Survey of Technologies for Developers of Parallel Applications OmpSs, PyCOMPSs, Parsl, Swift

Daniele Lezzi – Barcelona Supercomputing Center

Parallel Applications Workshop, Alternatives to MPI+X

Held in conjunction with SC22:
The International Conference for High Performance Computing, Networking, Storage, and Analysis

PAW ATM

Parallel Applications Workshop, Alternatives to MPI+X

Monday, November 14th, 2022
Hybrid Event

Held in conjunction with SC22:
The International Conference for High Performance Computing, Networking, Storage, and Analysis

SC22 Dallas, TX | hpc accelerates.

Parallel Applications Workshop
Alternatives to MPI+X

Monday, November 14th, 2022

Outline

- OmpSs programming model
  - OmpSs-2 and TA-X software stack
  - Runtime challenges on hybrid programming
- COMPSs programming framework
  - PyCOMPSs programming model
  - The COMPSs Runtime
  - Distributed machine learning
- PARSL: parallel programming in Python
  - PARSL task-based applications
  - PARSL executors
- SWIFT parallel scripting language

BSC vision on programming models

OmpSs-2
Programming Model

StarSs

OmpSs

@ SMP
@ GPU
@ FPGA

COMPSs

pyCOMPSs

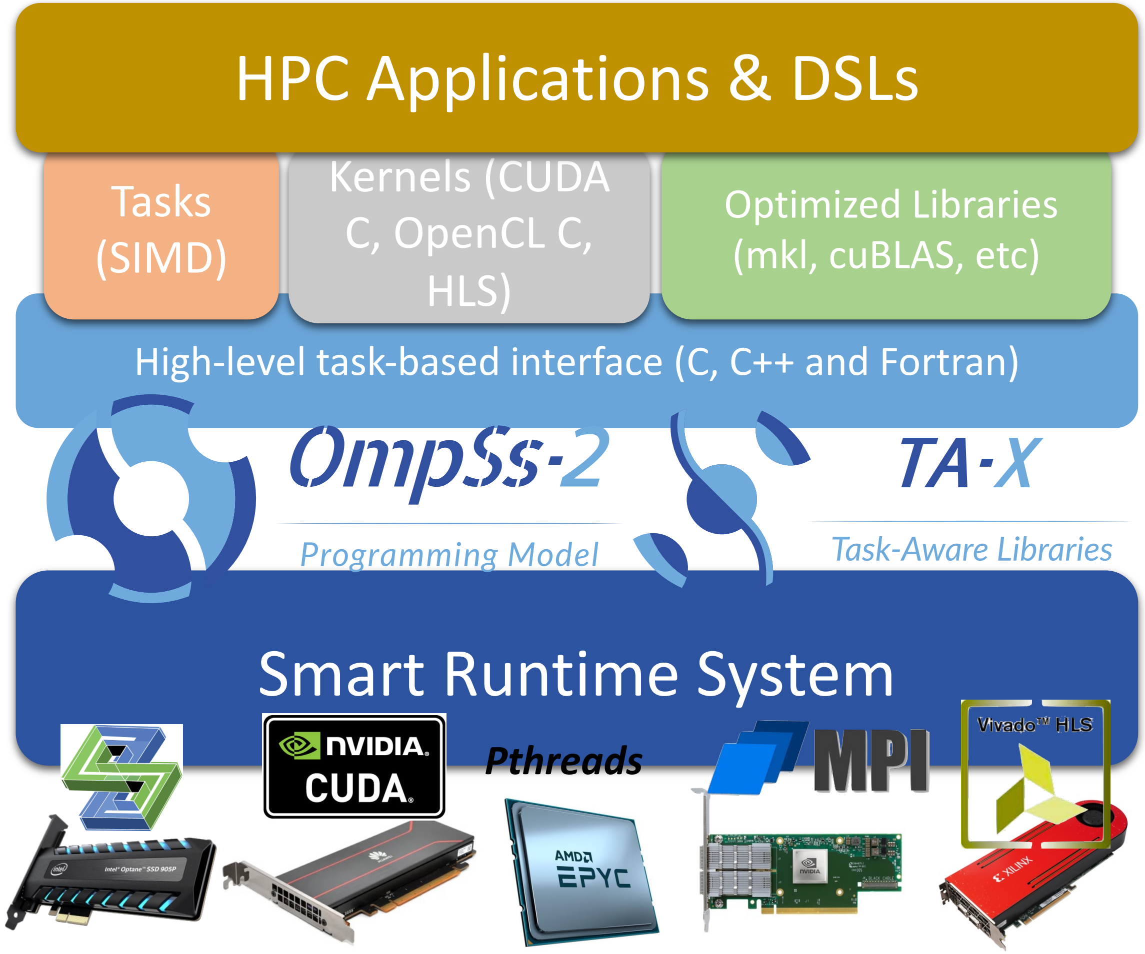
@ Cluster
@ Clouds
@ Edge

	OmpSs	COMPSs
Average task Granularity:	100 us – 10 ms	10 ms - 1 day
