 1(a). Node 0 has no input neighbor but has two output neighbors, as a result the second value of $cdeg^+$ is 2 and the second value of $cdeg^-$ is 0 etc. Associated neighbor nodes are stored in N_V^+ and N_V^-, the corresponding edges are stored in N_E^+ and N_E^-. The whole representation of this DAG is given by the following eight arrays : $cdeg^+ = [0, 2, 4, 4, 4, 7, 7, 7, 7]$, $cdeg^- = [0, 0, 1, 2, 3, 4, 5, 6, 7]$, $N_E^+ = [0, 1, 2, 3, 4, 5, 6]$, $N_E^- = [0, 1, 2, 3, 4, 5, 6]$, $N_V^+ = [1, 2, 3, 4, 5, 6, 7]$, $N_V^- = [0, 0, 1, 1, 4, 4, 4]$, $S = [0, 0, 1, 1, 4, 4, 4]$ and $D = [1, 2, 3, 4, 5, 6, 7]$.

Some modifications of the DAG data structure are needed to parallelize it on distributed memory architectures. Indeed, in this case, the DAG has to be partitioned into sub-graphs (of equivalent sizes) which are distributed among processors. Here is not discussed the way the graph is partitioned but only how to optimize local representation of each sub-graph to keep benefits of the data structure. Each processor receives a part of the graph where indexes are global, which means that the initial local indexing may be non-continuous. Therefore a re-indexing is needed to represent the local sub-graph. Figure 2(a) illustrates the extraction of the sub-graph managed by the processor 1 from a DAG $G = (V, G)$. This sub-graph $G_1 = (V_1, E_1)$ has eight nodes, $|V_1| = 8$, and seven edges, $|E_1| = 7$. Connections with the rest of the DAG G are drawn with dashed lines. These dashed elements are denoted as *halo* elements and represent needed information from other processors to compute the stencil.

The re-indexation of local edges is very simple, it goes from up to bottom and from left to right at each level (Figure 2(b)). The re-indexation of local nodes, on the other hand, sorts nodes in several classes to optimize the use of cache lines and to minimize the number of conditions in the SkelGIS code. First, roots and leaves are distinguished from other nodes. Actually, in most simulations, roots and leaves are computed differently to manage the physical border of the domain. Grouping those nodes together in memory allows a better use of cache lines.



(a) Sub-graph managed by proc. 1

(b) Re-indexing of the sub-graph.

Fig. 2. Sub-graph managed by proc. 1 (blue) and its connections with those of proc. 0 (red) and 2 (green).

Moreover, it makes possible to move through those elements avoiding conditions in the code (to test if a node is a root or a leaf). Secondly, remaining local nodes are partitioned into two sets: those needing communications to get halo elements and the others. This makes possible an efficient overlap of computations with communications [9], using non-blocking routines of MPI. As for roots and leaves, those two classes improve cache use and avoid conditions in the code. Thus, performances of final programs will be improved by the re-indexation.

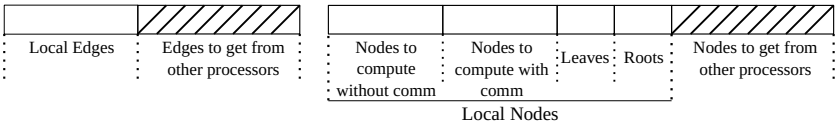


Fig. 3. Re-indexation for edges and nodes

Figure 3 illustrates the different classes used for re-indexation and an example is shown Figure 2. The eight arrays of the local data structure of the processor 1 are: $cdeg^+ = [0, 2, 5, 7, 7, 7, 7, 7, 7]$, $cdeg^- = [0, 1, 2, 2, 3, 4, 5, 6, 7]$, $N_E^+ = [2, 3, 4, 5, 6, 0, 1]$, $N_E^- = [0, 3, 1, 5, 6, 4, 2]$, $N_V^+ = [7, 1, 6, 4, 5, 0, 3]$, $N_V^- = [2, 0, 2, 1, 1, 1, 0]$, $S = [2, 2, 0, 0, 1, 1, 1]$ and $D = [0, 3, 7, 1, 6, 4, 5]$. This data structure must be completed with information on connections with halo elements of the rest of the DAG. To manage these incoming and outgoing information, the halo elements are added to the local DAG structure with indexes starting from the greater local index of local nodes and edges. This way, the cache line optimization obtained by the re-indexation is kept. To insert external nodes in the

