

HPCC 2014

**The 16th IEEE International Conference on
High Performance Computing and Communications
Paris, France, August 20-22, 2014**



Parallel patterns, data-centric concurrency, and heterogeneous computing

Marco Aldinucci, University of Torino, Computer Science Department



Outline

- ❖ Parallel patterns and structured parallel programming
 - ❖ On the hierarchy of abstractions for parallel programming
- ❖ Data-centric concurrency
 - ❖ FastFlow: a data-centric run-time support for heterogenous platforms
- ❖ Applications
 - ❖ With FastFlow and other frameworks
- ❖ ... and lot of cars

MPI

is like a car, you can drive it as you like



MPI, threads, HW-CAS spinlocks, ...
are like cars, you can drive it as you like



```

#include <stdio.h>
#include "mpi.h"
#define MAXPROC 8      /* Max number of procsses */
#define NAMELEN 80     /* Max length of machine name */
#define LENGTH 24      /* Length of send buffer is divisible by 2, 4, 6 and 8 */

main(int argc, char* argv[]) {
    int i, j, np, me;
    const int nametag = 42; /* Tag value for sending name */
    const int datatag = 43; /* Tag value for sending data */
    const int root = 0; /* Root process in scatter */
    MPI_Status status; /* Status object for receive */

    char myname[NAMELEN]; /* Local host name string */
    char hostname[MAXPROC][NAMELEN]; /* Received host names */

    int x[LENGTH]; /* Send buffer */
    int y[LENGTH]; /* Receive buffer */

    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get nr of processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* Get own identifier */

    gethostname(&myname, NAMELEN); /* Get host name */

    if (me == 0) { /* Process 0 does this */

        /* Initialize the array x with values 0 .. LENGTH-1 */
        for (i=0; i<LENGTH; i++) {
            x[i] = i;
        }

        /* Check that we have an even number of processes and at most MAXPROC */
        if (np>MAXPROC || np%2 != 0) {
            printf("You have to use an even np (at most %d)\n", MAXPROC);
            MPI_Finalize();
            exit(0);
        }

        printf("P %d on host %s is distributing array x to all %d processes\n\n",
            me, myname, np);

        /* Scatter the array x to all proceses, place it in y */
        MPI_Scatter(&x, LENGTH/np, MPI_INT, &y, LENGTH/np, MPI_INT, root, \
            MPI_COMM_WORLD);

        /* Print out own portion of the scattered array */
        printf("Process %d on host %s has elements", me, myname);

```

```

        for (i=0; i<LENGTH/np; i++) {
            printf(" %d", y[i]);
        }
        printf("\n");

        /* Receive messages with hostname and the scattered data */
        /* from all other processes */
        for (i=1; i<np; i++) {
            MPI_Recv (&hostname[i], NAMELEN, MPI_CHAR, i, nametag,
                MPI_COMM_WORLD, &status);
            MPI_Recv (&y, LENGTH/np, MPI_INT, i, datatag, MPI_COMM_WORLD, &status);
            printf("Process %d on host %s has elements", i, hostname[i]);
            for (j=0; j<LENGTH/np; j++) {
                printf(" %d", y[j]);
            }
            printf("\n");
        }

        printf("Ready\n");

    } else { /* all other processes do this */

        /* Check sanity of the user */
        if (np>MAXPROC || np%2 != 0) {
            MPI_Finalize();
            exit(0);
        }

        /* Receive the scattered array from process 0, place it in array y */
        MPI_Scatter(&x, LENGTH/np, MPI_INT, &y, LENGTH/np, MPI_INT, root, \
            MPI_COMM_WORLD);
        /* Send own name back to process 0 */
        MPI_Send (&myname, NAMELEN, MPI_CHAR, 0, nametag, MPI_COMM_WORLD);
        /* Send the received array back to process 0 */
        MPI_Send (&y, LENGTH/np, MPI_INT, 0, datatag, MPI_COMM_WORLD);

    }

    MPI_Finalize();
    exit(0);
}

```

```
bool push(void *const data) {
    unsigned long pw, seq;
    element_t * node;
    unsigned long bk = BACKOFF_MIN;
    do {
        pw = pwrite.load(std::memory_order_relaxed);
        node = &buf[pw & mask];
        seq = node->seq.load(std::memory_order_acquire);
        if (pw == seq) { // CAS
            if (pwrite.compare_exchange_weak(pw, pw+1, std::memory_order_relaxed))
                break;
            for(volatile unsigned i=0;i<bk;++i) ;
            bk <<= 1;
            bk &= BACKOFF_MAX;
        } else
            if (pw > seq) return false; // queue full
    } while(1);
    node->data = data;
    node->seq.store(seq+1, std::memory_order_release);
    return true;
}
```

CAS & memory consistency

```

__global__ void kernel1(int *lock, int *x) {
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    do {
        //nop
    } while(atomicCAS(lock, 0, 1));
    //critical region
    *lock = 0;
}

```

Deadlock

```

__global__ void kernel2(int *var, int *x) {
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int oldval, newval = 0;
    do {
        oldval = *var; //read shared var
        //critical region
        newval = oldval + 1; // f(oldval)
    } while(atomicCAS(var, oldval, newval) != oldval);
    x[idx] = newval;
}

```

Works?

- ❖ Deadlock on GPGPU (unless nvcc -G).
- ❖ A different execution model. Impossible to make any assumptions about scheduling
- ❖ Data-dependency can be managed only via lock-free algorithms

Message-passing (e.g. MPI)

Shared-memory (e.g. threads + mutex/CAS)

- ❖ (Can be) Efficient
- ❖ Available in almost all platforms
 - ❖ Often the only access to net, e.g. HRLS Cray XE6 Hermit: 32 x 3552 nodes = 113 664 cores
 - ❖ De-facto standards, used for decades

- ❖ Parallel primitives fully inter-waived with business code
- ❖ How compose “computing phases”, “SW modules” ...
 - ❖ Parallel behaviour and data layout not explicit in the code. Primitives often do not compose.
- ❖ Often to be coupled with shared-memory for intra-node
 - ❖ e.g. MPI & Pthreads, MPI & OpenMP, PGAS-MPI & OpenCL, ...

Lot of
freedom

I ♥ MPI

I ♥ threads

No whiners

```
        (PCP_NO_INC(pbOut).X), &(PCP_
INC(pbIn);
INC(pbOut);
if IsFULL(pbOut)
{
    if (IS_ON_SM(pbOut))
        MPI_Waitall(pbOut->ndest, pbOut
    else
        SET_ON_SM(pbOut);
        SWAP(pbOut);
        SetLAST(pbOut, 0);
        for(skie_i=0; skie_i<(pbOut->ndest
            MPI_Isend(pbOut->c, sizeof(stre
                MPI_BYTE, pbOut->dest
                    &(pbOut->req_Out[skie
        RESET(pbOut);
}
if (IsEMPTY(pbIn))
{
    MPI_Wait(&(req_task_r), &(sta_tas
    MPI_Wait(&(pbIn->req_In), &(pbIn-
    pbIn->nexttag=pbIn->sta_In.MPI_T
    SWAP(pbIn);
    if (!IsLAST(pbIn))
    {
        MPI_Isend(&task_r, sizeof(tas
            _S_REQUEST_TAG, MPI
        MPI_Irecv(pbIn->c, sizeof(str
            pbIn->mitt, MPI_ANY
    }
    RESET(pbIn);
}
}
/* EDS Management */
pbOut->acttag=pbIn->nexttag;
WriteMARK(pbOut, MARK(pbIn));
if (IS_ON_SM(pbOut))
    MPI_Waitall(pbOut->ndest, pbOut->req_Out
SWAP(pbOut);
SetLAST(pbOut, 1);
for(skie_i=0; skie_i<(pbOut->ndest); skie_
    MPI_Isend(pbOut->c, sizeof(strean_type_
        MPI_BYTE, pbOut->dest[skie_i]
            &(pbOut->req_Out[skie_i]));
    MPI_Waitall(pbOut->ndest, pbOut->req_Out,
}
```

Design and validation of the MPI backend for the *SkIE* compiler

Marco Aldinucci

Dec 18, 1997

Quadrics Supercomputing World (QSW)
Technical Report

Unstructured programming considered harmful

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CC Categories: 4.22, 4.23, 4.34

Intro:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming

dynamic program is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual inlines, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (do, while, repeat, or repeat until). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well accustomed to certain non-inducted cases in the

Send-Receive Considered Harmful: Myths and Realities of Message Passing

SERGEI GORLATCH

Universität Münster

During the software crisis of the 1960s, Dijkstra's famous thesis "goto considered harmful" paved the way for structured programming. This short communication suggests that many current difficulties of parallel programming based on message passing are caused by poorly structured communication, which is a consequence of using low-level send/receive primitives. We argue that, like goto in sequential programs, send/receive should be avoided as far as possible and replaced by collective operations in the setting of message passing. We dispute some widely held opinions about the apparent superiority of pairwise communication over collective communication and present substantial theoretical and empirical evidence to the contrary in the context of MPI (Message Passing Interface).