Dependences:	Memory address space	Files, Objects
Language bindings:	C, C++, FORTRAN	Java, C/C++, Python

Intranode ← → **Distributed**

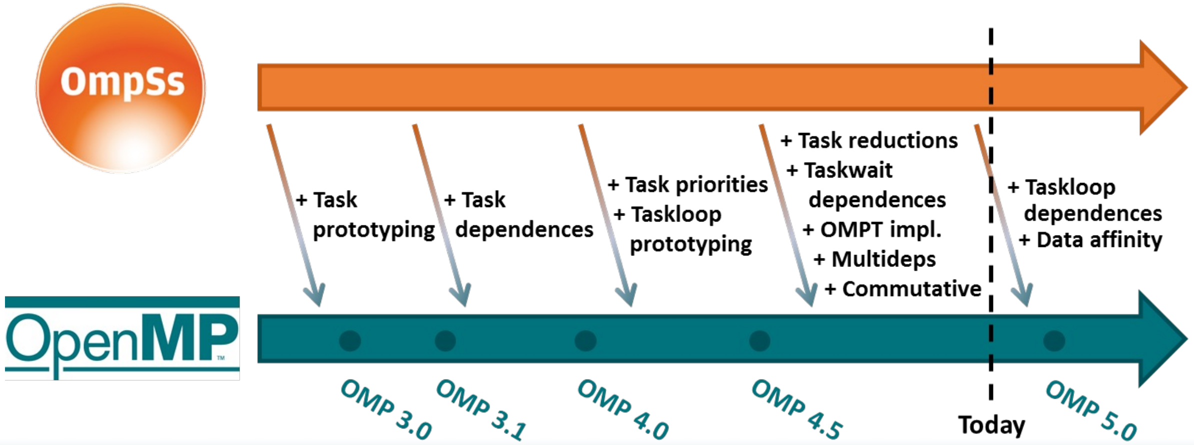
OmpSs-2 and TA-X software stack

- OmpSs-2 provides a high-level unified dataflow execution model to deal with:
  - Express parallelism: many-cores (tasks), accelerators (kernels), data transfers and leverage high-performance libraries
    - SIMD, CUDA C, HLS, OpenACC, etc
- Task-aware libraries
  - Communication libraries: MPI and GASPI
  - Storage: SPDK, io_uing
  - Offloading: CUDA, AscendCL, …



OmpSs-2 and TA-X software stack

- OmpSs-2 is the second generation of the OmpSs programming model:
  - OmpSs takes from **OpenMP** its viewpoint of providing a way to, starting from a sequential program, produce a parallel version of the same by introducing annotations in the source code
  - **StarSs**, is a family of programming models that also offer **implicit parallelism** through a set of compiler annotations.
    - Uses a different execution model, thread-pool where OpenMP implements fork-join parallelism
    - StarSs offers **asynchronous parallelism** as the main mechanism of expressing parallelism whereas OpenMP only started to implement it since its version 3.0.