local data structure, cumulative degrees arrays $cdeg^+$ and $cdeg^-$ and associated tables N_E^+, N_E^-, N_V^+, N_V^- must be updated. In Figure 2(b), one can note that local nodes 2 and 3 receive one input edge and node from processor 0. Thus, $cdeg^-$ is modified at indexes 3 and 4, adding one to each. As $cdeg^-$ represents cumulative degrees, nodes 3 to $|V_1|$ and 4 to $|V_1|$ have to be updated. Modifications on N_V^- consist in adding new indexes to store halo elements to receive from other processors. Thus, indexes 8 and 9 are inserted at the right place. The parallel data structure for processor 1 is then $cdeg^+ = [0, 2, 5, 7, 9, 11, 16, 16, 16]$, $cdeg^- = [0, 1, 2, 3, 5, 6, 7, 8, 9]$, $N_E^+ = [0, 2, 3, 4, 5, 6, 0, 1, 9, 10, 11, 12, 13, 14, 15]$, $N_E^- = [0, 3, 7, 1, 8, 5, 6, 4, 2]$, $N_V^+ = [7, 1, 6, 4, 5, 0, 3, 10, 11, 12, 13, 13, 14, 15]$ and $N_V^- = [2, 0, 8, 2, 9, 1, 1, 1, 0]$. As a consequence, neighborhood information can still be obtained in $O(1)$, even if information comes from other processors, and the parsing of local elements is cache-optimized. However, it is not possible with resulting arrays to determine communication scheme to proceed MPI exchanges. For this reason six additional tables are added in the parallel data structure to manage interprocess communications. These arrays rely on the cumulative degrees applied to processor ranks. Tables $cdeg^{tor}$ and $cdeg^{tos}$ are the cumulative degrees of nodes and edges to send to and to receive from other processors, where $|cdeg^{tor}| = |cdeg^{tos}| = p + 1$. And arrays N_E^{tor}, N_E^{tos}, N_V^{tor} and N_V^{tos} are the associated sets of nodes and edges indexes to send and receive.

The data structure presented in this section is independent from the graph partitioning method used to distribute the DAG. It is optimized to efficiently parse each sub-graphs and to allow communications/computations overlaps. However, the key point to obtain good performances of irregular structures lies in a good partition of the structure. This problem is known to be NP-complete, as a consequence, heuristics are used to approximate the solution. The network partitioning problem is not presented in this paper, a sibling-edges heuristic has been used for experiments and produces sensible results. Two other solutions, using the hypergraph partitioning model, with Mondriaan [13], are under study. Finally, ParMetis [12] and PTScotch [4] are known to obtain good partitioning for unstructured meshes, and could probably be used for a network partitioning.

4 SkelGIS Implementation for Network Simulations

SkelGIS is an implicit parallelism library for distributed memory architectures, which proposes a sequential programming view to the user while producing SPMD parallel programs. Figure 4 illustrates the user- and the real-view of a SkelGIS program. SkelGIS hides parallelization of codes through four concepts which are illustrated in Figure 4: *DDS* which represents a distributed data structure. The DDS is responsible for storing the domain and its connectivity and partitioning it automatically. In SkelGIS multiple DDSs are available to represent different kinds of domains and meshes. *DPMap* which maps data on the DDS. Each instantiation of such an object represents data used in the simulation and its mapping on the *DDS*. *AP* which is an applier. It is used to apply a sequential user code, called an *operation, OP*, to a set of DPMaps. An applier also

transparently proceeds MPI communications between processors. I represents the programming interface of SkelGIS. I is used to navigate through DPMaps, read and update them. This interface is based on *iterators* and specific functions to access the *neighborhood* of elements (stencil). Applied to network simulations,

