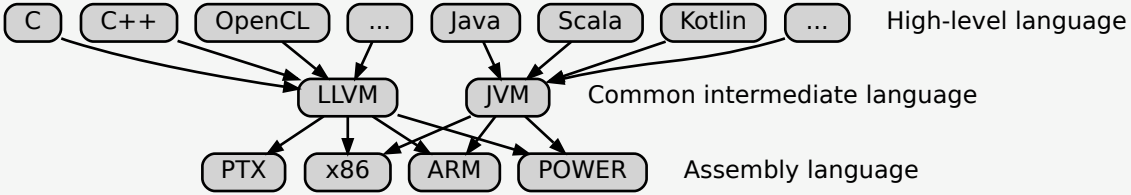
Intermediate language for parallel and distributed computing

Ivan Gankevich

Saint Petersburg State University

May 2021

Motivation

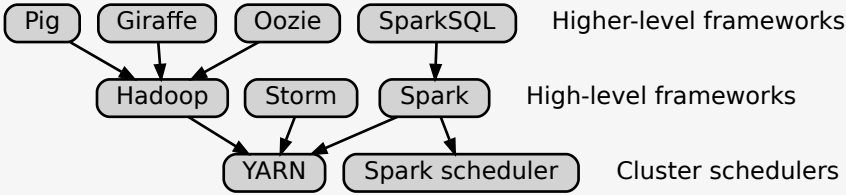


LLVM, JVM: no parallelism, no distributed computing, no fault tolerance.

Current state of the art

Distributed and parallel system = common intermediate language
+ cluster scheduler
+ high-level API.

In Big Data:



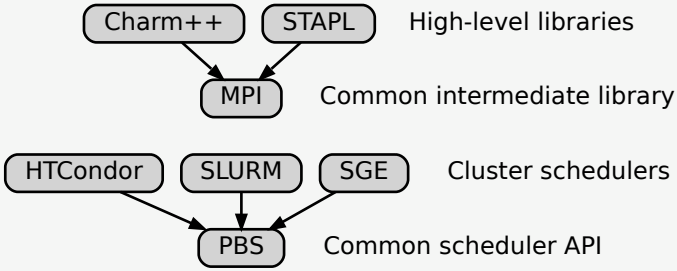
Problems:

- Duplicated effort.
- No intermediate library.
- Partial fault tolerance.

Current state of the art

Distributed and parallel system = common intermediate language
+ cluster scheduler
+ high-level API.

In HPC:



Problems:

- Schedulers are actually resource allocators (no load balancing).
- MPI and PBS are loosely connected.

Goals

> *Write parallel and distributed applications the same way as we write sequential ones.*

Programmers provide business logic, the system provides

- load balancing,
- fault tolerance,
- automatic parallelism (when possible).

Plan

- The intermediate language.
- The cluster scheduler.
- High-level interface.

The intermediate language
Asynchronous function calls

Kernel = data + code + result of the computation.

Regular function call:

```
int nested(int a) {
  return 123 + a;
}
```

Kernel:

```
struct Nested: public Kernel {

  int result;
  int a;

  Nested(int a): a(a) {}

  void act() override {
    result = a + 123;
    async_return();
  }
};
```

The intermediate language

Asynchronous function calls

Nested function call:

```
void main() {
  // code before
  int result = nested();
  // code after
  print(result);
}
```

Kernel:

```
struct Main: public Kernel {

  void act() override {
    // code before
    async_call(new Nested);
  }

  void react(Kernel* child) override {
    int result = ((Nested*)child)->result;
    // code after
    print(result);
    async_return();
  }
};

void main() { async_call(new Main); wait(); }
```

The intermediate language

Copying kernels

Regular function call:

```
int nested(int a) {
  return 123 + a;
}
```

Kernel (can be sent to other nodes):

