HPC-Big Data Convergence: A library to apply highly parallel algorithms on big data clusters

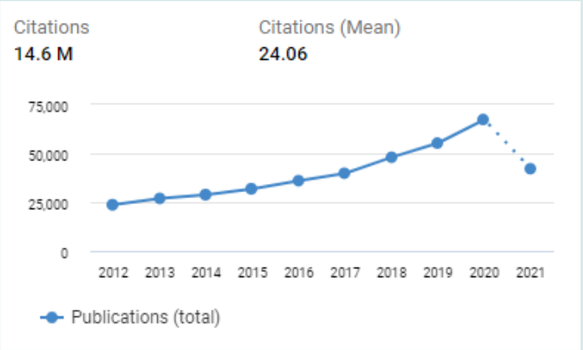
HPC-SCALABLE-ECOSYSTEM and SOLHARIS days
July 2021

Speaker: Arthur CHEVALIER

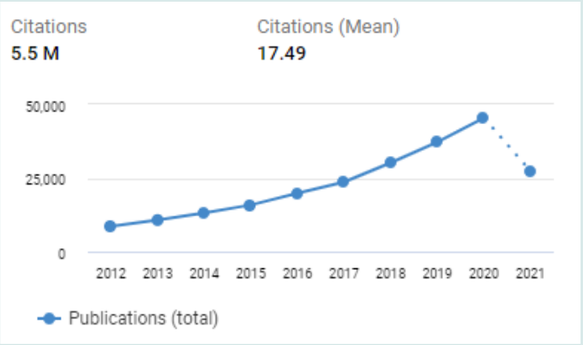
Context and goals 1

- Lots of research about HPC: 42,134 citations just for 2021*
- Lots of research about Big data: 27,365 citations just for 2021*

How can we use HPC algorithms on huge dataset ?



HPC

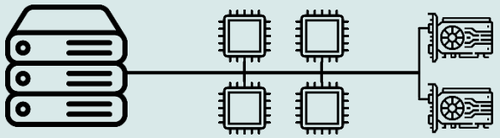


Big Data

*source: https://app.dimensions.ai/

Context and goals 2

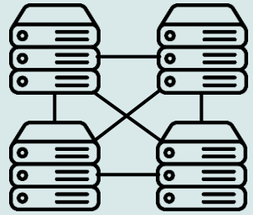
HPC



- Based on many-cores nodes and heterogeneous configurations
- Fast runtime language (C/C++, Fortran, …)
- CPU/GPU intensive applications

Example: use of FMM for material physics

Big data



- Based on many-cores nodes and high-memory/storage configurations
- Majorly in Java, Scala, Python, …
- Memory/Storage intensive applications

Example: use of FMM for graph drawing

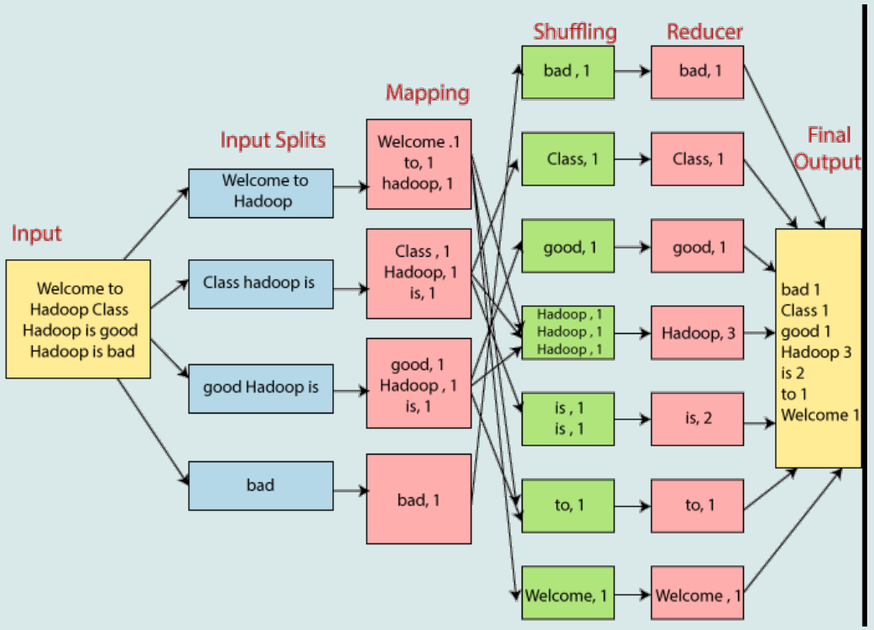
Are we able to perform a simulation of tens of billions of particles or can we speed up graph drawing using HPC ?