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming

General Terms: Algorithms, Languages, Performance, Experimentation, Measurement

Additional Key Words and Phrases: Programming methodology, Message Passing Interface (MPI)

Car

Val D'Orcia, Tuscany, Italy



Designer utopia



False sharing



Irregular data

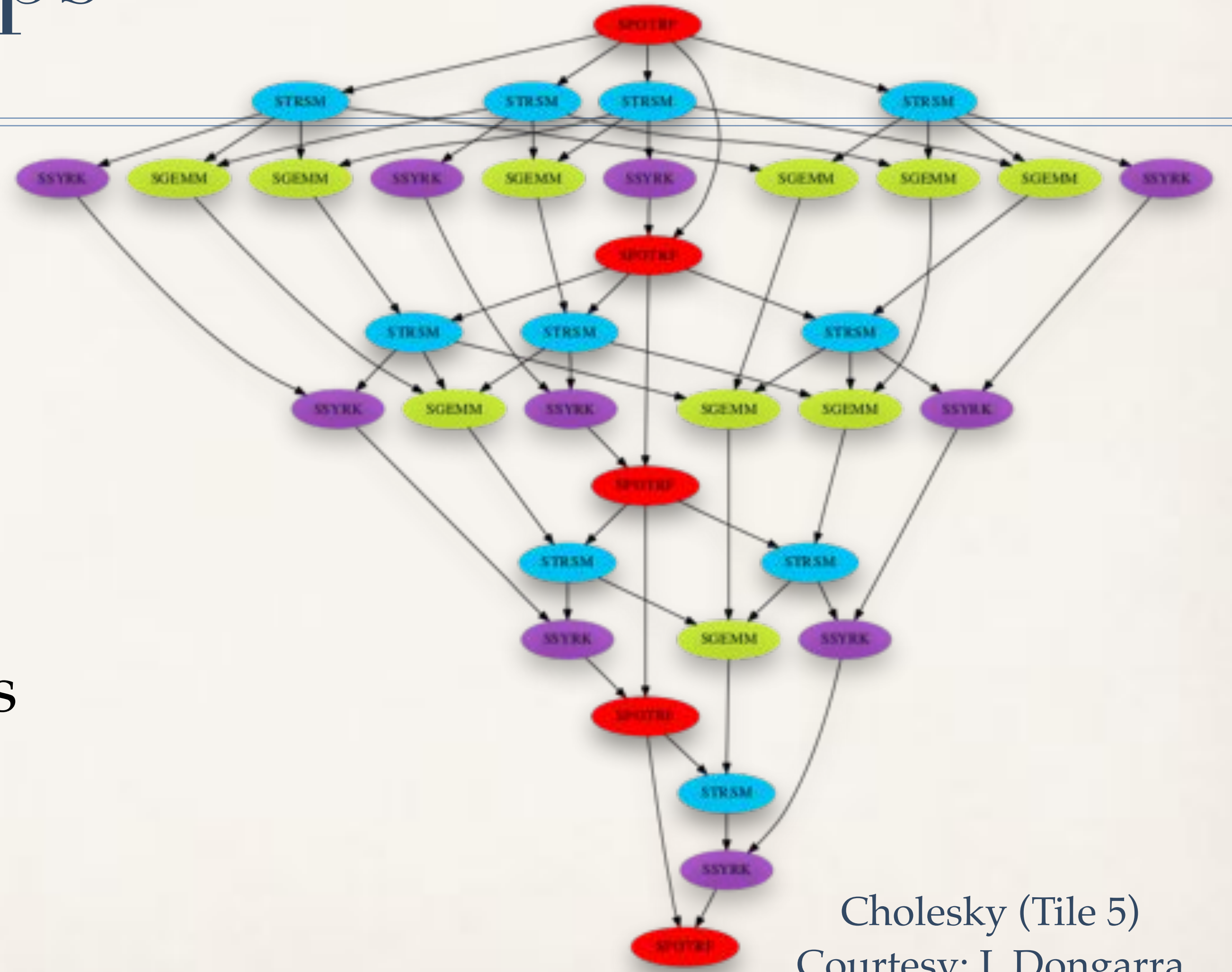


Hot spots

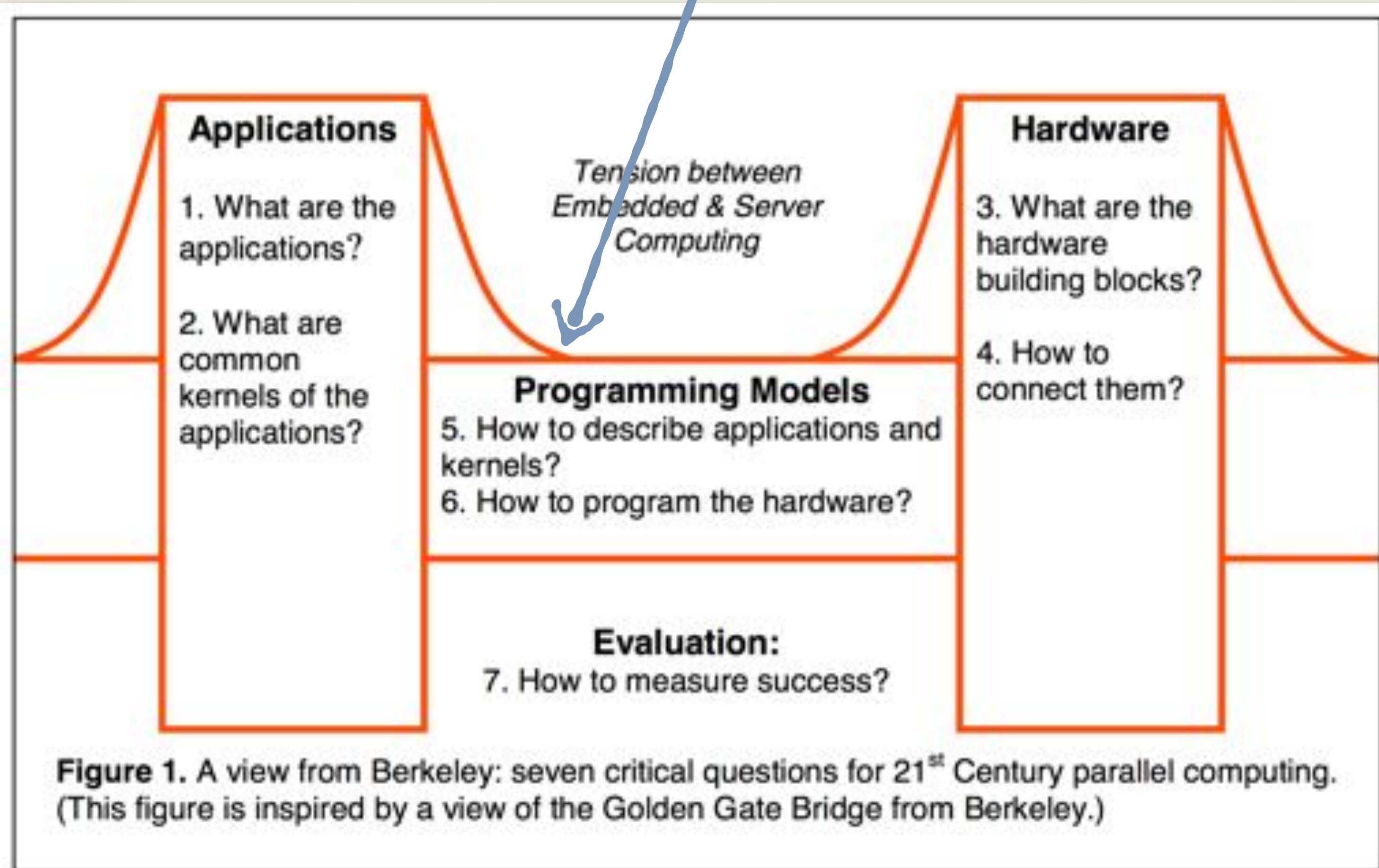
REALITY

Loop parallelism helps

- ❖ OpenMP is great example
 - ❖ It simply works well on loops
- ❖ Not really useful for all problems
 - ❖ Tasks, Graphs, ...



Programmings model is paramount



Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers.