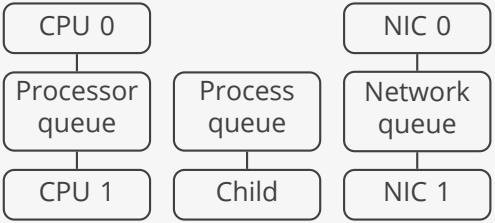
```
struct Nested: public Kernel {

  int result;
  int a;

  Nested(int a): a(a) {}

  void act() override {
    result = a + 123;
    async_return();
  }

  void write(Buffer& out) const override;
  void read(Buffer& in) override;
};
```

The intermediate language
Summary

- Minimal: each function call translates to an object with four methods.
  - `act` do useful work
  - `react` collect the results
  - `write` save
  - `read` load
- Parallelism: all calls are asynchronous.
- Load balancing: distribute kernels between cores, nodes etc.
- Fault tolerance: read kernels from and write to files, network connections etc.

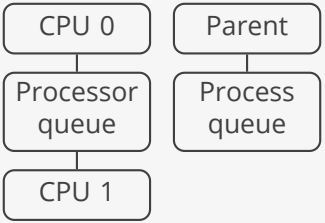
The cluster scheduler
Architecture

Daemon process:



- Run applications in child processes.
- Route kernels between application processes running on different cluster nodes.
- Maintain a list of available cluster nodes.

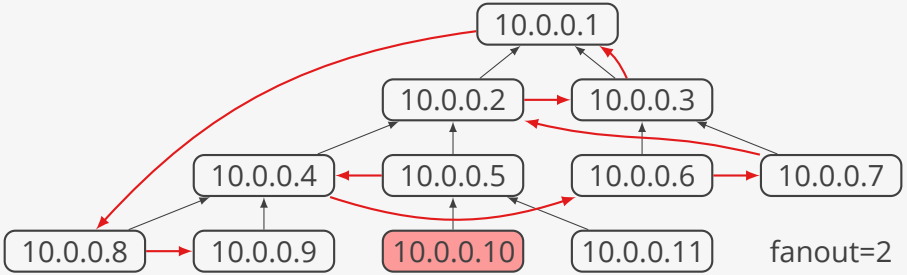
Application process:



`async_call`	push child kernel to the queue
`async_return`	push current kernel to the queue
`async_message`	send a kernel to another one via the queue

The cluster scheduler
Node discovery

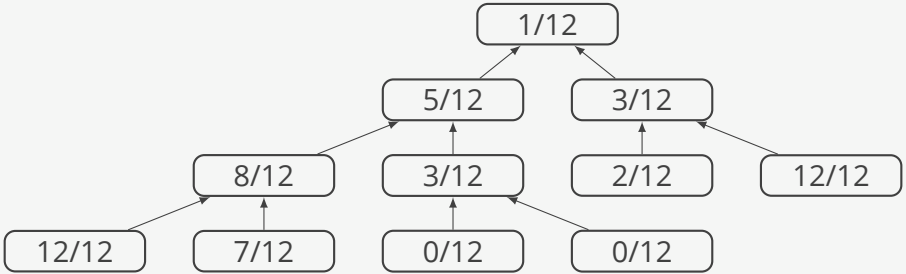
- Implemented using kernels and network queue.
- Ensures tree topology.
- Further communication uses this topology.
- Efficient for the large number of nodes (≈2 seconds for 400 virtual nodes).



I. Gankevich, Yu. Tipikin, V. Gaiduchok Subordination: Cluster management without distributed consensus, HPCS'15, 2015.

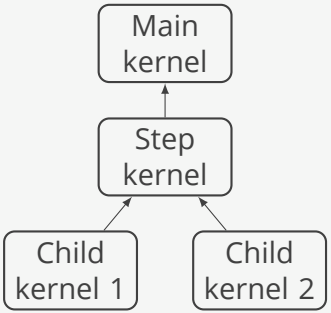
The cluster scheduler
Load balancing

- Track the kernels sent to other nodes.
- Track the kernels sent to each application process.
- Choose the neighbour with min. weight recursively.



