NEAR DATA PROCESSING (PIM) - MY VISION

• Capability to store (possibly permanent) data
• Shared & Distributed
• Capability to process data, e.g. Forking, Zeroing VM cloning, Checkpointing, Crypto, Hashing

3D memory + logic

memmove & memcpy: 5% cycles in Google’s datacenter [Kanev+ ISCA’15]
NEAR DATA PROCESSING (PIM) - MY VISION

- Capability to store (possibly permanent) data
- Shared & Distributed
- Capability to process data, e.g. Forking, Zeroing
  VM cloning, Checkpointing, Crypto, Hashing

PIM API

Key1=put(blob_x)
Key2=PIM(pagerank(key1))
blob_y=get(key_y)

memmove & memcpy: 5% cycles in Google’s datacenter [Kanev+ ISCA’15]
INMEMORY DATA SERVICE

M. Aldinucci and M. Torquati, “Accelerating apache farms through ad-HOC distributed scalable object repository,” in Proc. of 10th Intl. Euro-Par 2004 Parallel Processing, 2004,

Memcached, 2003
API: x=get(key),
key=put(x)

ADHOC, 2003
API: x=get(key),
key=put(x),
execute(key, f())
OUTLINE

• Programming models and data processing models (my own vision)
  • Message Passing vs Distributed Shared Memories and the PGAS model
  • Batch vs Streaming

• Fastflow
  • Intro
  • PiCo (Pipeline Composition): a C++ BigData analytics based on fastflow
  • GAM (Global Asynchronous Memory): a (modern) PGAS supporting streaming

• PGAS + Stream computation = Modern InMemory data service for DL/BDA
  • GAM+PiCo

• PGAS + Deep Learning = Deep Learning at scale
ABSTRACTION: PARALLEL PROGRAMMING MODEL (LOW-LEVEL)

• Message-Passing
  • Scalability and performance
  • Developer-based precise knowledge of code and overhead
  • Processes + communications (symmetric/collection, blocking/nonblocking)

• Shared-Memory
  • Productivity
  • Global and uniform vision of data layout
  • Threads + synchronisations mechanisms (mutex, atomics, transactions, …)
Already mature 20 years ago, now spoiled.
ABSTRACTION: DATA PROCESSING MODELS (DATA-CENTRIC VIEW)

- **Batch**
  - finite input datasets ➔ output dataset
  - Random access
  - Lists, collections, bags...

- **Stream**
  - unbound sequence of data input ➔ unbound sequence of data input
  - Sequential access
  - Streams
PGAS PROGRAMMING MODEL (DSM EVOLVED)

- A set of processor, each with own local memory
- Part managed as private, part as shared
  - Sharing implemented HW or SW
- Explicitly NUMA
  - Each location has an affinity with a processor
  - Model differentiates between local and remote data partitions
- Explicitly partitioned
  - Collective synchronisations, i.e. barriers and fences
COROUTINES: SEMANTICS IN SEARCH OF A SYNTAX

by N. Douglas McIlroy

Oxford University and
Bell Telephone Laboratories, Incorporated.

ABSTRACT: Unlike subroutines, coroutines may be connected, and reconnected, in nonhierarchical arrangements. Coroutines are particularly useful for generating and processing data streams. Semantics for coroutines are developed and examples are given.
STREAMING: CORE PARADIGM

control actors

operator

T-gate

F-gate

Boolean actors

or

decider

merge

and

Figure 2. Node types for data flow programs.
Windowed Stream Processing (Data-Centric View)

- Windows approximate infinite stream history
  - tuple significance is often time-decaying
  - only the most recent tuples are kept
- Different windowing policies
  - Sliding windows: window size + sliding factor
  - Session windows [Apache Flink, Apache Beam…]

An expensive operation. Useful for BDA, maybe useful for ML
WINDOW-FARMING (WF)

Common implementation.
More parallelism.
MESSAGE PASSING + SHARED MEMORY = FASTFLOW