Big data architecture

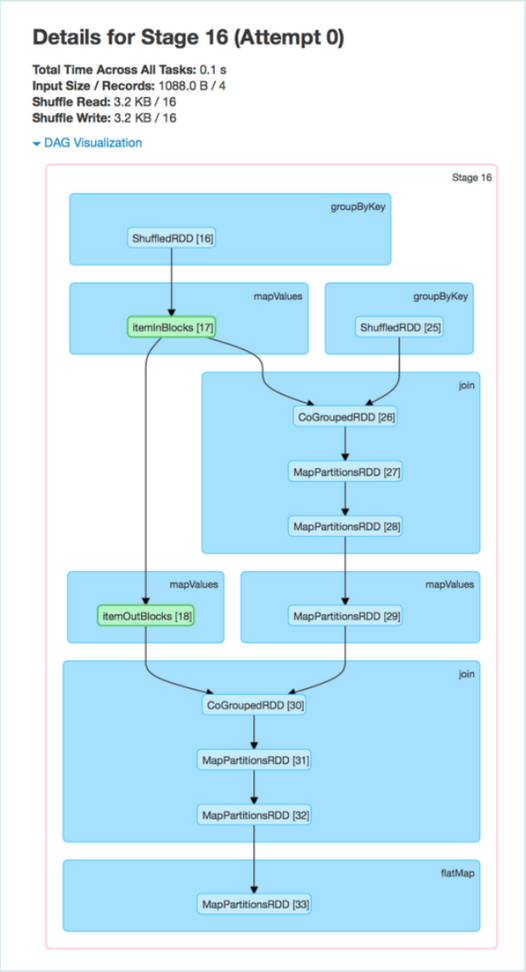
Map-Reduce | Spark

Yarn: Resource manager

HDFS: Storage



Map-Reduce flow



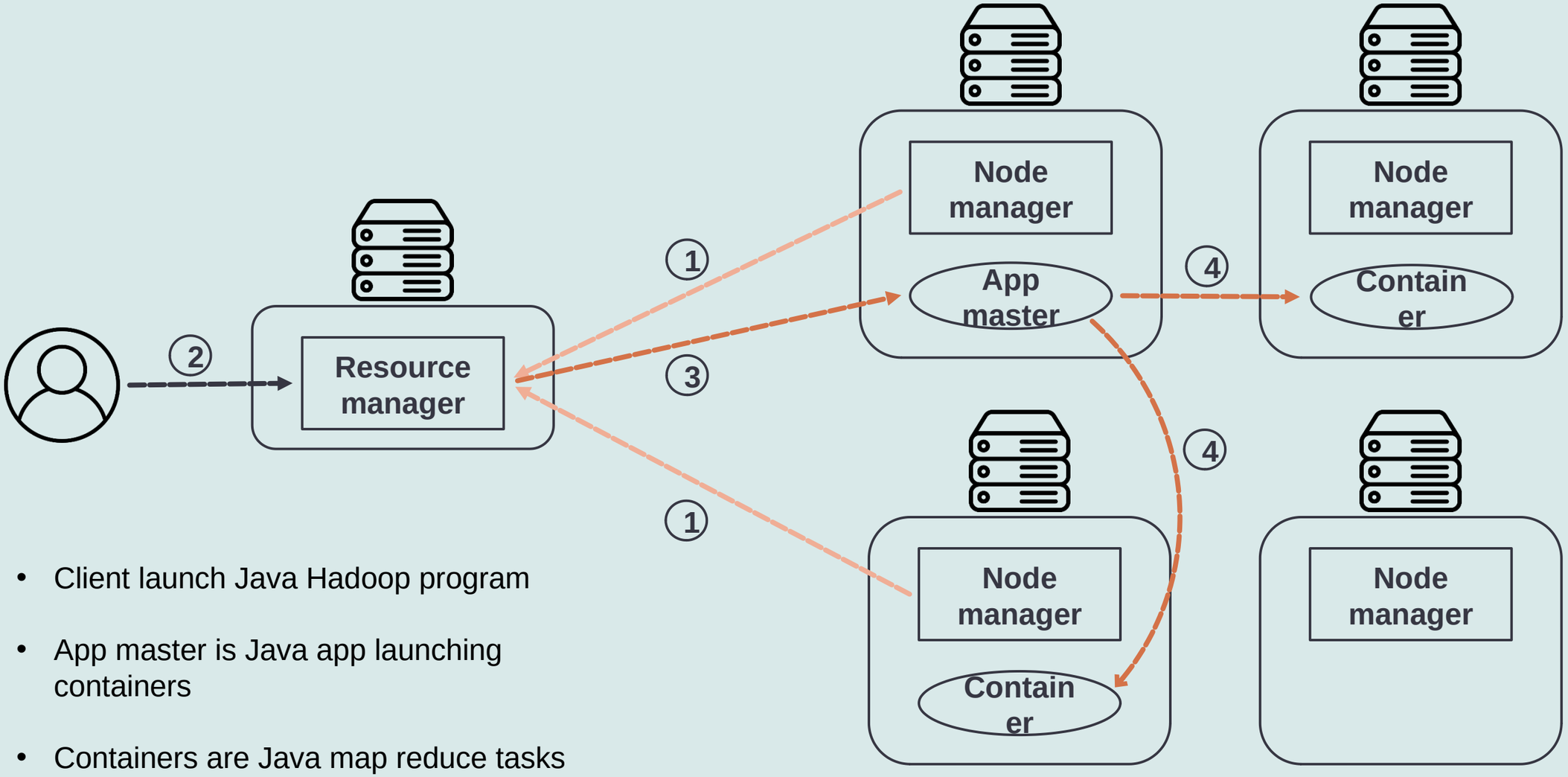
Spark task DAG

Existing tools

- Map Reduce for C (mr4c) by Google (stopped since 2015)
- Use of Java Native Access/Interface to use C compiled code from Java
- Providing Spark with custom libraries (.so)
- Use of Intel MKL in Spark tasks for IA in Big data (BigDL)

No tool allows the use of native code only. We are bound to Java.
We, therefore, introduce Yarn++: a library allowing any C/C++ based code to run on Hadoop clusters without special configuration.