The cluster scheduler
Fault tolerance

- Assumption: *main* kernel has only one child kernel at a time.
- Every *step* kernel (a child of *main*) has a copy of the *main*.
- Scheduler ensures that *main* and *step* are on different cluster nodes.
- Every *step* is also appended to the local log file.



Failure	Resolution
Child 1	resend Child 1 to the remaining nodes
Child 2	resend Child 2 to the remaining nodes
Step	resend Step to the remaining nodes
Main	restore Main from the copy
Main and Step	restore Main and Step from the log

- I. Gankevich, Yu. Tipikin, V. Korkhov Subordination: Providing resilience to simultaneous failure of multiple cluster nodes, HPCS'17, 2017.
- I. Gankevich, Yu. Tipikin, V. Korkhov, V. Gaiduchok, A. Degtyarev, A. Bogdanov Master node fault tolerance in distributed big data processing clusters, International Journal of Business Intelligence and Data Mining, 2017.

The cluster scheduler
Summary

- Minimal: kernels are used for both logic and as a protocol.
- Full control: all kernels go through the scheduler.
- Flexible load balancing: based on data locality, based on the load.
- Dynamic parallelism: application processes are created on demand.
- Fault tolerance: tolerates failure of any node at a time and electricity outage.

High-level interface

Subordination — reference implementation (cluster scheduler + high-level interfaces): https://github.com/igankevich/subordination.

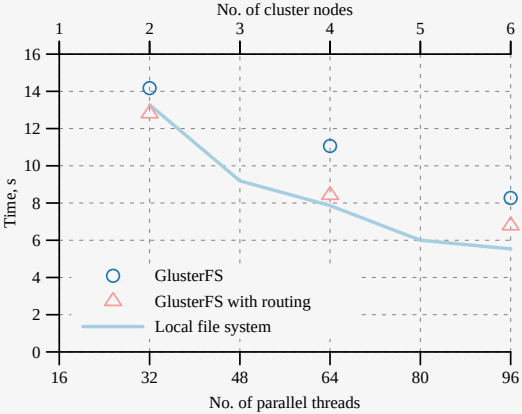
Language	Parallelism	Status
C++	manual	OK
Python	manual	OK
Guile	automatic	work-in-progress
LISP	automatic	work-in-progress
Javascript	manual	planned

File system	Status
GlusterFS	OK
HDFS	planned

Framework	Parallelism	Status
Java/Spark	manual	proprietary
MPI	manual	planned

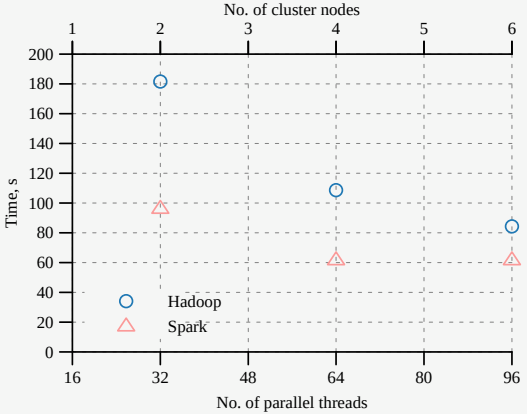
High-level interface

GlusterFS data locality



Hardware:

Processor	Intel Xeon L5630
No. of nodes	6
No. of processors per node	2
No. of cores per processor	4
No. of threads per core	2
RAM	24 GB
Network	Ethernet 1 Gbit



Software:

GlusterFS version	8.0
GCC version	7.5.0
Hadoop version	3.2.1
Spark version	3.0.0-preview2
GlusterFS Hadoop plugin version	2.3.13