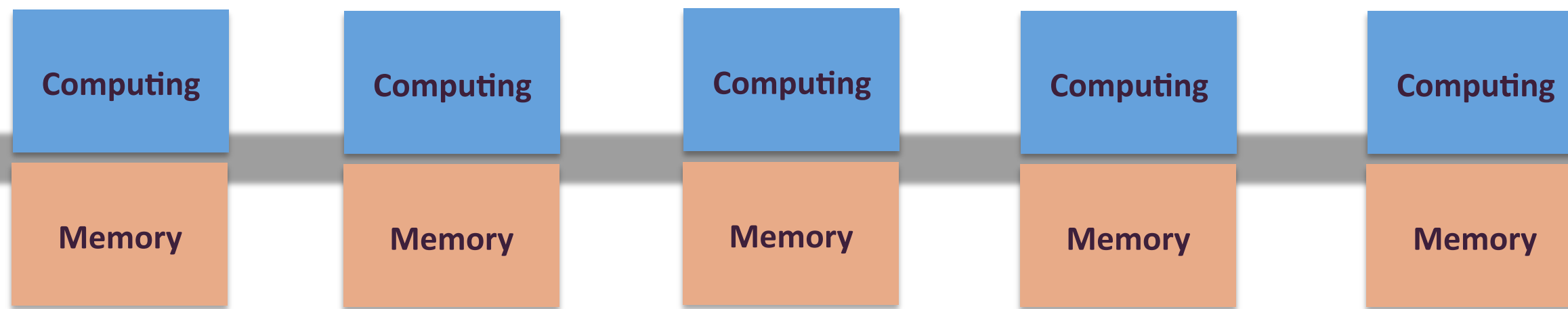
BY KRSTE ASANOVIC, RASTISLAV BODIK, JAMES DEMMEL, TONY KEAVENY, KURT KEUTZER, JOHN KUBIATOWICZ, NELSON MORGAN, DAVID PATTERSON, KOUSHIK SEN, JOHN WAWRZYNEK, DAVID WESSEL, AND KATHERINE YELICK

A View of the Parallel Computing Landscape

tech
form
plic
that
pow
as a
sequ
cont
infe
pow
issu
exec
and
perf
quer
T
hit t
diss
char
proc
tion
con
two
was
ceed
in 2
toda
2007
A
arch

And it is paramount also at the large scale

The Abstract Machine Model & Execution Model



- A computer language is not a programming model
 - “C++ is not scalable to exascale”
- A communication library is not a programming model
 - “MPI won’t scale to exascale”
- A unique application inventory response...
 - Should we be talking “Execution Model”?
- What is a programming model?

Thought: real issue is mapping science problem to execution model and run-time system



- ❖ Coarse grain concurrency is nearly exhausted
- ❖ Often, it is not about FLOPS, it is about data movement
- ❖ Programming systems should be designed to support fast data movement and enforce locality
- ❖ Variable coherency & inter-socket messaging

Excerpts

- ❖ MPI, the current dominant programming model for parallel scientific programming, **forces coders to be aware of the exact mapping of computational tasks to processors.** This style has been recognised for years to increase the cognitive load on programmers, and has persisted primarily because it is expressive and delivers the best performance. [Snir et al 1998]
[Gursoy and Kale 2004]
- ❖ Because we anticipate a massive increase in exploitable concurrency, we believe that this model will break down in the near future, **as programmers have to explicitly deal with decomposing data, mapping tasks, and performing synchronisation over thousands of processing elements.** [Asanovic et al 2006]

Abstraction (D. Reed)

“To date, attempts to develop higher level programming abstractions, tools and environments for HPC have largely failed.

There are many reasons for this failure, but I believe many are rooted in our **excessive focus on hardware performance measures.**

By definition, the *raison d'être* for high-performance computing is high performance, but FLOPS need not be the only measure.

Human productivity, total cost and time to solution are equally, if not more important.”

BLOG@CACM

High-Performance Computing: Where

By Daniel Reed

May 7, 2009

[Comments](#)

VIEW AS:  SHARE:     



You are a young entrepreneur with a “can’t miss” idea for the next great social networking service, one so hip, so cool and so awesome that it will spread like a virus, while venture capital firms line up to beg for a piece of the action. It’s just a simple matter of programming to turn your idea into code that scales seamlessly across tens of thousands of data center servers.

You know the drill. Fire up the *coffee percolator*, break out your copy of *Kernighan and Ritchie*, grab your cheat sheet of TCP/IP functions and parameters, slam a cassette into the tape player and start programming. Oh, wait; that’s so 70s and 80s! In the hierarchy of abstractions, it’s only slightly above toggling absolute binary into the front panel of the

machine.

In the web service world, we have moved beyond these low level tools. Over the past twenty years, we have built and embraced a suite of powerful libraries, scripting languages, software services and tools that allow developers to create complex software systems, while hiding the low level attributes of networks and computer systems. We focus on composition, abstraction, rapid deployment, software scaling and human productivity.

Meanwhile, in the world of high-performance computing, message passing has remained the programming paradigm of choice for over twenty years. The durable *Message Passing Interface (MPI)* standard, with *send/receive*, *broadcast* and *reduction* operators is still used to construct parallel programs composed of tens to hundreds of thousands of communicating processes. Each interprocess communication is orchestrated by the developer based on knowledge of code function and overhead.

To date, attempts to develop higher level programming abstractions, tools and environments for high-performance computing have largely failed. There are many reasons for this failure, but I believe many are rooted in our excessive focus on hardware performance measures. By definition, the *raison d'être* for high-performance computing is high performance, but floating point operations per second (FLOPS) need not be the only measure. Human productivity, total cost and time to solution are equally, if not more important.

I am confident that high-performance computing can and should learn a few tricks from the world of web services. We need a *Ruby on Rails* for defining parallel application frameworks and an *Erlang* for concurrent specification. We need to focus on high-productivity computing, balancing human and machine performance.

You know the drill. Pour some hot water in the French press, break out your copy of *Lattice QCD for Dummies*, grab your cheat sheet of high-performance computing abstractions, download some digital tunes and start programming. In all seriousness, a new world of opportunity and scientific discovery awaits those who first embrace and master the abstractions needed to create rich, multidisciplinary parallel applications.

Low-level programming models: assessment

- ❖ Efficient (can be) and widely supported
- ❖ Not likely to scale to mainstream (industrial) development
- ❖ Not compositional
- ❖ Parallel behaviour not explicit (can be hard to catch)
- ❖ At the bottom line, higher-level “mental” overlays are already there
 - ❖ In the mind of designers (i.e. data organisation, partition, data patterns, ...)

Programming model: my wish list

- ❖ **Should enforce to think to problems in parallel & and at high-level of abstraction**
 - ❖ Clear semantics: functional and extra-functional (parallel), describing collective behaviours
 - ❖ Trading memory coherency for power, and power for parallelism should be a matter of the implementation
- ❖ **Should support containment and composition**
 - ❖ At large scale: clear fault model with containment & recovery
- ❖ **Should integrate synchronisation/communication with scheduling**
 - ❖ Weak execution model rather than per device. Multicore, GPGPUs, distributed with an unifying vision
- ❖ **System programmers should use the techniques they advocate**

An alternative mobility approach



Skeletons and Patterns

- ❖ From HPC community
 - ❖ Started in early '90
(M. Cole's PhD thesis)
 - ❖ Pre-defined parallel higher-order functions, exposed to programmers as constructs / lib calls
- ❖ From SW engineering community
 - ❖ Started in early '00
 - ❖ "Recipes" to handle parallelism (name, problem, algorithms, solutions, ...)

Algorithmic Skeletons

Parallel Design Patterns

Evolution of the concept

'90

- Complex patterns, no composition
- Targeting clusters
- Mostly libraries (run-time system)

'00

- Simple data / stream parallel patterns
- Compositional patterns, targeting COW / NOW
- Libraries + first compilers

'10

- Optimised, compositional building blocks
- Targeting cluster of heterogeneous multicore + GPGPU
- Quite complex tool chain (compiler + run-time system)

Evolution of the concept

'90

- Cole PhD thesis skeletons
- P3L (Pisa), SkiE (QSW / Pisa)
- SCL (Imperial college London)