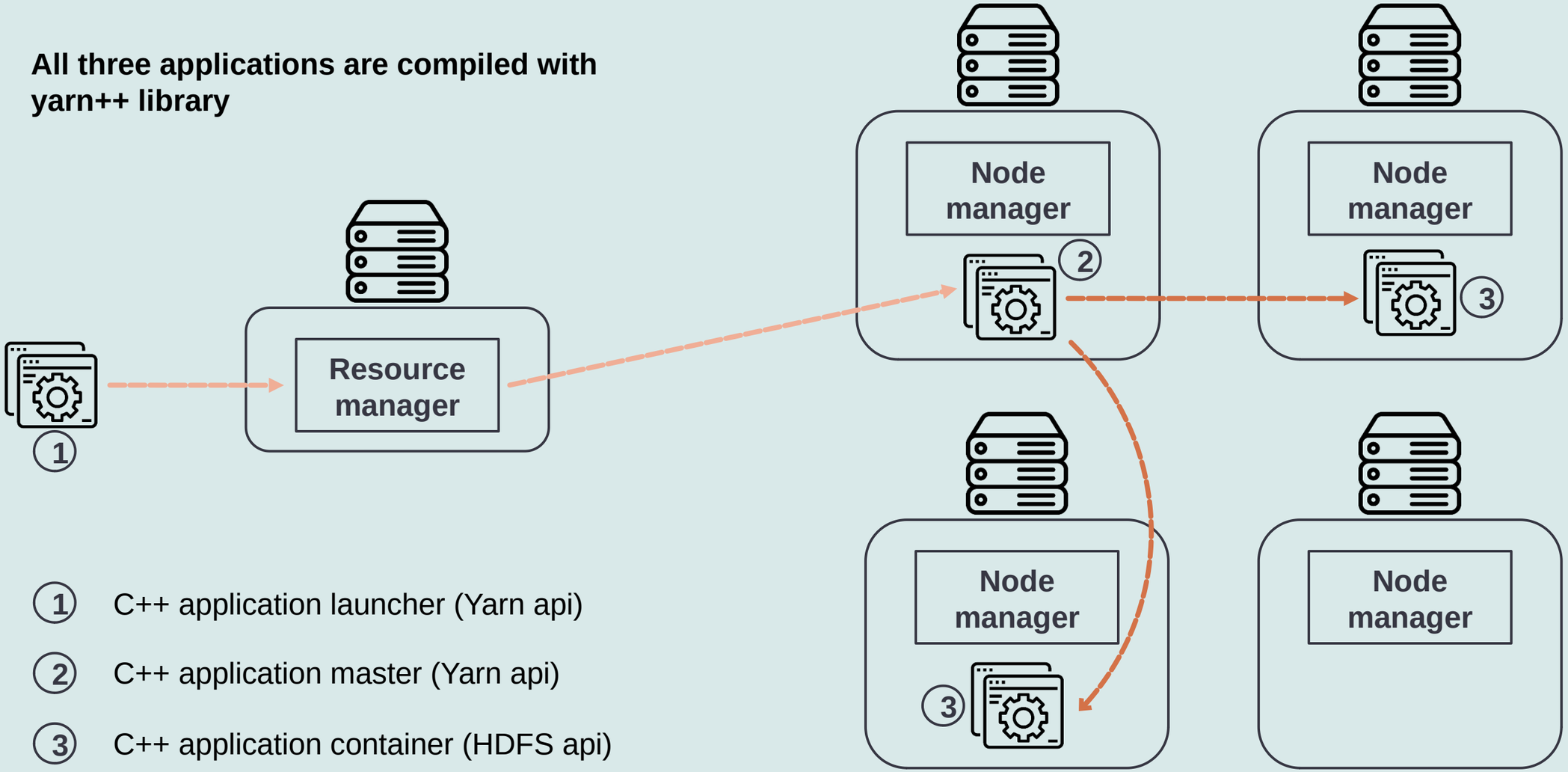
Architecture of Hadoop clusters



- Client launch Java Hadoop program
- App master is Java app launching containers
- Containers are Java map reduce tasks

Yarn++ library

All three applications are compiled with yarn++ library



1. C++ application launcher (Yarn api)
2. C++ application master (Yarn api)
3. C++ application container (HDFS api)

Usage: Launcher

```cpp
#include <Yarn++/ApplicationClient.hpp>

Yarnpp::ApplicationClient client(host, port, "appClient", 1);

…

cout << "Creating application..." << endl;
int32_t appId = client.createApplication();
cout << "Got application ID: " << appId << endl;

Yarnpp::Resources resources = client.getAssociatedCapabilities(appId);
cout << "Associated capabilities:" << endl;
cout << "Memory: " << resources.memory() << endl;
cout << "Virtual cores: " << resources.virtual_cores() << endl;
cout << "Resources:" << endl;
for (auto itr = resources.resource_value_map().begin(); itr != resources.resource_value_map().end(); ++itr) {
    cout << "  - " << (*itr).key() << ": " << (*itr).value() << " " << (*itr).units() << endl;
}

// Creating command from options
stringstream commandStream;
commandStream << master << " -h " << host << " -p " << masterport;
string command = commandStream.str();

// Setting environment variables
vector<pair<string, string>> envVars;
envVars.push_back({"LD_LIBRARY_PATH", ".:/usr/local/lib/sasl2:/usr/local/lib"});