A. Gavrikov, I. Petriakov, I. Gankevich *A case for data-driven scheduling in high-performance* HPCS'20, 2020.

High-level interface
Java/Spark

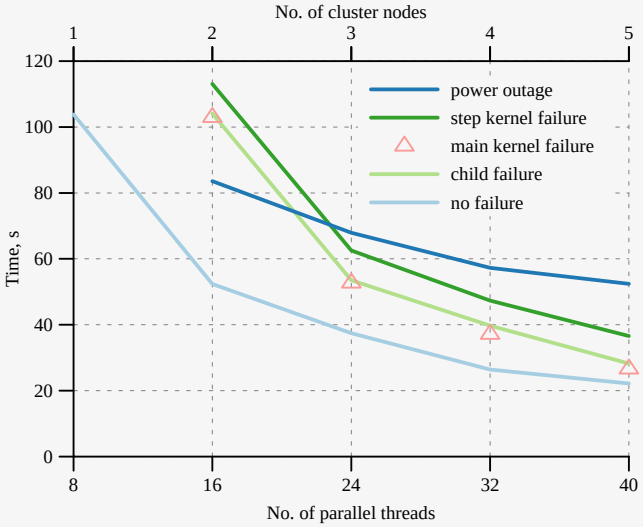
Benchmark: signal decimation using Fast Fourier transform on SoCs.

Platform	Scheduler	Interface	Average throughput points/s
Intel Edison	Spark	Spark	375
Intel Edison	Subordination	Spark	995
Intel Edison	Subordination	Subordination	517 676
Intel Core i5-4200H	Spark	Spark	487 594
Intel Core i5-4200H	Subordination	Spark	511 618
Intel Core i5-4200H	Subordination	Subordination	5 046 540

V. Korkhov, I. Gankevich, O. Iakushkin, D. Gushchanskiy, D. Khmel, A. Ivashchenko, A. Pyayt, S. Zobnin, A. Loginov
Distributed Data Processing on Microcomputers with Ascheduler and Apache Spark, Springer, 2017.

High-level interface

C++ with fault tolerance (NDBC spectrum preprocessing)

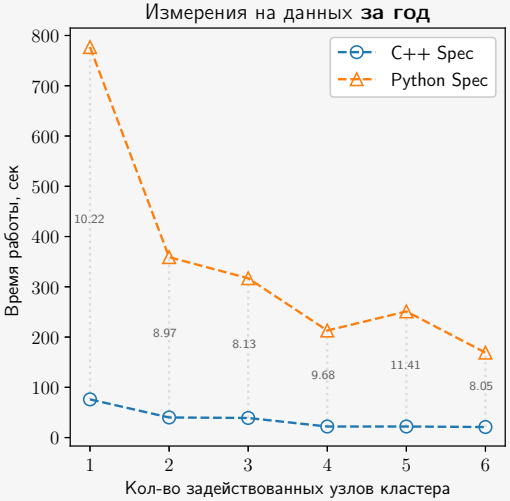


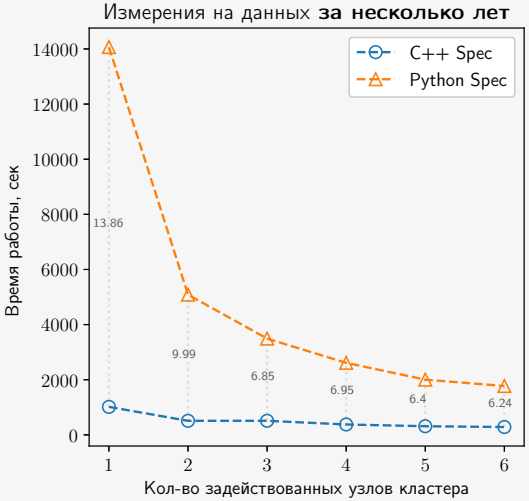
Dataset size: 512MB
Dataset size (uncompressed): 2 851MB
No. of wave rider buoys: 25
Time span: 10 years (2010–2019)
Total no. of spectra: 1 637 809

Data source: https://www.ndbc.noaa.gov/data/historical/

High-level interface

Python (NDBC spectrum preprocessing)