HEADER-ONLY “VANILLA” C++17 SOURCE CODE + TEMPLATE METAPROGRAMMING

- **Toreador** (EC-RIA, H2020, ICT-16-2015 big data): Trustworthy model-aware Analytics Data platform (2016, 36 months, total cost 6.5M €)
- **REPARA** (EC-STREP, 7th FP): Reengineering and Enabling Performance And power Of Applications (2013, 36 months, total cost 3.5M €)
- **ParaPhrase** (EC-STREP, 7th FP): Parallel Patterns for Adaptive Heterogeneous Multicore Systems (2011, 42 months, total cost 4.2M €)
- **IBM Research** 3 faculty awards 2015 (50K $)
- **Noesis Solutions**: Machine learning for engineering 2015 (75K €)
- **NVidia Corp**: CUDA Research Center at University of Torino 2013

https://github.com/fastflow/fastflow
int n=300;
// parallel_execution_tbb p{}, f{);
parallel_execution_ff p{}, f{);
// sequential_execution p{}, f{);
// parallel_execution_omp p{}, f{);

Pipeline(p,
    // Pipeline stage S0
    [&](){
        std::vector<int> v(100);
        // C++11 business code
        return optional<std::vector<int>>(v);
    },
    // Pipeline stage S1
    Farm(f,
        [&](std::vector<int> v) {
            std::vector<int> acumm( v.size() );
            // C++11 business code
            return acumm;
        }),
    // Pipeline stage S2
    [&]( std::vector<int>  acc ) {
        double acumm = 0;
        for ( int i = 0; i < acc.size(); i++ )
            acumm += acc[ i ];
        return acumm;
    },
    // Pipeline stage S3
    [&]( double v ) {
        // C++11 business code
    });

// Pipeline stage S2
[&]( std::vector<int> acc ) {
    double acumm = 0;
    for ( int i = 0; i < acc.size(); i++ )
        acumm += acc[ i ];
    return acumm;
},

// Pipeline stage S3
[&]( double v ) {
    // C++11 business code
};
OPENMP-LIKE LOOP PARALLELISM CODING STYLE

Sequential

```cpp
... const int Limit = 4;
bool allBlack = true;
if (restart) break;
if (abort) return;

for (int y = -halfHeight; y < halfHeight; ++y) {
    uint *scanLine =
        reinterpret_cast<uint *>(image.scanLine(y + halfHeight));
    double ay = centerY + (y * scaleFactor);
...

private:
...
```

FastFlow

```cpp
... const int Limit = 4;
bool allBlack = true;
if (restart) break;
if (abort) return;

pf_det.parallel_for(-halfHeight, halfHeight, 1, halfHeight,
    [&](const long y) {
        uint *scanLine =
            reinterpret_cast<uint *>(image.scanLine(y + halfHeight));
        double ay = centerY + (y * scaleFactor);
    ...

private:
    ParallelFor pf_det;
...
```
**Heterogeneous Computing Style**

**Iterative Stencil-Reduce (Multiple GPU)**

1. while (cond) {
   2. before (...) // [H] initialisation, possibly in parallel on CPU cores
   3. prepare (...) // [H+D] swap I/O buffers, set kernel args, D2D—sync overlays
   4. stencil<SUM_kernel, MF_kernel> (input, env) // [D] stencil and partial reduce
   5. reduce op data // [H] final reduction
   6. after (...) // [H] iteration finalisation, possibly in parallel on CPU cores
}

8. read(output) // [H+D] D2H—copy output

**Fig. 2** Loop-of-stencil-reduce pattern general schema.

---

A master-worker task executor (e.g. dataflow)

---

#include <vector>
#include <iostream>
#include <ff/farm.hpp>
#include <ff/pipeline.hpp>
#include <ff/node.hpp>

using namespace ff;