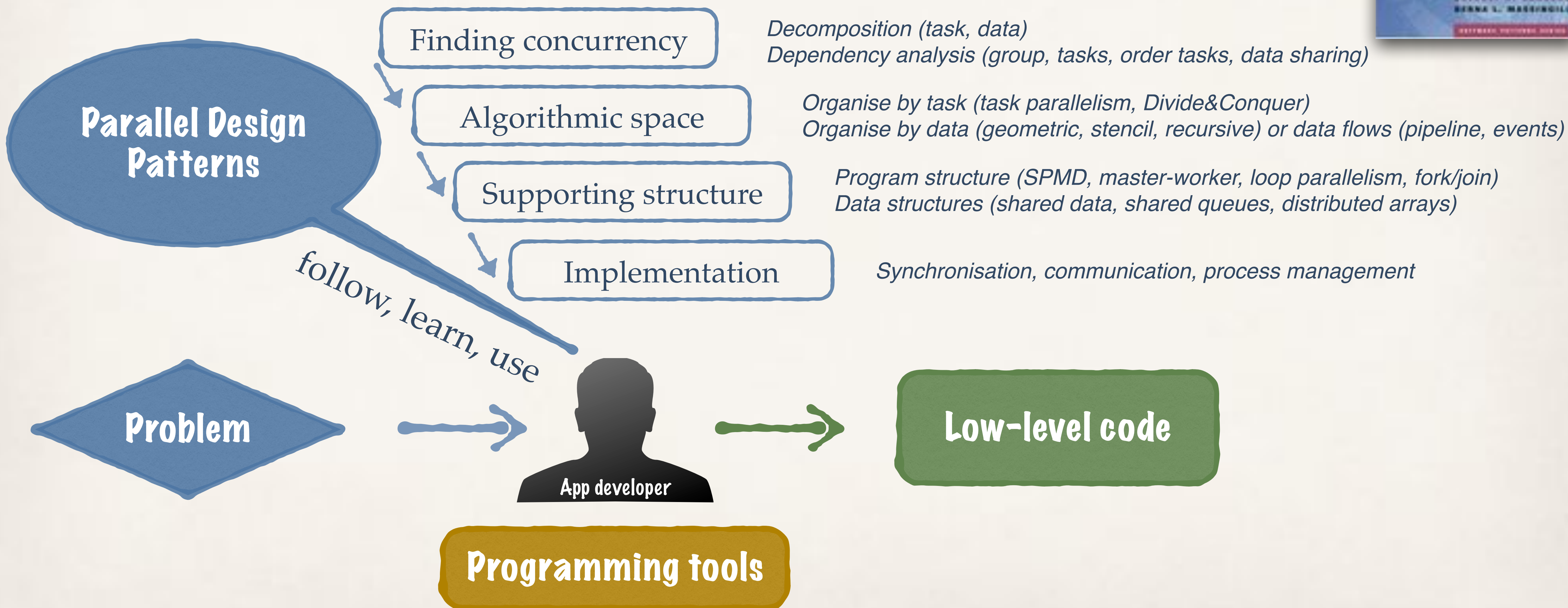
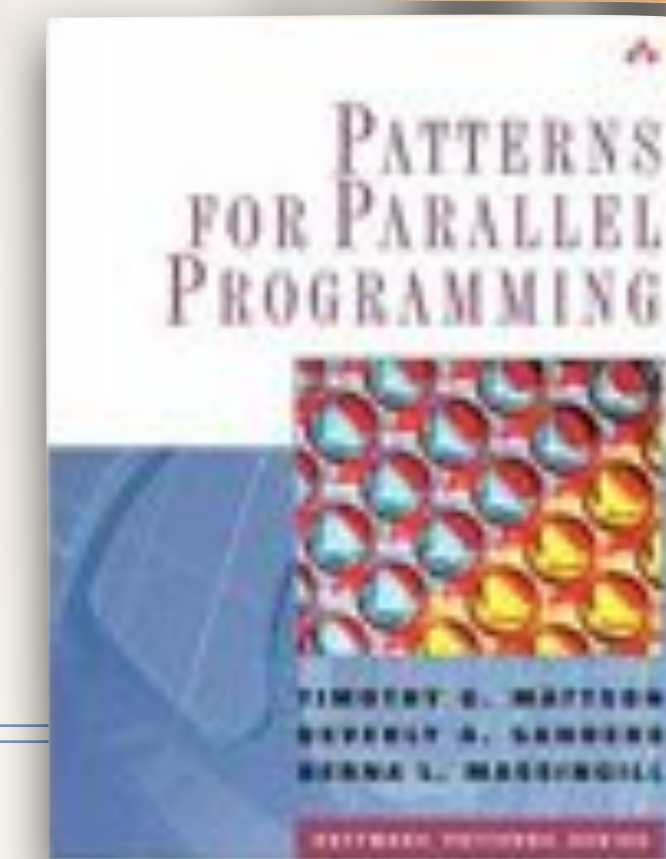
'00

- Lithium / Muskel (Pisa), Skandium (INRIA)
- Muesli (Muenster), SkeTo (Tokio)
- OSL (Orleans), Mallba (La Laguna), Skipper

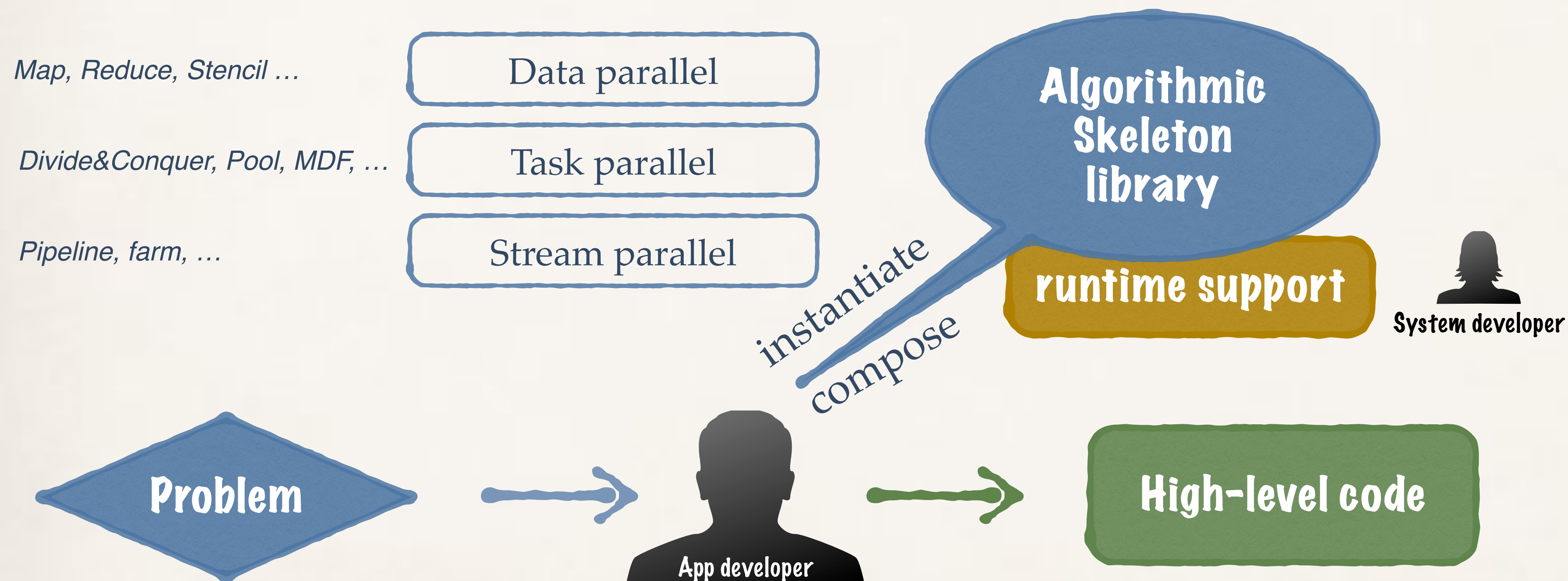
'10

- SkePu (Linkoping)
- FastFlow (Pisa / Torino)
- Intel/TBB/CnC (?), Microsoft/TPL (?), Google/MapReduce (?), ...