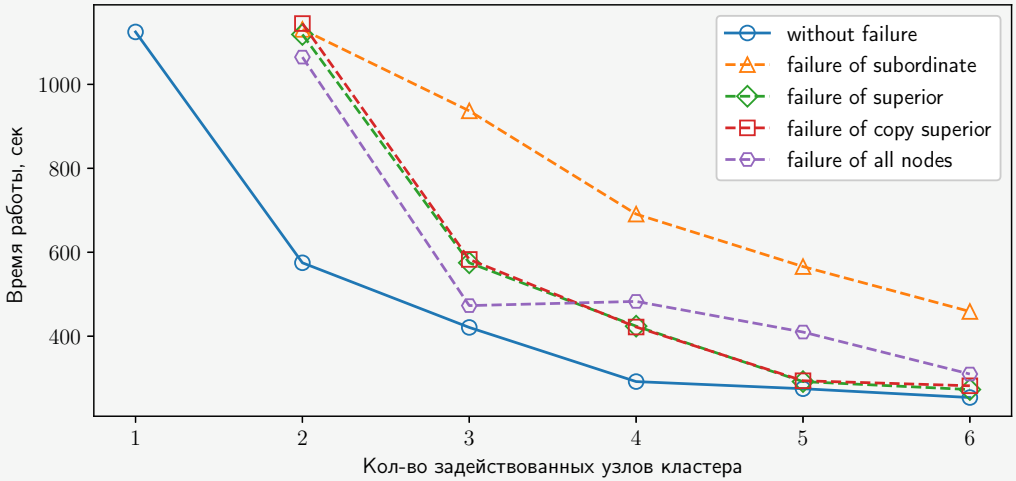




Dmitry Tereschenko *Fault-tolerant distributed computing in Python based on control flow objects*, Bachelor thesis, 2021 (to be published).

High-level interface

Python with fault tolerance (NDBC spectrum preprocessing)



Dmitry Tereschenko *Fault-tolerant distributed computing in Python based on control flow objects*, Bachelor thesis, 2021 (to be published).

High-level interface
Guile with automatic parallelism

The idea: evaluate arguments in parallel (one kernel for each argument).

```
(define (map proc lst) "Parallel map."
  (if (null? lst) lst
    (cons (proc (car lst)) (map proc (cdr lst)))))

(define (fold proc init lst) "Sequential fold."
  (if (null? lst) init
    (fold proc (proc (car lst) init) (cdr lst))))

(define (do-fold-pairwise proc lst)
  (if (null? lst) '()
    (if (null? (cdr lst)) lst
      (do-fold-pairwise proc
        (cons (proc (car lst) (car (cdr lst)))
              (do-fold-pairwise proc (cdr (cdr lst))))))))

(define (fold-pairwise proc lst) "Parallel pairwise fold."
  (car (do-fold-pairwise proc lst)))
```

High-level interface

Guile with automatic parallelism (synthetic benchmark)

The picture was removed because it was taken from the paper that will be published later this year. Please, contact the author directly.

I. Petriakov, I. Gankevich *Functional programming interface for parallel and distributed computing* ICCSA'21, 2021 (to be published).

Summary

Our goal was: “Write parallel and distributed applications the same way as we write sequential ones.”

- Kernels simplify writing distributed applications.
- Easy to implement in any language that supports threads.
- Cluster scheduler provides load balancing and fault tolerance.
- Additionally Guile/LISP interfaces provide automatic parallelism.
- Kernels can be used to re-implement schedulers for existing interfaces to get more performance (e.g. Apache Spark).

Contacts

Ivan Gankevich, associate professor

Email	i.gankevich@spbu.ru
Web site	https://igankevich.com/
Faculty home page	http://www.apmath.spbu.ru/en/staff/gankevich/index.html

Images:
- Python (NDBC spectrum preprocessing) © Dmitry Tereschenko
- Python with fault tolerance (NDBC spectrum preprocessing) © Dmitry Tereschenko