/*
 * NOTE: this is a multi-input node ff_minode !
*/

class MU: public ff_minode {
public:
    MU(int numtasks):
        numtasks(numtasks), k(0) {}

    void* svc(void* task) {
        if (task == NULL) {
            printf("MU starting producing tasks\n");
            for(long i=1;i<=numtasks;++i)
                ff_send_out((void*)i);
            return GO_ON;
        }

        long t = (long)task;
        if (--t > 0) ff_send_out((void*)t);
        else if (++k == numtasks) return NULL;
        return GO_ON;
    }

private:
    long numtasks;
    long k;
};

struct Scheduler: public ff_node {
    void* svc(void* task) {
        return task;
    }
};

struct FU: public ff_node {
    void* svc(void* task) {
        printf("FU (%ld) got one task\n", get_my_id());
        return task;
    }
};

int main(int argc, char* argv[]) {
    int nw=3;
    int numtasks=1000;
    if (argc>1) {
        if (argc < 3) {
            std::cerr << "use:\n" << " " << argv[0] << " numworkers ntasks\n";
            return -1;
        }
        nw=atoi(argv[1]);
        numtasks=atoi(argv[2]);
    }

    ff_pipeline pipe;
    ff_farm<> farm;

    for(int i=0;i<nw;++i)
        w.push_back(new FU);
    farm.add_emitter(new Scheduler);
    farm.add_workers(w);
    pipe.add_stage(new MU(numtasks));
    pipe.add_stage(&farm);

    /* this is needed to allow the creation of output buffer in the farm workers */
    farm.remove_collector();

    pipe.wrap_around();
    pipe.run_and_wait_end();
    return 0;
}
EXPRESSIVITY (PARSEC)

D. De Sensi et al. Bringing Parallel Patterns out of the Corner: the P3ARSEC Benchmark Suite. ACM TACO, 14(4), 2017, ACM.
**Task Graph vs Network of Executors**

- **Graph of tasks**
  - Dataflow — typically DAG
  - Each node is a task
  - E.g. a C++ object
  - Problems: firing, scheduling, etc.

- **Network of executors**
  - "Controlflow" — typically cyclic graph
  - E.g. threads or processes
  - Problems: pinning, mapping, pooling, etc.
SWSR QUEUES + MEDIATORS

FF bound shmem FIFO channel
Single-Producer-Single-Consumer
lock-free fence-free queue

FF unbound shmem FIFO channel
Single-Producer-Single-Consumer
lock-free fence-free queue

FF (lock-free) distributed memory channel
RDMA or TCP

Really any accelerator: FPGA, Knight Landing, Tilera, …

This is a thread.
Not a task!

network symmetric or asymmetric (scatter, gather, etc)
Generate the network
True data dependencies moves across arrows

Composing via mediator guarantee correctness (data races & deadlock freedom)
**TEST: MASTER-WORKER (INTEL E7)**

- Intel Xeon CPU E7-4820 @ 2.00GHz
- Task sizeof(long) messages, task execution time 1 us, 10 us
- All queues list-based unbound
  - 1 FastFlow (ff)
  - 2 Yang & Mellor-Crummey PPoPP 2016 (wf)
  - 3 Michael & Scott (ms)
**TEST: MASTER-WORKER (INTEL E7)**

- Intel Xeon CPU E7- 4820 @ 2.00GHz
- Task sizeof(long) messages, task execution time 1 us, 10 us
- All queues list-based unbound
  - ❶ FastFlow (ff)
  - ❷ Yang & Mellor-Crummey PPoPP 2016 (wf)
  - ❸ Michael & Scott (ms)

---

**Methodology!**

- **Master-Worker**
  - Ideal
  - ms, ex. time = 1 us
  - ms, ex. time = 10 us
  - wf, ex. time = 1 us
  - wf, ex. time = 10 us
  - ff, task ex. time = 1 us
  - ff, task ex. time = 10 us

---

**Better**

- Number of workers
- Speedup
ATOMICS: SOME CONSIDERATIONS

• Support **global** synchronisations. Have a cost, worth to pay if you need them
  - Not really needed for master-worker, data parallelism, embarrassingly parallel, pipeline, and other **unexpected** cases …