cout << "Submitting app with command: " << command << endl;
client.submitApplication(appId, "appClient example", 50, 1, command, envVars);
```

1. **Create launcher**
2. **Register app**
3. **Get hardware info**
4. **Create command:**
   /bin/master –h … -p …
5. **Modify master environment**
6. **Launch command**
   50MB of memory and 1 vcore here

Usage: Master

```
#include <Yarn++/ApplicationClient.hpp>

Yarnpp::ApplicationMaster master(host, port, 1);

…

master.registerApplicationMaster();

// Launch containers or computations

master.finishApplicationMaster(Yarnpp::ApplicationStatus::APP_SUCCEEDED);

return EXIT_SUCCESS;
```

1. **Create master app**
2. **Register master in Hadoop**
3. **Update Hadoop with status**

App master can:

- Get hardware info
- Launch containers (same way the client did)
- Update Hadoop during execution
- Executing computation itself

App container:

- Is basic C/C++ app
- Can use low level network
- Can use HDFS:

```
// Try to access HDFS file
string path("/data/big.txt");
hdfs::URI uri = hdfs::parse_path_or_exit(path);

hdfs::FileHandle *file_raw = nullptr;
hdfs::Status status = fs->Open(path, &file_raw);

//wrapping file_raw into a unique pointer to guarantee deletion
unique_ptr<hdfs::FileHandle> file(file_raw);

status = file-
>Read(input_buffer, sizeof(input_buffer), &last_bytes_read);
```

First performance tests

Simple performance test:

- Hadoop fs –ls versus Yarn++ is:

~1.5s for Hadoop and 0 for **Yarn++** (elapsed time is in sec.)