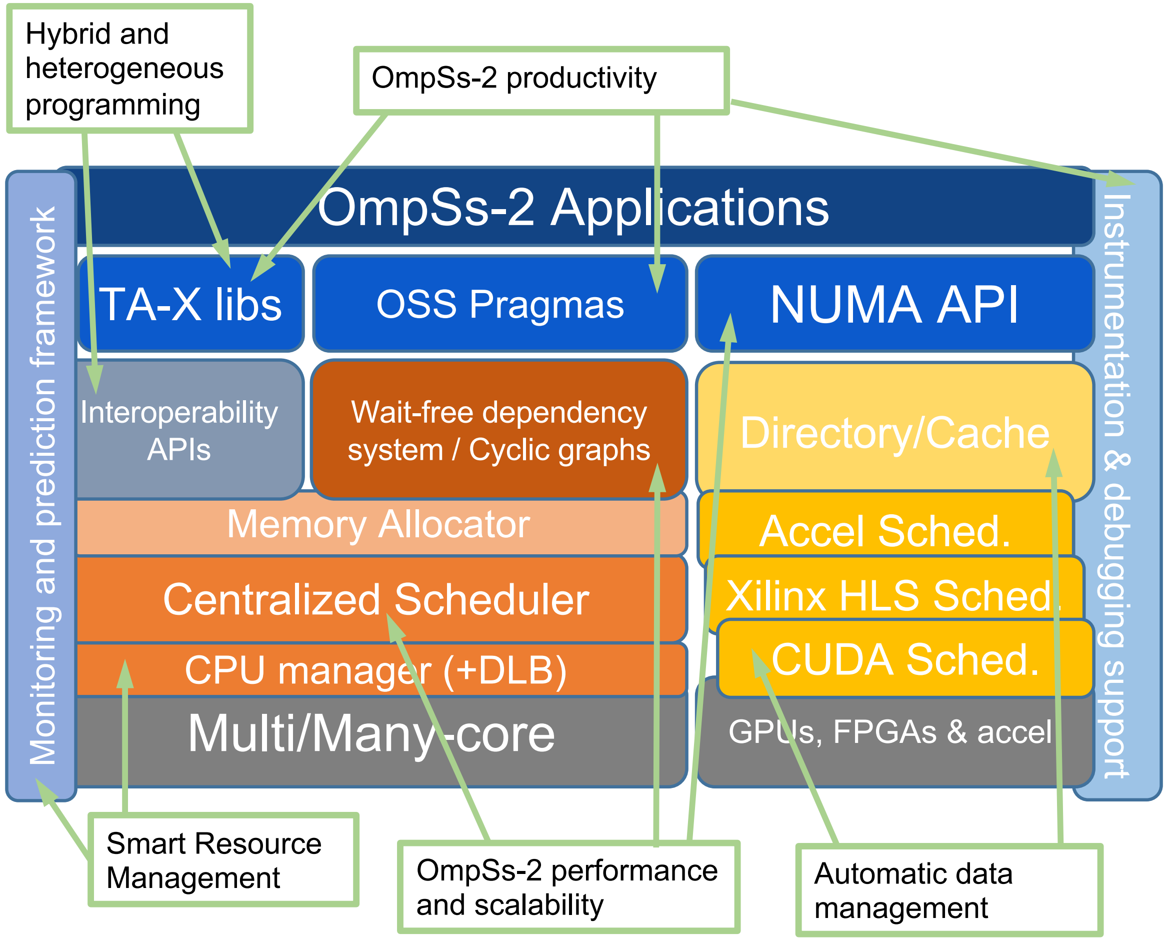


OmpSs-2 compilers and Nanos6 runtime

- The reference implementation of OmpSs-2 is based on the Mercurium source-to-source compiler and the Nanos6 Runtime Library:
    - The Mercurium source-to-source compiler provides the necessary support for transforming the high-level directives into a parallelized version of the application.
    - The Nanos6 runtime library provides the services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, and provide support for resource heterogeneity.

Source-to-source compiler
Multi-backed: GCC, LLVM, Intel, IBM, etc
Supports F90, C11 and C++11