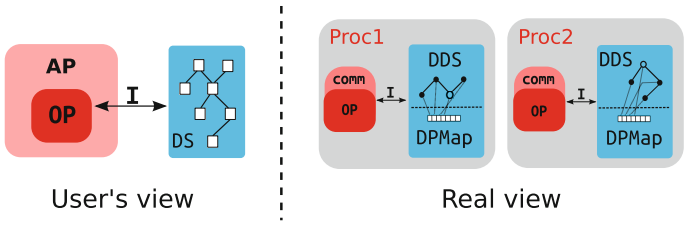


Fig. 4. SkelGIS user's view and its actual parallel execution

the DDS concept of SkelGIS is implemented using the distributed data structure presented in the previous section. Each of the three remaining concepts of SkelGIS, applied to networks, are implemented using this DDS, which makes possible and efficient the whole solution. Two DPMaps are needed for network simulations, one to map data on nodes and the other one to map data on edges. A DPMap can be compared to the array *values* of the CSR format. Using the DDS presented in the previous section, a DPMap is a light object which maps a distributed one-dimensional array to the local re-indexation of nodes and edges on each processor. This one dimensional array stores data associated to the DDS, and is able to store, for each edge or node, another one-dimensional array to implement complicated schemes. In this case, the order of the scheme gives information on needed communications. To be efficient with those different cases, partial template specializations [1] are used in SkelGIS. An applier is responsible for hiding communications from the user. As a result, *appliers* for networks use communication arrays $cdeg^{tor}$, $cdeg^{tos}$, N_E^{tor}, N_E^{tos}, N_V^{tor} and N_V^{tos} of the DDS. Three kinds of *iterators* are available to move through DPMaps applied to a network in SkelGIS. The first kind moves through all local nodes or edges of the *DDS*. The second kind moves through local nodes and edges which do not need communications to be computed, or on the contrary through local nodes and edges which need communications to be computed. This kind of iterators makes possible an overlap of computations with communications. Finally, the last kind of iterators moves through roots and leaves of the DAG, to manage the physical border of the domain. All moves of iterators are unordered as there are no dependencies in the numerical scheme (1), but all elements are guaranteed to be parsed by the iterator. All *iterators* use the re-indexation described in the previous section, to move contiguously in memory. Finally, functions to access neighborhood values directly use arrays $cdeg^+$, $cdeg^-$, N_E^+, N_E^-, N_V^+ and N_V^-. As a result, neighborhood values can be obtained in $O(1)$.

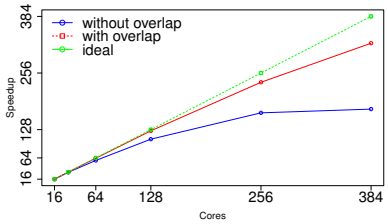
5 Experiments

SkelGIS performances have been evaluated using three different clusters to compute an arterial blood flow simulation [14]. This simulation is exclusively composed of double precision operations. Configurations of the three clusters are detailed in Table 1. The first one, at the university of Paris 6, is a well-equipped mid-size cluster. The second one, the TGCC-Curie in France, is the 20th cluster in the top500 list of November 2013. Finally, the third one, Juqueen in Germany, is the 8th cluster of the same list. Each experiment were evaluated four times, and the standard deviation of observed execution times is less than 1%. No specific optimization are proposed in SkelGIS for vectorization, however the simulation was compiled with the -O3 compilation flag.

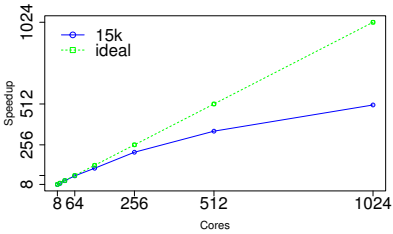
Table 1. Hardware specifications of clusters

Cluster	**System** (clock)	**Cores/n**	**Mem.**/n	**Comp** -O3	**Net**
Paris 6	2×Intel Xeon (3GHz)	12	24 GB	OpenMPI	InfiniBand
TGCC	2×SandyBridge (2.7GHz)	16	64 GB	Bullxmpi	InfiniBand
Juqueen	IBM PowerPC (1.6GHz)	16	16 GB	mpich2	5D Torus 40 GBps