• Support a **on-demand** approach. Scheduling policy is embedded in the protocol, hardly programmable
  - Shuffle (e.g. for MapReduce): data movement path is fixed and known at run-time
  - Affinity problems: parallel memory allocation, data-dependent scheduling
  - Memory subsystem and cache coherency can be bottlenecks
PERF (PARSEC)

D. De Sensi et al. Bringing Parallel Patterns out of the Corner: the P3ARSEC Benchmark Suite. ACM TACO, 14(4), 2017, ACM.
PROGRAMMING MODEL: SYNCHRONISATIONS HAPPEN BY WAY OF P2P DATA DEPENDENCIES (THUS NO ATOMICS ARE NEEDED)

Synchronisations are in a message-passing style, but designers are not forced to think in a distributed way.

No copies are needed, the memory fences are but asynchrony helps.

Shared-memory cache-coherent or non-coherent multicore
**Programming Model: Synchronisations Happen by Way of P2P Data Dependencies (Thus No Atomics Are Needed)**

Shared-memory cache-coherent or non-coherent multicore

Distributed GAM
THIS YEARS NOVELTIES

- Dynamically blocking-nonblocking behaviour
- Optimisation (graph rewriting)
AUTOMATIC BLOCKING-AND-NONBLOCKING QUEUE
(SORT OF USER-SPACE FUTEX)

FIGURE 11 Performance and power consumption comparison of blocking, nonblocking and automatic concurrency control strategies for the Protocol identification application.
FIGURE 12 Performance and power consumption comparison of blocking, nonblocking and automatic concurrency control strategies for the Malware detection application.
**SKELETON COMBINE (DEEPFARM)**

Data split (possibly with affinity) moving pointers only

Independent work

\[
\text{pipeline} (\text{map}(f), \text{reduce}(\oplus))
\]

**Skeleton Combine (DeepFarm)**

```
pipeline(map(f), reduce(⊕))  pipeline(map(f), reduce_by_key(⊕))
```
STREAMING: THE FASTFLOW ECOSYSTEM

• FastFlow: efficient streaming and parallel programming for C++

• GAM: Global Asynchronous Memory
  • A PGAS on top of FastFlow, designed for streaming

• PiCo: Pipeline Composition
  • A (fully functional) DSL for programming data transformation
  • Fully equivalent to Spark (but fully C++11)

• GAM+PiCo = Near Data Processing service
  • A “MemCached” with a “rich API”

GLOBAL ASYNCHRONOUS MEMORY
A STREAM-ORIENTED PGAS (BASED ON FF)

• From MPI Style:
  communicate pointers
  (a.k.a., capabilities)

• From DSM Style:
  shared address space

→ Capability = both data reference
  and synchronisation token
PUBLIC POINTERS

• Read-only (single assignment)
• Cacheable
• Can be copied

PRIVATE POINTERS

• Exclusive read-write

• Not cacheable

• Can be moved

GAM MEMORY MODEL

- (Trivial) Sequential Consistency
- **Avoiding** consistency issues (vs solving as in DSM/PGAS)
- SWMR cache-coherence invariant:
  - public $\rightarrow$ (NW)MR
  - private $\rightarrow$ SWSR
SMART GLOBAL POINTERS

• Rooted in modern C++
  • Intentional programming:
    public → shared, private → unique

• Automatic Memory Management — the C++ way
  • Smartness = memory-reference lifetime binding
  • No memory leaks, no dangling pointers
  • No garbage collection (vs Java & friends)
PUBLIC POINTERS

public_ptr(T * const, Deleter);
public_ptr<T> make_public(Args&&...);

//copy constructor/assignment...
//move constructor/assignment...

public_ptr(private_ptr<T> &&);
public_ptr& operator=(private_ptr<T> &&);

std::shared_ptr<T> local();

void push(executor_id to);
public_ptr<T> pull_public(const exec_id from);
public_ptr<T> pull_public();

enables plain C++ code
PRIVATE POINTERS

private_ptr(T * const, Deleter);
private_ptr<T> make_private(Args&&...);

//move constructor/assignment...