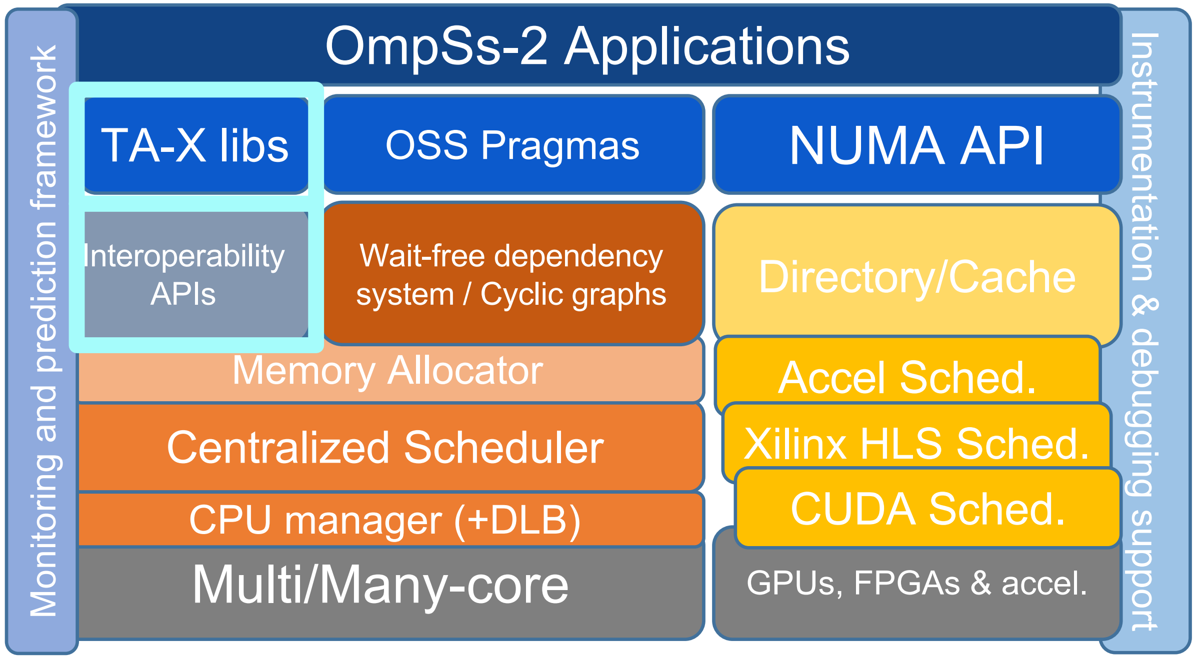
LLVM compiler with OmpSs-2 support
Supports C11 and C++20 (Fortran WiP)



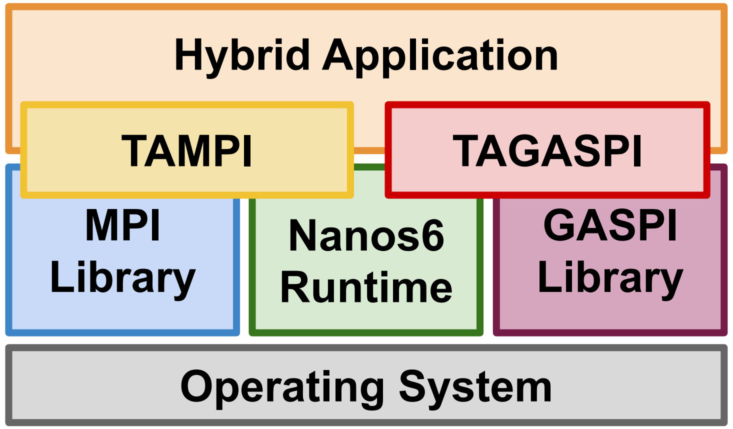
https://github.com/bsc-pm/ompss-2-releases

Runtime Challenges: Hybrid programming

Runtime interoperability APIs: Pause/resume (**blocking**) and event counter (**non-blocking**)
TA-X libraries: Task-Aware MPI (**TAMPI**) and Task-Aware GASPI (**TAGASPI**)



TA-X
Task-Aware Libraries



OpenMP vs OmpSs-2 features

Feature	OpenMP (Fork-Join)	OpenMP (tasks)	OmpSs-2 (tasks)
Supported parallelism	Structured	Structured, dynamic and irregular	Structured, dynamic irregular and nested
Synchronization method	coarse-grained barriers	fine-grained dependencies	fine-grained, weak and reduction dependencies
Runtime overhead	very low	low	very low – minimal
Load-balancing	poor	good	good
Data locality	excellent	poor	good
Programmability / Complexity	low	medium	medium
Overlap of computation phases	hard	easy	easy
Interoperability with other APIs	limited	limited / good(non-blocking APIs)*	**excellent**(blocking and non-blocking APIs)
Overlap of computation and communication phases	hard	hard / easy (TAMPI, …)*	**easy** (TAMPI, …)
Overlap of computation and data-transfer phases	hard	hard / easy (TA-CUDA, TACL, …)*	**easy** (TA-CUDA, TACL, …)