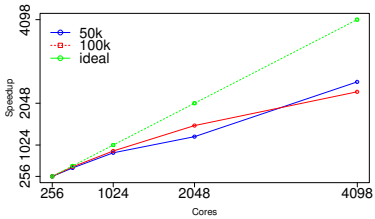
On the Paris 6 cluster has been analyzed performances of the overlap of computations with communications. A DAG with 15k nodes and edges, with an average degree of nodes equal to 2, and with a unique root, has been used. Results are shown in Figure 5(a) and clearly state that this optimization is convincing, as expected. For all other experiments, the overlap optimization is used. On the TGCC, the same 15k nodes/edges DAG has been used (Figure 5(b)) and the speedup scale linearly until 256 cores, which is inferior than on the Paris 6 cluster. However, the use of these clusters is very different. Indeed, on the Paris 6 cluster we were the unique user and we were sure that no other processes were running on the attributed machines. On the TGCC, which is massively used, machines are attributed at their maximum and it is almost sure that other processes were running on some machines. On the Juqueen cluster, three different sizes of DAGs have been used to evaluate performances: 50k, 100k and 500k nodes/edges DAGs. The three of them were similarly shaped with an average degree of nodes equal to 2, and with a unique root. Results are shown on Figures 5(c) and 5(d). Those computations were very long (8 hours for the 500k with 1024 cores), and hours of use on clusters are limited. Then for 50k and 100k DAGs speedups are relative to 256 cores, and for the 500k DAG, the speedup is relative to 1024 cores. The speedups scale linearly to 4098 cores for 50k and 100k DAGs, and to 8192 cores for the 500k DAG. One can note a knee in the scaling for the 50k DAG at 2048 cores. This is probably due to a weakness in the graph partitioning solution.



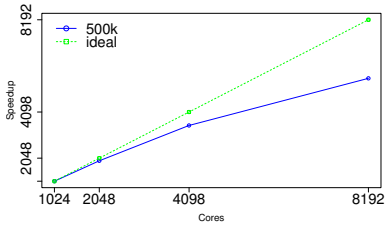
(a) Speedup obtained with comp/comm overlap on a 15k edges DAG on the LMM cluster.



(b) Speedup on a 15k edges DAG on the TGCC-Curie cluster.



(c) Speedups on both 50k and 100k edges DAGs on the Juqueen cluster.



(d) Speedup on a 500k edges DAG on the Juqueen cluster.

Fig. 5. Results

6 Related Work

There are several well established implicit parallelism libraries to solve mesh-based PDEs. Some of them are more specific and some others are more generic, however, the purpose is always to find the good level of abstraction to obtain the easiest solution for the end-user. PETSc [2] proposes specific solutions for each kind of mesh possible (matrices, sparse matrices, unstructured meshes etc.). It is based on specific functions to solve PDEs such as, for example, functions to *interpolate* or execute a *jacobian*. PETSc is close to a standard function-based library but is specifically made to solve PDEs. The framework OP2 [11], to solve unstructured mesh-based PDEs, is closer to SkelGIS in the chosen abstraction level. It relies on four concepts, first to define unstructured meshes, then to apply data and make computations on it: *sets*, *data on sets*, *connectivity* between the sets and *operations* over sets. However, it differs from SkelGIS on several points. First, using OP2, the user can define the needed mesh and its connectivity, through the definition of different *sets*. This point offers a great flexibility, however it is adapted to unstructured meshes, and it is not possible to define a network with it. In addition to this, the OP2 *operation* concept is different from the SkelGIS one. Actually, with OP2, a higher level of abstraction is proposed to the user through *operations*, where loops do not have to be explicitly written. Thus, the programming style of OP2 is closer to an algorithmic skeleton library [10]. In a SkelGIS *operation*, the user is in charge of loops, as in a

