Who we are

I3S is the computer science laboratory of Université Côte d’Azur. It is located at the heart of Sophia Antipolis.

- COATI — theoretical and experimental aspects of graph algorithms. Software production: 3 librairies:
  - MascOPT network optimization (2001-)
  - Grph computing large graphs in-memory (2010-)
  - BigGrph platform — distributed library for computing largER graphs (2014-)

- SCALE — theoretical and experimental aspects of distributed computing. Software: ProActive, a platform for component-based computing.

COATI is hosted/supported by Inria.
Graphs, Digraphs, Hypergraphs

- Undirected Graph: Vertices + Edges (vertices pairs) ↔ Symmetric binary relation
- Directed graph: Vertices + Arcs = Couple of vertices
- Hypergraphs: (hyper)Edges = Groups of vertices

Mostly study the topology (structure) of the graph, however graphs are often weighted → values labeling the vertices and arcs.

Figure: Weighted graph, directed graph and Hypergraph
Graphs: becoming ubiquitous in sciences

some say pervasive ...

Useful model for ...

http://networkrepository.com/
http://linkeddata.org/
A graph with \( n \) vertices \( m \) edges graph can be encoded (stored) as:

Its *Adjacency Matrix* \( \rightarrow n \times n \) matrice.

for each vertex the list of neighbors \( \rightarrow n + m \).

Structure may allow compression.

**Figure**: Interval graph, Union of 3 cliques.
Graph representation, memory usage (II)

Labeling of the vertices matters.
- Hypercube of dimension $n +$ proper labeling $\rightarrow$ edges are encoded in the labels. With a \textit{random} labeling edges appear as arbitrarily.
- In a Tree one can always label the sons of a vertex consecutively. A node on store only the ID of its first neighbor and its degree.
- In a subgraph of a Grid one can always label the potential neighbors of a node as 0, 1, 2, 3.

\textbf{Figure:} The hypercube of dimension 4, a Tree, and a subgraph of the grid.
There are cases in which the natural representation is not the list of edges.

Using Alternative representation

- For a planar graph, a planar embedding (as example the list of faces) may be necessarily to run efficiently the algorithms.
- For an interval graph, the natural representation is to encode node as intervals.
- More generally additional information, such as a Tree decomposition may be useful.
Graph representation: Very large graphs

Some specificities of very large graphs

- Constants do matter, using high level abstract data structures increase the memory footprint by a large factor. Efficient solutions are often ad-hoc.

- For very large graph finding and using some hidden specific structure or compressing the graph representation may be unfeasible.

- There are many cases in which some aspect of the structure are known in advance. As example graphs in the plane or physical space, graph for which a natural partitioning do exist.
We may distinguish two different but related types of questions:

A) Determine some properties of the graph per se.

B) Find some properties of the graph that allow to answer to the questions of type A.
A few Graph properties, type A questions

**Statistics**

- degree sequence, average distance, average connectivity
- (approx) count small subgraphs, (e.g. count triangles)
- correlation and clustering
  
  \[
  \text{Prob}[(u, v) \in E \mid \{(u, z), (z, v)\} \in E]
  \]

**Global properties**

- (strongly) connected components, (approximated) Minimum Dominating Set.
- Diameter.
Approximated representation & compression

- Find a Map $f : G \rightarrow R^d, l_1$ which “preserve” the distances
  \[
  \frac{1}{\rho} \leq \frac{d(x,y)}{d(f(x),f(y))} \leq \rho
  \]  
  (low distorsion mapping).

- Find a simple Random Graph model such that $G$ looks like a
  typical event drawn from the associated distribution (bloc
  models, preferential attachment models, random graph in the
  Euclidian plane).

- Fit $G$ into an existing random graph model.

- Determine clusters in $G$, Find congested cuts.
Goal:

One wish to use a cluster of multi-core computer to implement some of these algorithms.

BSP is a message-based iterative distributed algorithm. It runs a sequence of steps. During a step:

- All messages sent at the previous steps are delivered
- All vertices in the graph are scheduled for execution

The algorithm stops when no messages remain.
Practical Implementation for large graphs?

**key performance factors**

- Can we manage to fit the graph in the RAM?
- **Multi-threading!** NEF provides CPU with 48 threads.
- 48 cores ~ PRAM with 48 processing unit.
- Can we split the data or space search without too much synchronization & communications.