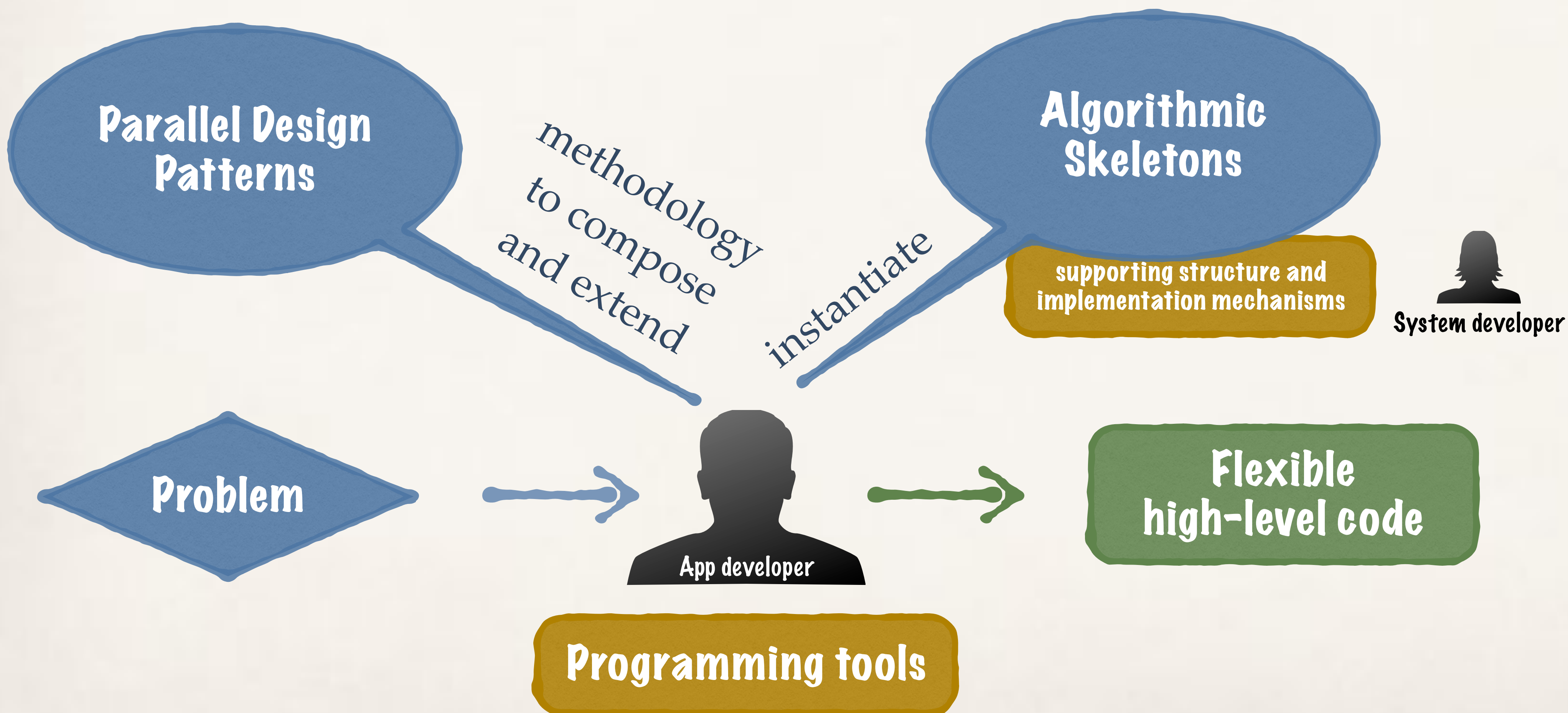
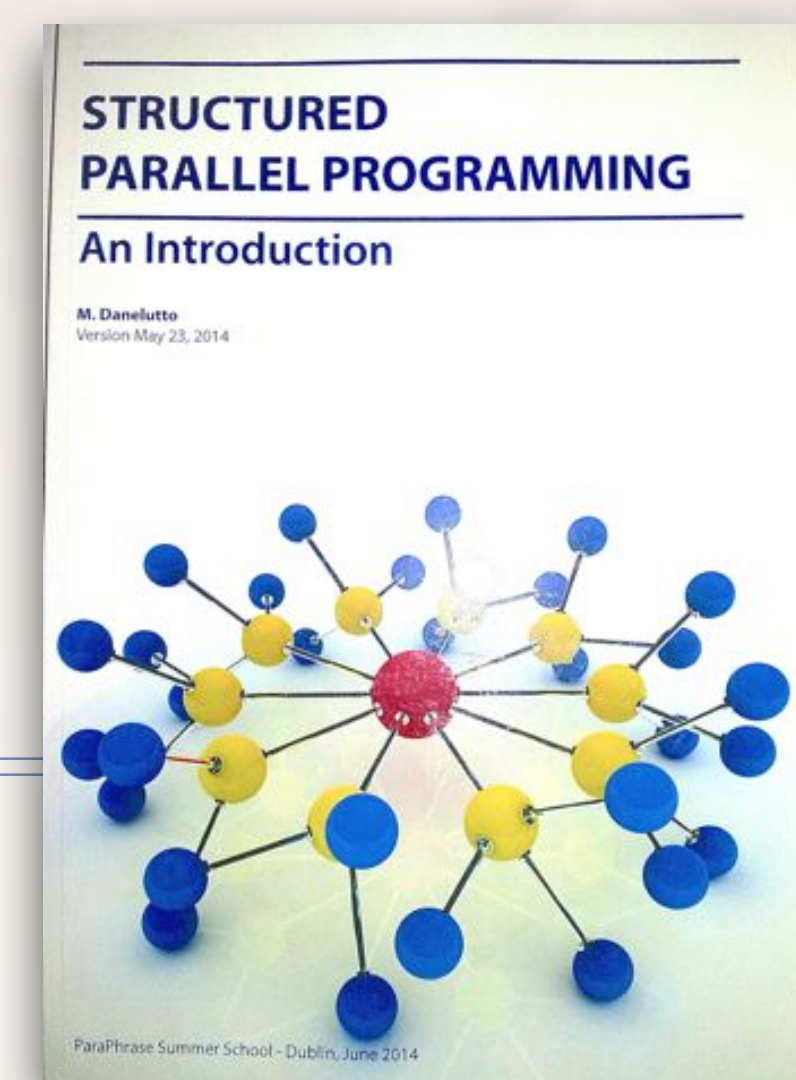
Parallel Design Patterns



Algorithmic skeletons



Structured parallel programming



MapReduce a success story

Innovative implementation
(even if theoretically an
instance of Map+Reduce)

TOWARDS PARALLEL PROGRAMMING BY TRANSFORMATION: THE FAN SKELETON FRAMEWORK*

M. ALDINUCCI[†], S. GORLATCH^{†,‡}, C. LENGAUER[†]
and S. PELAGATTI[†]

[†]Dipartimento di Informatica, Università di Pisa, 40 Corso Italia,
I-56125 Pisa, Italy; [‡]Fakultät für Mathematik und Informatik,
Universität Passau, D-94030 Passau, Germany

FAN programs obey the single-
defining each variable. All outp
in the program's body. Each eq
Expressions (e) can be const
function applications or skeleto
as map, reduce, etc.,

$e = c \mid x \mid ee \mid \lambda x.e \mid E_1 ee \mid E_2 e \mid E_3 eee \dots$
 $E_1 = \text{map} \mid \text{map}_k \mid \text{reduce} \mid \text{scanL} \mid \text{copy} \mid \text{split} \mid \text{part} \mid \text{rearrange}$
 $E_2 = \text{pair} \mid \text{proj1}$
 $E_3 = \text{loopfor} \mid \text{loopwhile} \mid \text{looprepeat}$

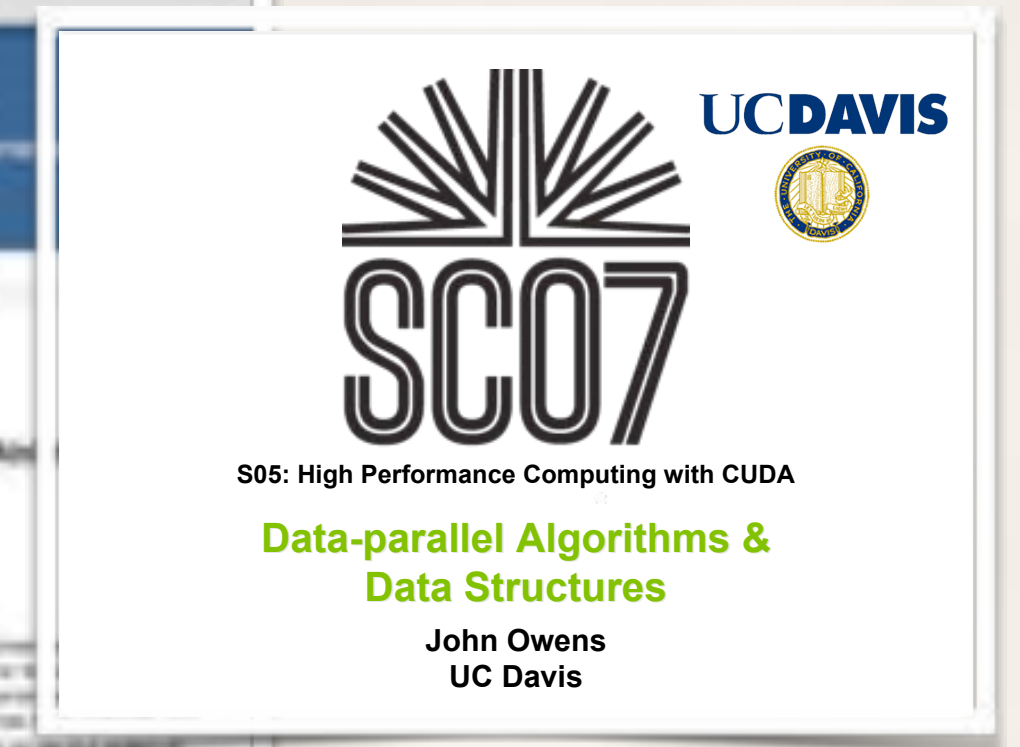
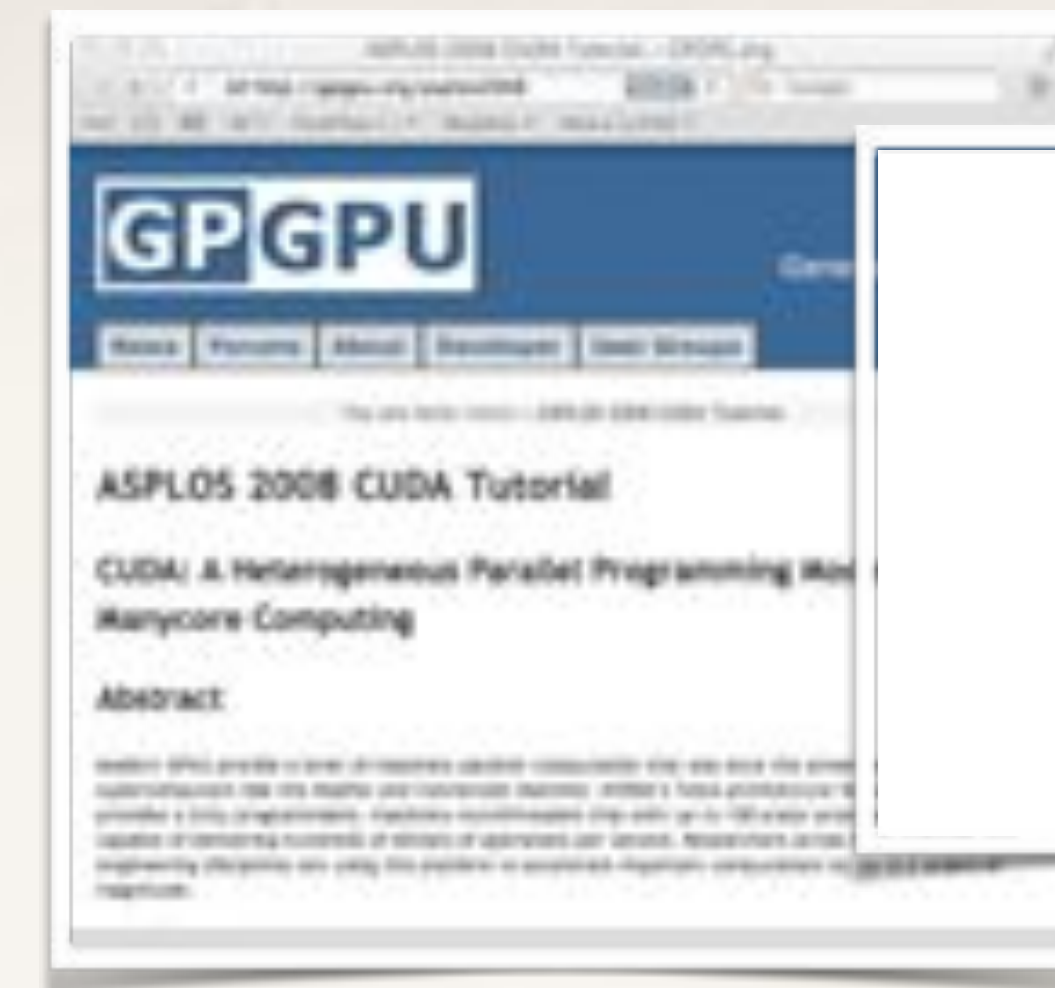
The scope of each variable definition extends across all subsequent
definitions in the same body. At the end of the body, we can specify local
definitions using a `where` clause. Names of FAN programs can be used as
functions in expressions.

```
inner.product (in a, b : Array n Scalar, out c : Scalar)
  t = map (*) (pair (a,b));
  c = reduce (+) t;
```