PyCOMPSs

- Sequential programming, parallel execution
- General purpose programming language + annotations/hints
  - To identify tasks and directionality of data
  - Task based: task is the unit of work
- Builds a task graph at runtime
  - Express potential concurrency
  - Exploitation of parallelism
- Offers a shared memory illusion to applications in a distributed system
  - The application can address larger data storage space:
    - support for Big Data apps
- Simple linear address space
- Agnostic of computing platform
  - Runtime takes all scheduling and data transfer decisions

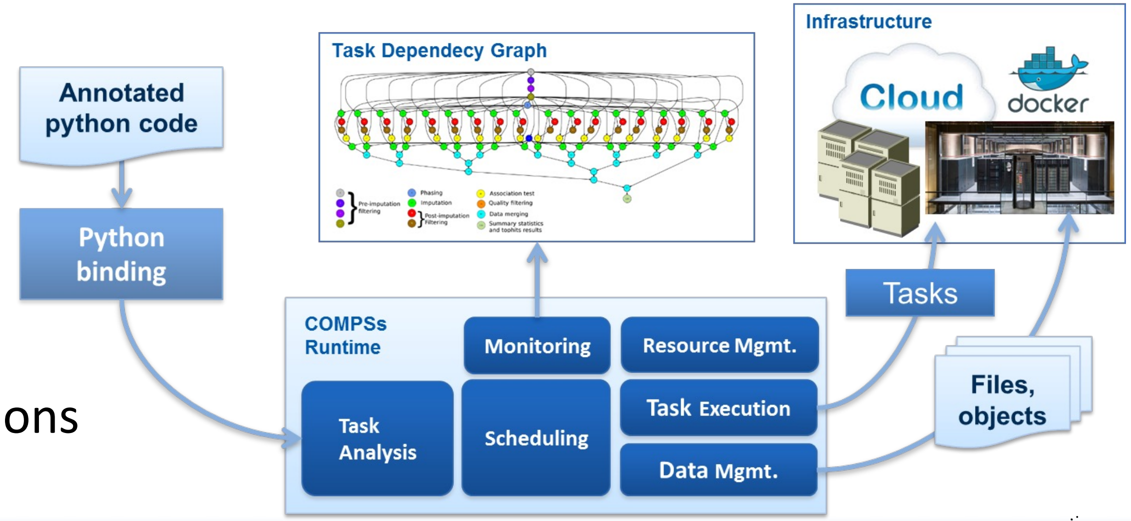
```
@task()
def word_count(block):
    ...
    return res

@task(f_res=INOUT)
def merge_count(f_res, p_res):
    ...
```

(a) Task annotation example

```
for block in data:
    p_result = word_count(block)
    reduce_count(result, p_result)
result = compss_wait_on(result)
```

(b) Main code example



PyCOMPSs adavanced features

- Task constraints: enable to define HW or SW requirements

```
@constraint (MemorySize=6.0, ProcessorPerformance="5000")
@task (c=INOUT)
def myfunc(a, b, c):
    ...
```

- Linking with other programming models:

```
@mpi (runner="mpirun", processes= "16", ...)
@task (returns=int, stdOutFile=FILE_OUT_STDOUT, ...)
def nems(stdOutFile, stdErrFile):
    pass
```

- Task failure management

```
@task(file_path=FILE_INOUT, on_failure='CANCEL_SUCCESSORS')
def task(file_path):
    ...
    if cond :
        raise Exception()
```

PyCOMPSs adavanced features

- Tasks in container images

```
@container(engine="DOCKER", image="ubuntu")
@binary(binary="ls")
@task()
def task_binary_empty():
    pass
```

```
@container(engine="DOCKER", image="compss/compss")
@task(returns=1, num=IN, in_str=IN, fin=FILE_IN)
def task_python_return_str(num, in_str, fin):
    print("Hello from Task Python RETURN")
    print("- Arg 1: num -- " + str(num))
    print("- Arg 1: str -- " + str(in_str))
    print("- Arg 1: fin -- " + str(fin))
    return "Hello"
```