**Tricks: Sampling, Monte Carlo methods**

- Some properties can be derived from a sample of the graph (select randomly a subset of $V$ or a subset of $E$).
- → can work on a smaller graph that fits in the RAM.
- Distant computer can work on different “chunk” of the graph.
Challenges for Distributed Algorithms

**BSP framework**

- Each Node is assigned a set of vertices.
- **Processing phase** = local computations, communication via the RAM.
- **Update phase** = communications, synchronization,

Performance collapses if the graph is random (or if the vertices are mapped randomly on the nodes).

\[
\text{Amount of communications } |S| \overline{d} |S| \times \frac{|V \setminus S|}{|S|} \sim \overline{d} |S|
\]
Structured data → low communication overhead

Structure is important

- There are many cases in which communications are lower.
- for grid or planar network the border of a set $S$ is only $Θ(\sqrt{|S|})$.
- **Bad news:** for most random networks $|Γ(S)| = Θ(|S|)$.

**Figure:** Grid-like graph, the data is affected to 4 computing nodes.
A practical case: A snapshot of twitter

### Input graph

- Twitter data set (crawled by A. Legout/INRIA-DIANA):
- 240GB on disk, 398M vertices, 23G edges
- Average degree of 58 and max degree 24,635,412

### Goals

- Compute the Strongly connected Components
- Compute the number of $TT_3$ and $K_{2,2}$.
- Compute the diameter.
Suitability of existing frameworks

mainly Two platforms: Giraph (atop Hadoop), GraphX (atop Spark).

many flaws

- limited support for graph and programming models
- poor memory performance (GraphX can't load our large Twitter graph dataset)
- unreliable (GraphX again) steep learning curve (GraphX is written in Scala) while lacking flexibility and documentation.
- unsuitable for experimentation (slow startup, low monitoring, etc)
Our own solution: The BigGrph library

- Developed since 2014 upon Grph (single computation flow library)
- a Java library for the manipulation of very big graphs.
- originally developed in a joint-project of Coati, Scale and Diani Inria teams: Inria provided a Research Engineer during 4 years.
- Objective: running algorithms on bigdata -large graphs
BigGrph workflow

BigGrph’s workflow consists of:

1. deploy the executable code
2. bootstrap the application (incremental using rsync ; takes less than a second)
3. partition the graph, by loading each piece on cluster nodes (arbitrary only)
4. perform the distributed computation (BSP model)
5. get the result
List of algorithms

- Single-source shortest path (Dijkstra, BFS)
- iFUB (Compute the diameter using a “few” shortest path runs).
- Page Rank
- Connected Strongly Connected components.
- Clustering coefficients, triangle counting
- Numerous stats (degrees, counting, etc)
BigGrph’s performance?

BigGrph:
- loads the graph 20x faster than Giraph
- computes BFS 3x faster than Giraph, 4x faster than GraphX
- uses 3x less memory than Giraph
- can load the big Twitter database (even on 24GB workstations) while GraphX cannot (even on 192GB cluster calculators)
Limitation of BigGrph

Why we decided not to build upon Grph

- Abstract high level library → memory intensive.
- not designed for multi-threading.
- designed to hide the implementation → not suitable for fine tuning.
- It was too complex (it took many days for our engineer to implement the SCC algorithm)
Our solution

Jmaxgrph

- Just like most of others, it is written in Java, because it is the most used, taught, clean, portable, complete language/platform today
- Low memory footprint.
- Non blocking data structures
- Target platform: Unix 64-bit (all Linux distributions, MacOSX)
- Use straightforward array structures.
Important side functionalities

The framework offers non-core functionalities that are essential.

- deploy the executable code
- bootstrap the application
- partition the graph, and load each piece on cluster nodes
- execute in parallel, communicate
- get and centralize the results
Computing strongly connected components

- Tarjan algorithm cannot be implemented transparently.
- Instead we compute local SCC → reduce the instance size.
- The we perform 2 BFS.
- Last we call the algorithm recursively.

**Performance**

<table>
<thead>
<tr>
<th>Node Configuration</th>
<th>Computation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>One node (512 GB RAM)</td>
<td>7:00 hours</td>
</tr>
<tr>
<td>8 nodes</td>
<td>3:10 (gc)</td>
</tr>
<tr>
<td>12 nodes</td>
<td>2:20</td>
</tr>
<tr>
<td>16 nodes</td>
<td>2:35 (more messages)</td>
</tr>
</tbody>
</table>

Largest SCC size = 256M vertices (64% of \( V \)); 141 M of size 1; 651,000 Of size 2; ... typical random graph phenoma of isolated singleton or pairs.