FIGURE 1 A FAN program to compute the inner product of two vectors.

Patterns are natural for GPGPUs

Courtesy: J. Owen, UC Davis



Think In Parallel



- **The GPU is a data-parallel processor**
 - Thousands of parallel threads
 - Thousands of data elements to process
 - All data processed by the same program
 - SPMD computation model
 - Contrast with task parallelism and ILP
- **Best results when you “Think Data Parallel”**
 - Design your algorithm for data-parallelism
 - Understand parallel algorithmic complexity and efficiency
 - Use data-parallel algorithmic primitives as building blocks

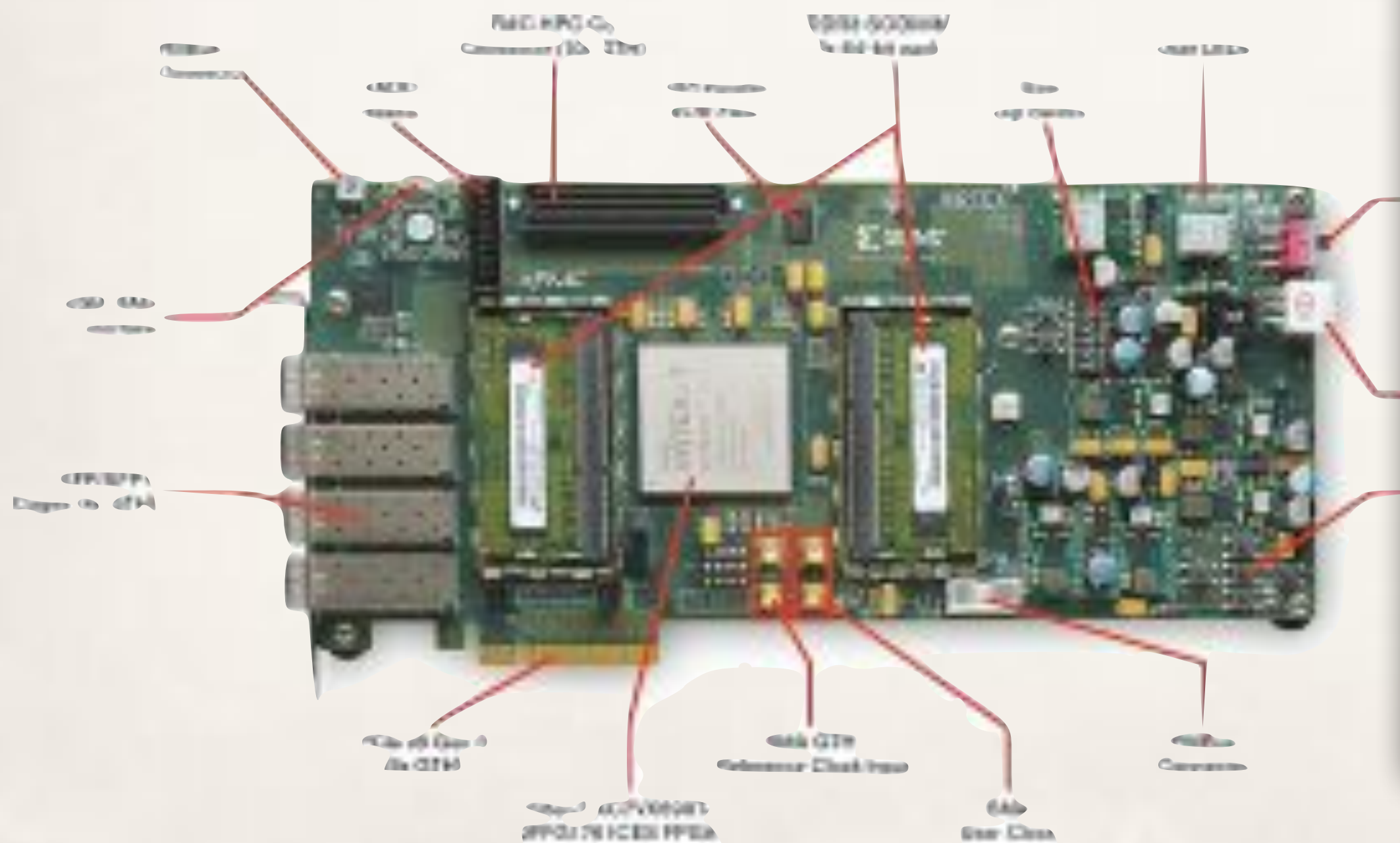
Data-Parallel Algorithms



- **Efficient algorithms require efficient building blocks**
- **This talk: data-parallel building blocks**
 - Map
 - Gather & Scatter
 - Reduce
 - Scan