//NO copy constructor/assignment

gam_unique_ptr<T> local(); //unique_ptr + custom deleter

void push(executor_id to);
private_ptr<T> pull_private(const exec_id from);
private_ptr<T> pull_private();
SMARTNESS FOR PUBLIC POINTERS

- Distributed **reference counting** protocol
- Creation/copy/push trigger +1, destruction triggers -1
- C++ shared pointers: atomic-based reference counting
SMARTNESS FOR PRIVATE POINTERS

- Distributed **memory releasing** protocol
- Destruction triggers releasing
- C++ unique pointers: destruction-triggered release
- Inherently simpler than public pointers (as unique vs shared)
KP - Inherently Efficient

- Disjoint IN(A)/IN(B)
  - each tuple accessed “exclusively”

- GAM implementation
  - private pointers - exclusive capabilities
  - 1 RMA access per tuple
WF - Inherently Complex

- $\text{IN}(A) = \text{IN}(B) = \text{IN}$
  - each tuple accessed "by-any"

- GAM implementation
  - public pointers - read-only replicas
  - multiple RMA accesses per tuple
GAM ADVANTAGES

• Passing capabilities versus data
  • efficient worker-side windowing
  • extreme case: static dispatching not viable
PiCo — C++ API

- Algebra of **Pipelines** and Operators
- Unified model for batch and stream processing
- Clear functional semantics
- RISC

PiCo "Pipelines"


Figure 1: Graphical representation of PiCo Pipelines.
PiCo — C++ DSL (Operators)

- Source/Sink
  - From/to file, network…
- Map/Reduce
  - and Binary variants
- Modifiers
  - Windowing
  - Partitioning (e.g., by-key)

<table>
<thead>
<tr>
<th>Operator family</th>
<th>unary</th>
<th>element-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>combine</td>
<td></td>
<td>collective</td>
</tr>
<tr>
<td>b-map</td>
<td>binary</td>
<td>pair-wise</td>
</tr>
<tr>
<td>b-combine</td>
<td>binary</td>
<td>collective</td>
</tr>
<tr>
<td>emit</td>
<td></td>
<td>produce-only</td>
</tr>
<tr>
<td>collect</td>
<td></td>
<td>consume-only</td>
</tr>
</tbody>
</table>
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(tokenizer)
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);

explicit data collections
WORD COUNT IN PiCo

Pipe CountWords;
CountWords
  .add(FlatMap<string,string>(tokenizer))
  .add(Map<string,KV>[](string in){return KV(in,1);}))
  .add(PReduce<KV>[](KV a, KV b){return a + b;})a
Read [foo-pages] → Map → ReduceByKey → 
JoinByKey + FlatMap → Empty → Map → Write [bar]
PERFORMANCES — STREAMING WORDCOUNT

- Fast
  - C++ little runtime overhead

- Scalable
  - Streaming on top of FastFlow (state-of-the-art performance)

- Low memory footprint
  - Better memory management than Java

<table>
<thead>
<tr>
<th></th>
<th>Flink</th>
<th>Spark</th>
<th>PiCo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. Exec. Time</td>
<td>24.78 s</td>
<td>42.22 s</td>
<td>7.35 s</td>
</tr>
<tr>
<td>Relative Speedup</td>
<td>9.21</td>
<td>2.24</td>
<td>14.87</td>
</tr>
<tr>
<td>CPU %</td>
<td>14.31</td>
<td>10.23</td>
<td>38.85</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>4.88 GB</td>
<td>3.17 GB</td>
<td>315 MB</td>
</tr>
</tbody>
</table>
AND IT IS SCALABLE
GAM + PiCo = Distributed NDP