sequential program, through *Interfaces*. This last concept does not exist in the OP2 framework. Finally, SkelGIS is a library exclusively made of C++ header files (it is called a "h-only" library), while OP2 first generates code from code, before a final compilation. Liszt [7] is a *Domain Specific Language* (DSL) which stays close to a standard C programming style, where new language features and operators have to be used. Liszt, as OP2, is a solution to solve unstructured mesh-based PDEs, and the abstraction level is similar. The main difference is the way *sets* and their connectivity are defined. In OP2 the user is free to declare and connect as much sets as he wants. Using Liszt, on the other hand, *sets* are groups of mesh-elements pre-defined by the DSL: *vertices*, *edges*, *faces* and *cells*. Liszt is closer to SkelGIS than OP2 for its abstraction level. Actually, the Liszt code is close to a sequential program, as in SkelGIS. Loops are managed by the user through the concept of *for-comprehension*, which expresses computation on all mesh-elements in a set. Thus, *for-comprehension* could be compared to *iterators* of SkelGIS. However, it is not clear how the specific case of physical border elements can be parsed in Liszt, while using SkelGIS, specific *iterators* are available. Both OP2 and Liszt propose implicit parallelism solutions to solve unstructured mesh-based PDEs. The current version of SkelGIS can deal with two-dimensional regular meshes [5,6] but do not propose solution for unstructured meshes. This point is under study, and a specific new kind of *DDS* and its associated *DPMaps*, *appliers* and *interfaces* will be proposed. Note that SkelGIS, on the other hand, proposes a solution adapted to network simulations where nodes and edges do not form the cells of a mesh but a network. As far as we know, neither Liszt nor OP2 can manage such simulations. The same lack can be noticed for PETSc, however, using multiple sparse matrices, it seems possible but very complicated to work on networks.

7 Conclusion

SkelGIS is an implicit parallelism header-only library to solve mesh-based PDEs on distributed memory architectures. In this paper has been presented the implementation of SkelGIS for the specific case of networks which can be represented by DAGs. This implementation relies on an adaptation, a re-indexation and a parallelization of the CSR format for DAGs. The implementation of the associated *DPMaps*, *applier*, *iterators* and *neighborhood* is based on this new *DDS*, and its optimizations for cache lines and overlap of computations with communications. Experiments show that this solution scale on different architectures and different sizes of DAGs. Some improvement are under study to obtain a better graph partitioning algorithm and to use vectorization optimizations. Finally, SkelGIS is able to deal with two-dimensional meshes and network simulations, the next two steps are to propose implementations for unstructured meshes and for adaptive meshes. This last work will require mutable distributed data structures, not managed in the format presented in this paper.

References

1. Alexandrescu, A.: Modern C++ design: Generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
2. Balay, S., Gropp, W.D., Curfman McInnes, L., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Press (1997)
3. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM (1994)
4. Chevalier, C., Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(68), 318–331 (2008)
5. Coullon, H., Le, M.-H., Limet, S.: Parallelization of shallow-water equations with the algorithmic skeleton library SkelGIS. In: ICCS. Procedia Computer Science, vol. 18, pp. 591–600. Elsevier (2013)
6. Coullon, H., Limet, S.: Algorithmic skeleton library for scientific simulations: SkelGIS. In: HPCS, pp. 429–436. IEEE (2013)
7. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based PDE solvers. In: Proc. of 2011 Intern. Conf. for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 1–12. ACM (2011)
8. Nick Edmonds and Andrew Lumsdaine. Distributed compressed sparse row (2010)
9. Fishgold, L., Danalis, A., Pollock, L., Swany, M.: An automated approach to improve communication-computation overlap in clusters. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS 2006, pp. 290–290. IEEE Computer Society, Washington, DC (2006)
10. Javed, N., Loulergue, F.: Parallel programming and performance predictability with Orléans Skeleton Library. In: HPCS, pp. 257–263. IEEE (2011)
11. Mudalige, G.R., Giles, M.B., Reguly, I., Bertolli, C., Kelly, P.H.J.: OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: Innovative Parallel Computing (InPar), pp. 1–12. IEEE (2012)
12. Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience 14(3), 219–240 (2002)
13. Vastenhouw, B., Bisseling, R.H.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47(1), 67–95 (2005)
14. Wang, X., Fullana, J.-M., Lagrée, P.-Y.: Verification and comparison of four numerical schemes for a 1D viscoelastic blood flow model. Technical report, Institut Jean Le Rond d'Alembert - IJLRA (2012)