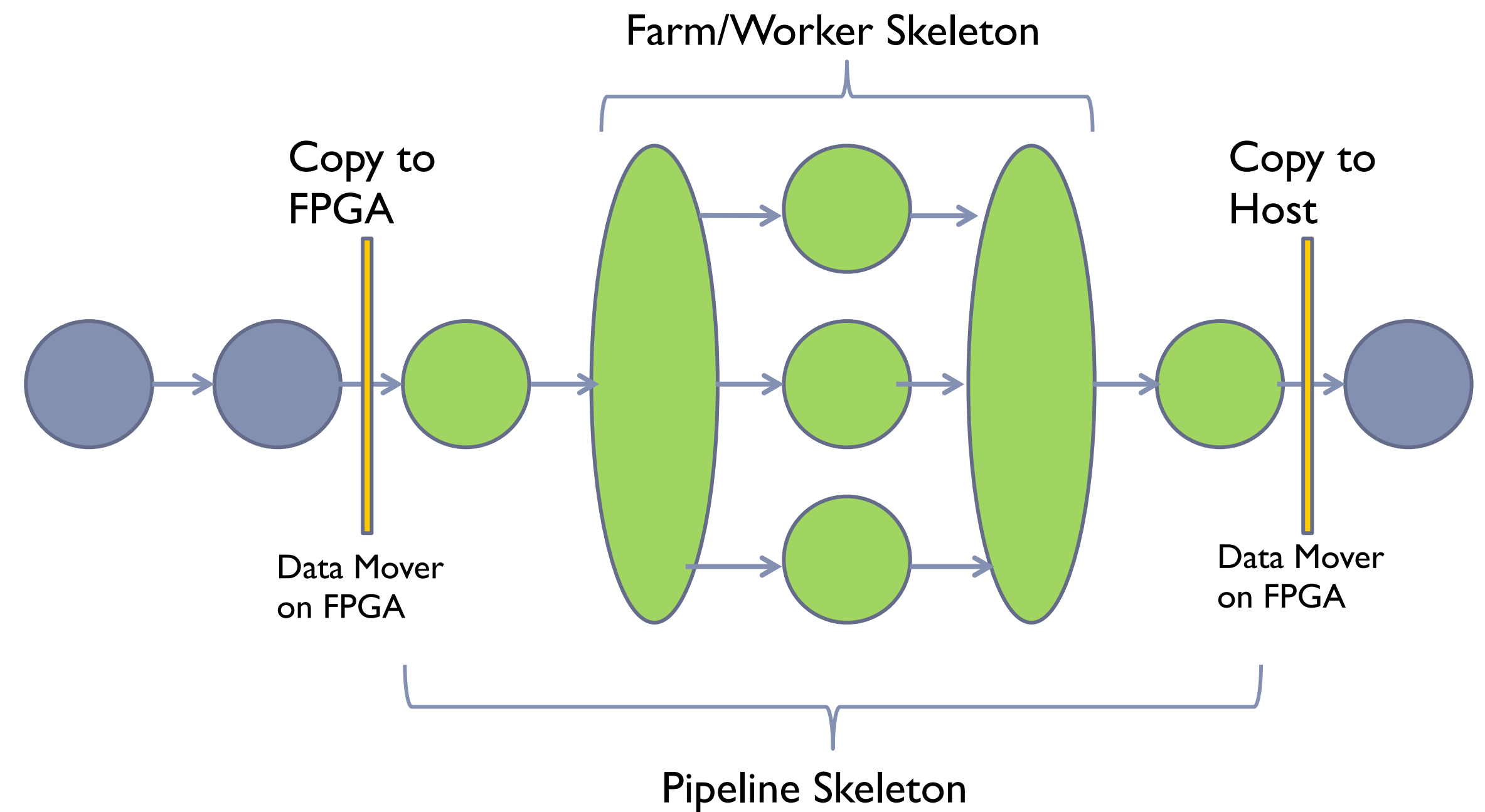
Maybe also on FPGA

Courtesy: A. Koch, TU Darmstadt



Role of FPGA in FastFlow

- ▶ FPGA can act as entire FastFlow subgraph



Assessment

Separation of concerns

- Application programmer: **what to do**
- System programmer: **how to make it in parallel**

Inversion of control

- Structure suggested by programmer - no idiom recognition
- Clear functional and parallel semantics

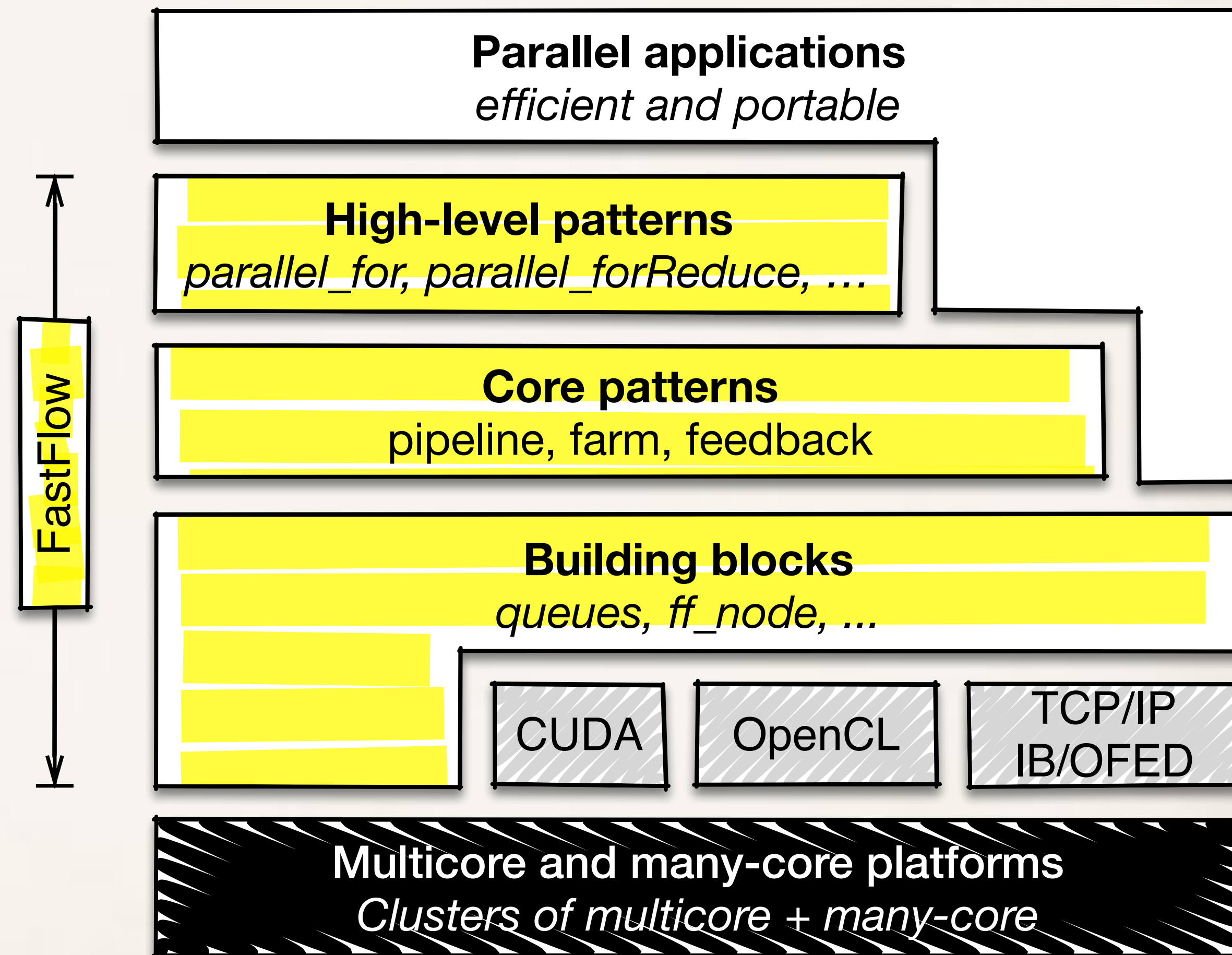
Performance

- Close to hand tuned code (better for not expert programmers)
- Reduced development time. Graceful tuning curve.

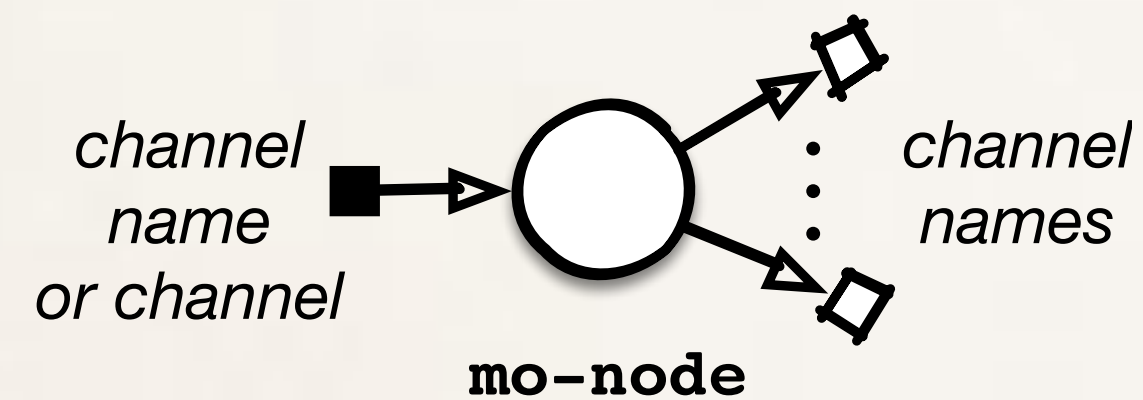
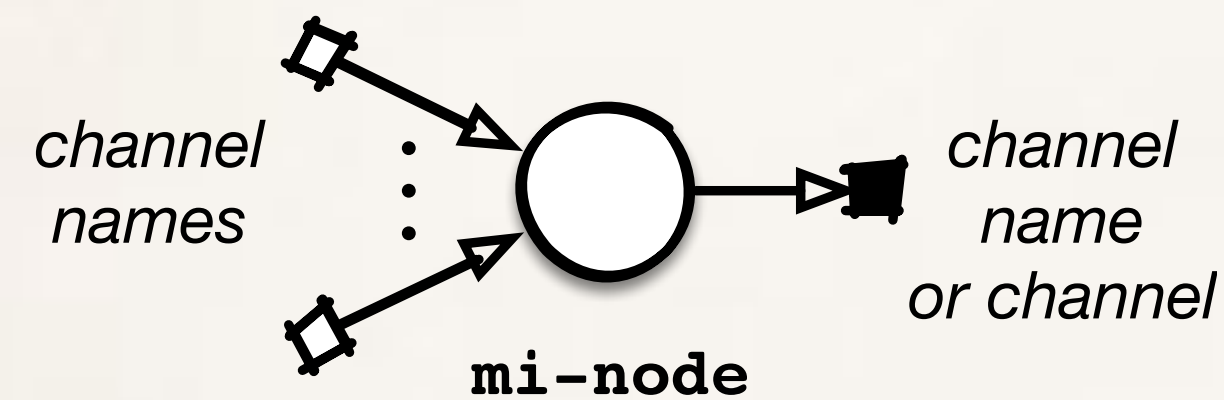
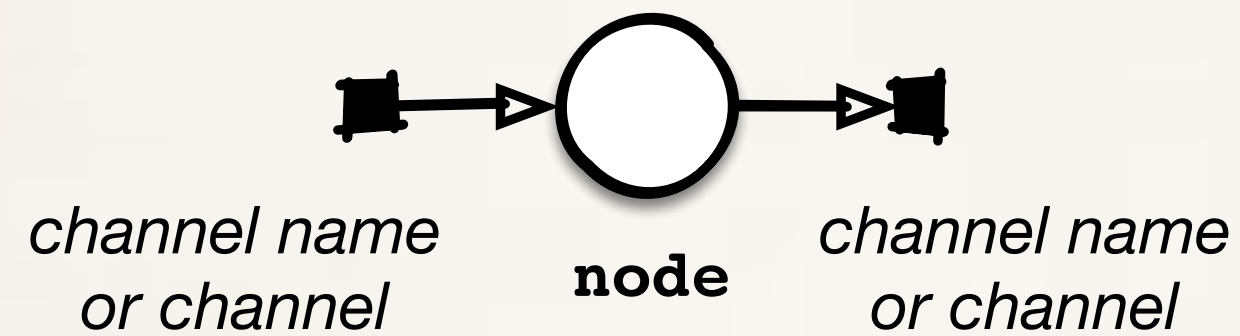
FastFlow (FF)