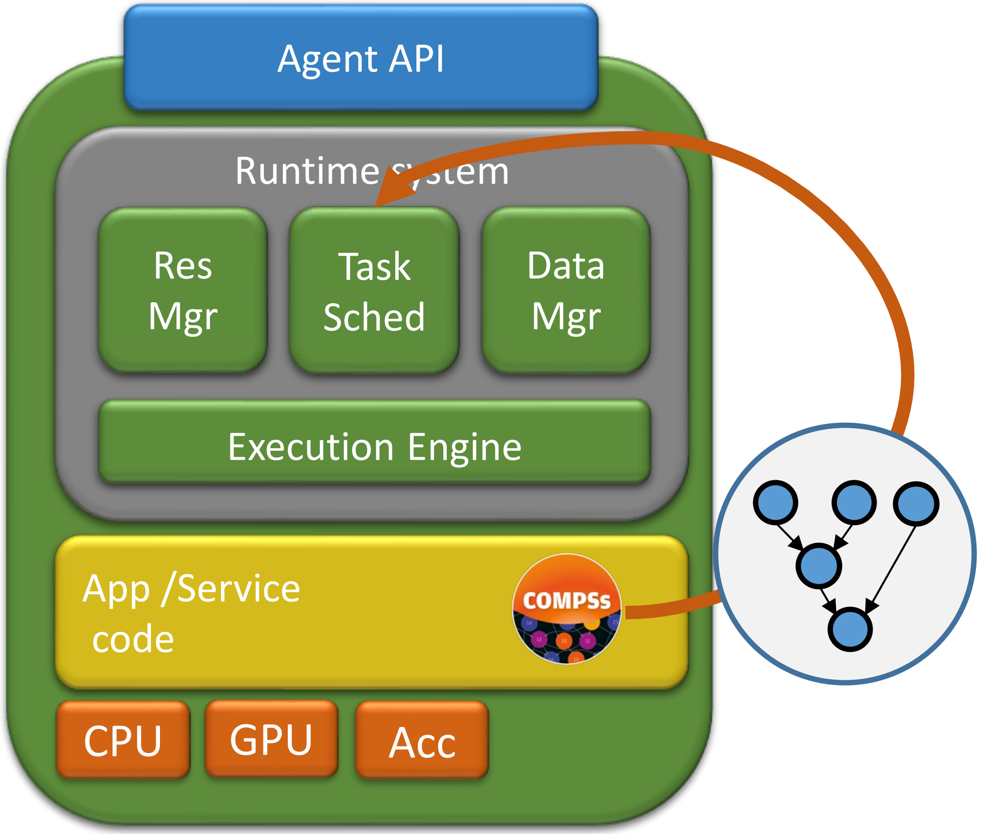
- Provenance
  - Enable reproducibility and replicability of COMPSs applications
  - use RO-Crate 1.1 Specification
- Checkpointing

Policy name	Class name	Params	Description
Periodic Time (PT)	es.bsc.compss.checkpoint.policies.CheckpointPolicyPeriodicTime	period.time	Checkpoints every n time
Finished Tasks (FT)	es.bsc.compss.checkpoint.policies.CheckpointPolicyFinishedTasks	finished.tasks	Checkpoints every n finished tasks
Instantiated Tasks Group (ITG)	es.bsc.compss.checkpoint.policies.CheckpointPolicyInstantiatedGroup	instantiated.group	Checkpoints every n instantiated tasks

```
from pycompss.api.api import compss_snapshot
        compss_snapshot()
```