- Hadoop fs –cat versus **Yarn++** is:

	Hadoop	Yarn++
Sherlock complete work (6.2MB)	~2sec	0sec
Enwik9: first 10^9B of eng. wiki (953.7MB)	~1.5min	~7sec

Ongoing common map-reduce examples:

- Word count:

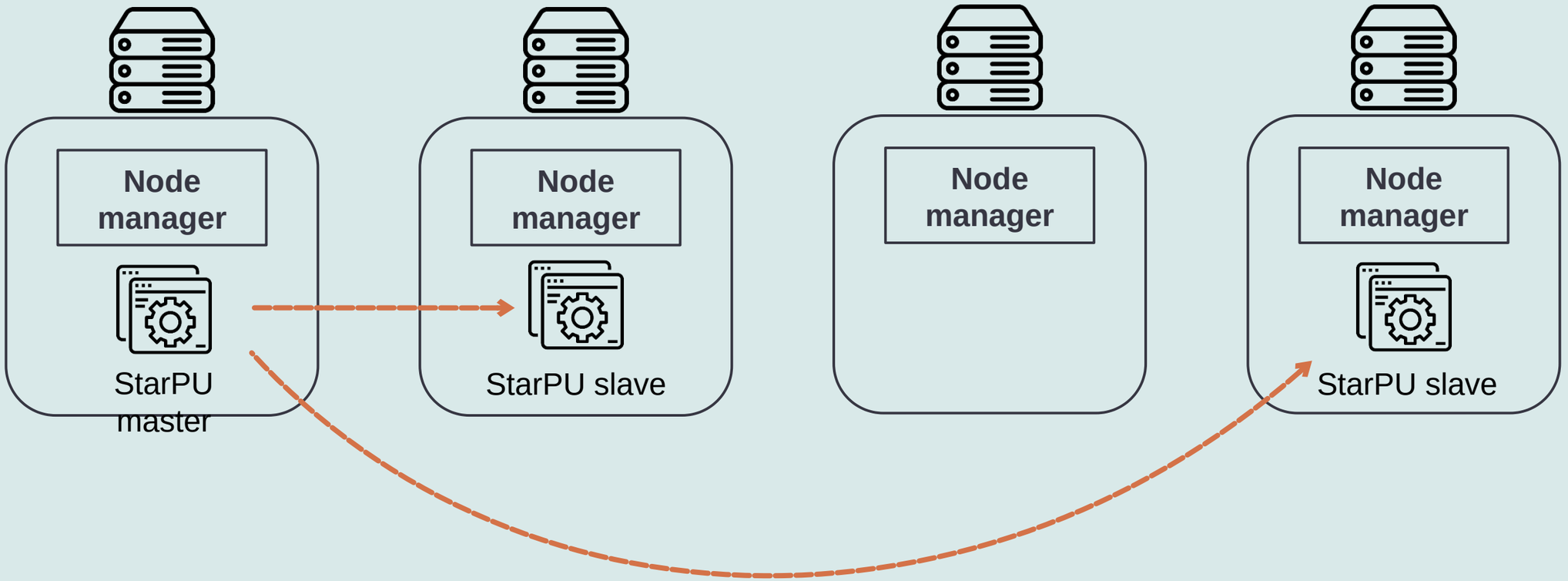
Good results but still ongoing

- K-Means on big dataset

Intensive use of map-reduce but still ongoing

Using Yarn++ for HPC: current work

As app master and containers are C/C++ based applications, we can use HPC runners on them like StarPU*.
Since 1.3, StarPU has MPI master-slave support.



*https://starpu.gitlabpages.inria.fr/

Current goal

Working with MABioVis team of LaBRI on graph visualization.
Contact: David AUBER

- Use of FM³ (Fast multipole multilevel method) iterations to draw huge graph
- Possible convergence between FM³ and ScalFMM (on-going work by Pierre Esterie)
- Existing StarPU implementation of ScalFMM
- Launching FM³ algorithm on big data clusters with Yarn++ and new ScalFMM should be possible

Typical objective: Drawing the Facebook interaction graph

Future work

- Use StarPU in master-slave mode (before implementing parallel unrolling)
- Use one ScalFMM algorithm (Cholesky) to compare performance on Big data cluster
- Performance test on many-nodes cluster (LSD/Plafrim)
- Comparison with Hadoop Map-Reduce and Spark performances
- Try using Spark as scheduler but send tasks to underlying StarPU for computation

Thanks

Contact at:

arthur.chevalier@u-bordeaux.fr

Credits to monkik, becris, those icons, linector, and pixel perfect for images