- GAM is a (distributed) Memory
- PiCo is a language to program data transformation
- PiCo can be used to “program” memory operations
  - simple operations (as in Onur’s talk)
  - complex operations, e.g. classification, regression, segmentation … (as in Ernesto’s talk)
INMEMORY DATA SERVICE

M. Aldinucci and M. Torquati, “Accelerating apache farms through ad-HOC distributed scalable object repository,” in Proc. of 10th Intl. Euro-Par 2004 Parallel Processing, 2004,
INMEMORY DATA SERVICE

M. Aldinucci and M. Torquati, “Accelerating apache farms through ad-HOC distributed scalable object repository,” in Proc. of 10th Intl. Euro-Par 2004 Parallel Processing, 2004,

**Figure 16.2 memcached Architecture Overview**

Memcached, 2003

API: \( x = \text{get}(\text{key}), \quad \text{key} = \text{put}(x) \)

ADHOC, 2003

API: \( x = \text{get}(\text{key}), \quad \text{key} = \text{put}(x), \quad \text{execute}(\text{key}, f) \)
Deep Learning

Key concepts

- Training is an optimization process in a hugely high-dimensional space.
- The most efficient algorithm is Stochastic Gradient Descent (SGD).
- Hyper-parameters tuning is key to achieve any result. **Mini-batch size** is especially relevant for this work.

Mini-batches, introduced to improve convergence properties over plain SGD, are incidentally the most trivial way to achieve parallelism in Deep Neural Network (DNN) training.
Distributed Deep Learning taxonomy

Main categories from DL literature

- Data Parallelism
- Model Parallelism
- Domain Parallelism

Research scope

These three approaches represent orthogonal directions of improvement, hence they can be treated as layer of optimization on top of each other in any order.

The scope of this work focuses on Data Parallelism, as it is considered the most prone to large-scale applications.
Distributed Deep Learning taxonomy

Data parallelism

Dataset

Partitions

Gradients

Gradients
Distributed Deep Learning taxonomy

Data parallelism
- Synchronous SGD
- Asynchronous SGD
  - Centralized
  - De-centralized

Asynchronous SGD
- Each worker trains on a partition of the dataset
- Centralized: gradients are accumulated and applied on a central parameter server (PS)
- De-centralized: gradients are broadcast and enqueued by each worker, who applies them when ready.
- This approach either requires multiple expensive broadcast communications, or introduces a bottleneck (PS)
We argue that the distinction between synchronous and asynchronous approaches is not really useful.

**Issues**

- The synchronous approach is not distinguishable from a large mini-batch, so it is not always suitable.
- Asynchronous, centralized approach artificially introduces a bottleneck that delays gradient updates. While can be an advantage for convergence, scalability is at risk even with careful engineering.
- De-centralization removes the bottleneck, but introduces broadcasts. Moreover, pushing the engineering to the limit of bandwidth and latency, it is *not distinguishable from the synchronous version.*
Proposed approach

Requirements

- Scalability up to very large scale
- Different convergence dynamic w.r.t. very large mini-batch approximations

Ideas

- Drop completely the worker consistency
- Get inspiration from large-scale simulations
- Keep communications local

We advocate an approach based on nearest-neighbour communications, where gradients are broadcast by a worker only to a pre-defined set of neighbours.
Proposed approach

Nearest-neighbours communications
Proposed approach

Nearest-neighbours communications

- Gradients are shared asynchronously with nearest neighbours
- Topology can be defined to best match the available resources (e.g. 2D/3D torus)
- Workers are initialised with the same weights
- Gradients are not propagated directly to all workers, but information reaches far away nodes mediated with intermediate "hops"
- Workers consistency is not guaranteed, not even eventually
  - However, under certain assumptions (rotation of data partitions), it is possible to prove that the workers are optimizing the same target function
- This approach does not approximate a large mini-batch under any assumption
- Lack of global communications enables scalability
Thanks!

Claudia Misale
IBM TJ Watson

Maurizio Drocco
Pacific Northwest National Laboratory (PNNL)

Paolo Viviani
Noesis Solutions