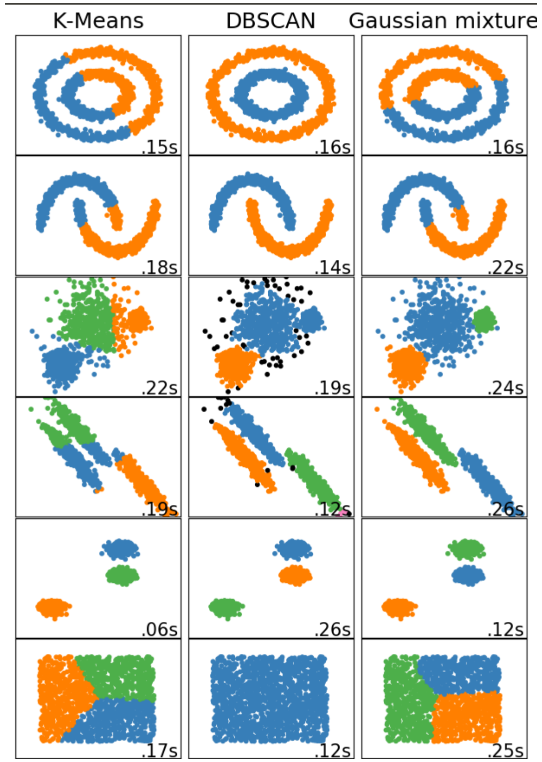
PyCOMPSs in computing continuum

- Runs as a standalone microservice
- The API receives function execution requests and sends a task into the runtime system (**FaaS** approach)
- Agents can offload part of the computation onto other agents by adding them onto their resource pool
- When a task is executed, the Programming Model detects **nested tasks** and requests their execution to runtime



Dislib: parallel machine learning

- dislib: Collection of machine learning algorithms developed on top of PyCOMPSs
  - Unified interface, inspired in scikit-learn (fit-predict)
  - Based on a distributed data structure (ds-array)
  - Unified data acquisition methods
  - Parallelism transparent to the user – PyCOMPSs parallelism hidden
  - Open source, available to the community
- Provides multiple methods:
  - data initialization
  - Clustering
  - Classification
  - Model selection, ...



DISLIB Distributed Computing Library

Parsl: parallel programming in Python

- Parsl's dataflow model allows data to be passed between Apps
- *Apps* define opportunities for parallelism
  - *Python* apps call Python functions
  - *Bash* apps call external applications
- Apps return "futures": a proxy for a result that might not yet be available
- Apps run concurrently respecting dataflow dependencies. Natural parallel programming
- Parsl scripts are independent of where they run. Write once run anywhere

```python
@python_app
def hello ():
    return 'Hello World!'

print(hello().result())
```

```
Hello World!
```

```python
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

with open('echo-hello.stdout', 'r') as f:
     print(f.read())
```

```
Hello World!
```

Parsl: parallel programming in Python

- Python's Concurrent.futures **executor** (runtime) interface
  - High-throughput executor (HTEX)
    - Pilot job-based model with multi-threaded manager deployed on workers
    - Designed for ease of use, fault-tolerance, etc.
    - <2000 nodes (~60K workers), Ms tasks, task duration/nodes > 0.01
  - Extreme-scale executor (EXEX)
    - Distributed MPI job manages execution. Manager rank communicates workload to other worker ranks directly
    - Designed for extreme scale execution on supercomputers
    - 1000 nodes (>30K workers), Ms tasks, >1m task duration
  - Low-latency Executor (LLEX)*
    - Direct socket communication to workers, fixed resource pool, limited features
    - 10s nodes, <1M tasks, <1m tasks
  - Others: WorkQueue, RADICAL-Cybertools, Flux

Swift & Parsl

swift

- 2000s — Virtual Data Language
- 2006 — Swift parallel scripting language
- 2010 — Swift/T scripting library
- 2015 — Parallel Works
- 2016 — Parallel scripting library (Parsl)

Swift

- C-like language with implicit parallelism
  - Manages calls to “leaf” tasks
  - Dynamic dependency graph
- Designed for execution across wide-area resources
- Supports diverse schedulers
- Manages data distribution
- Multi-site execution, coasters, etc.

```
type file;

app (file o) simulation (int sim_steps, int sim_range, int sim_values)
{
  simulate "--timesteps" sim_steps "--range" sim_range "--nvalues" sim_values
stdout=filename(o);
}

app (file o) analyze (file s[])
{
  stats filenames(s) stdout=filename(o);
}

int nsim   = toInt(arg("nsim","10"));
int steps  = toInt(arg("steps","1"));
int range  = toInt(arg("range","100"));
int values = toInt(arg("values","5"));

file sims[];

foreach i in [0:nsim-1] {
  file simout <single_file_mapper; file=strcat("output/sim_",i,".out")>;
  simout = simulation(steps,range,values);
  sims[i] = simout;
}

file stats<"output/average.out">;
stats = analyze(sims);
```

Swift/T

- Designed for HPC execution at extreme scale
  - 1.5 billion tasks/s
- Swift language implementation that supports arbitrary leaf functions
- Runs as a single MPI job using a scalable, distributed-memory runtime based on Turbine and ADLB
- Data movement via MPI
- Rapidly subdivide large partitions for MPI jobs