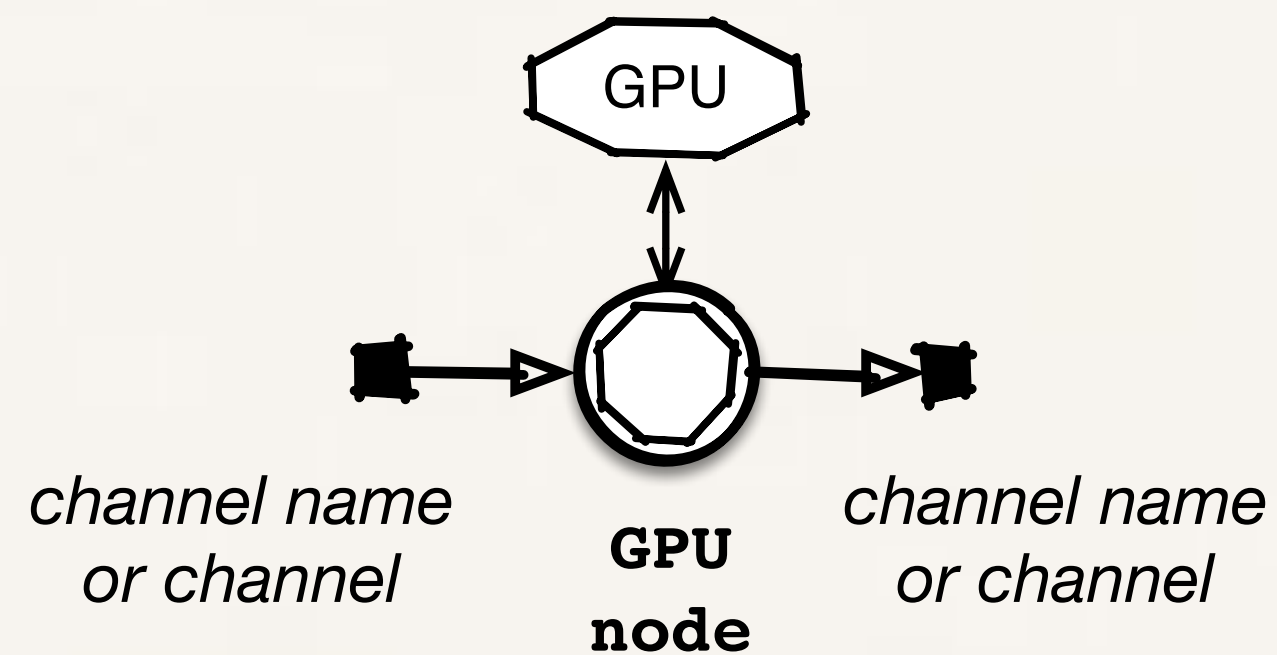
and its data-centric run-time support



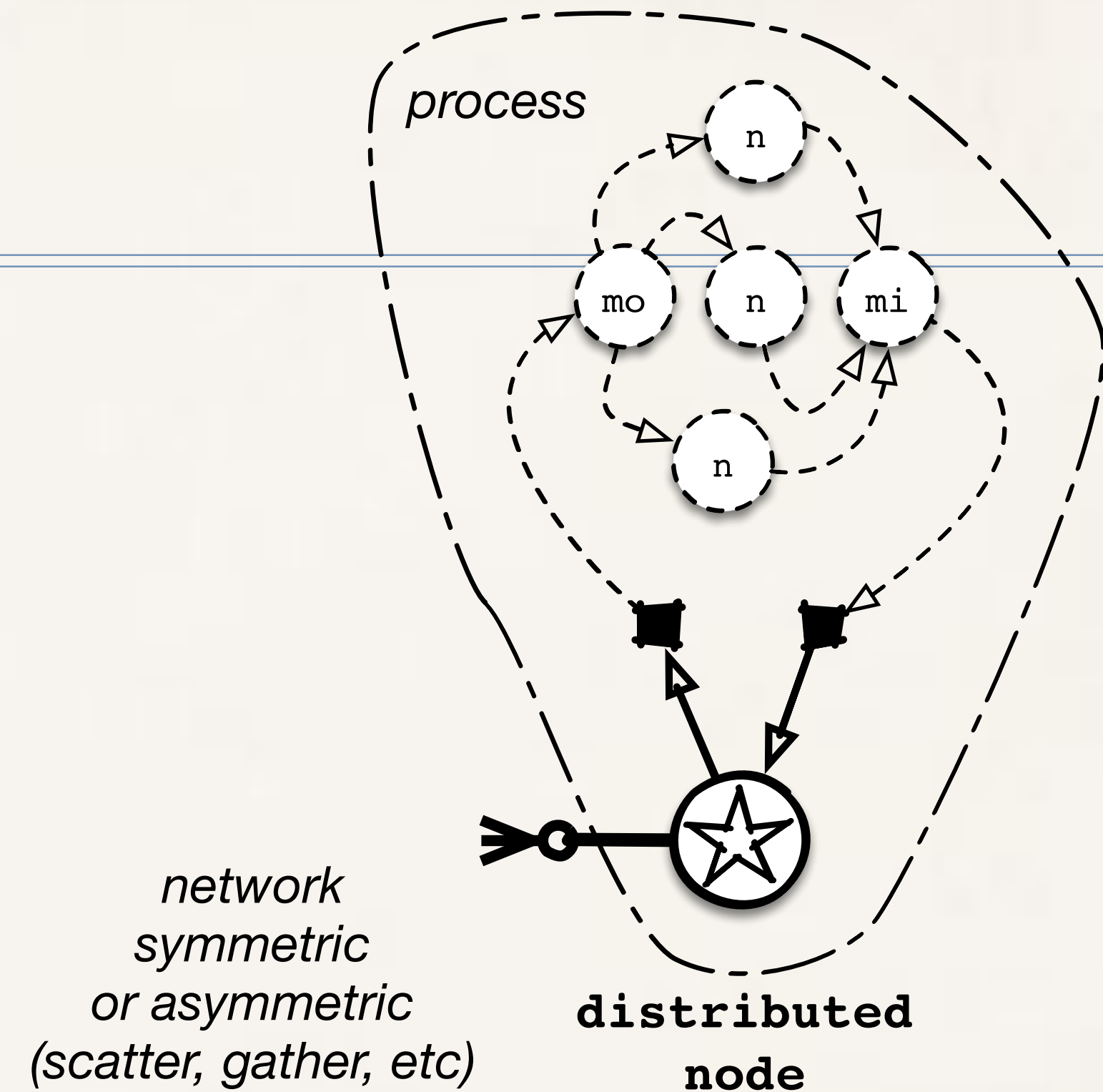
Building blocks



non-blocking threads
(can switch to blocking at runtime by way of a native protocol)



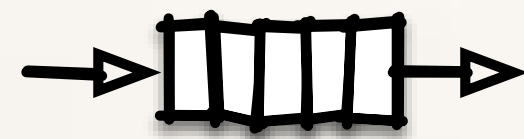
nonblocking thread
GPU/accelerators
(OpenCL, CUDA)



process of nonblocking threads
distributed zero-copy
TCP/IP, OFED/IB, MPI (ongoing),
HW/SW PGAS (ongoing)

Building blocks

Everything is SPSC (in the shared-memory)



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



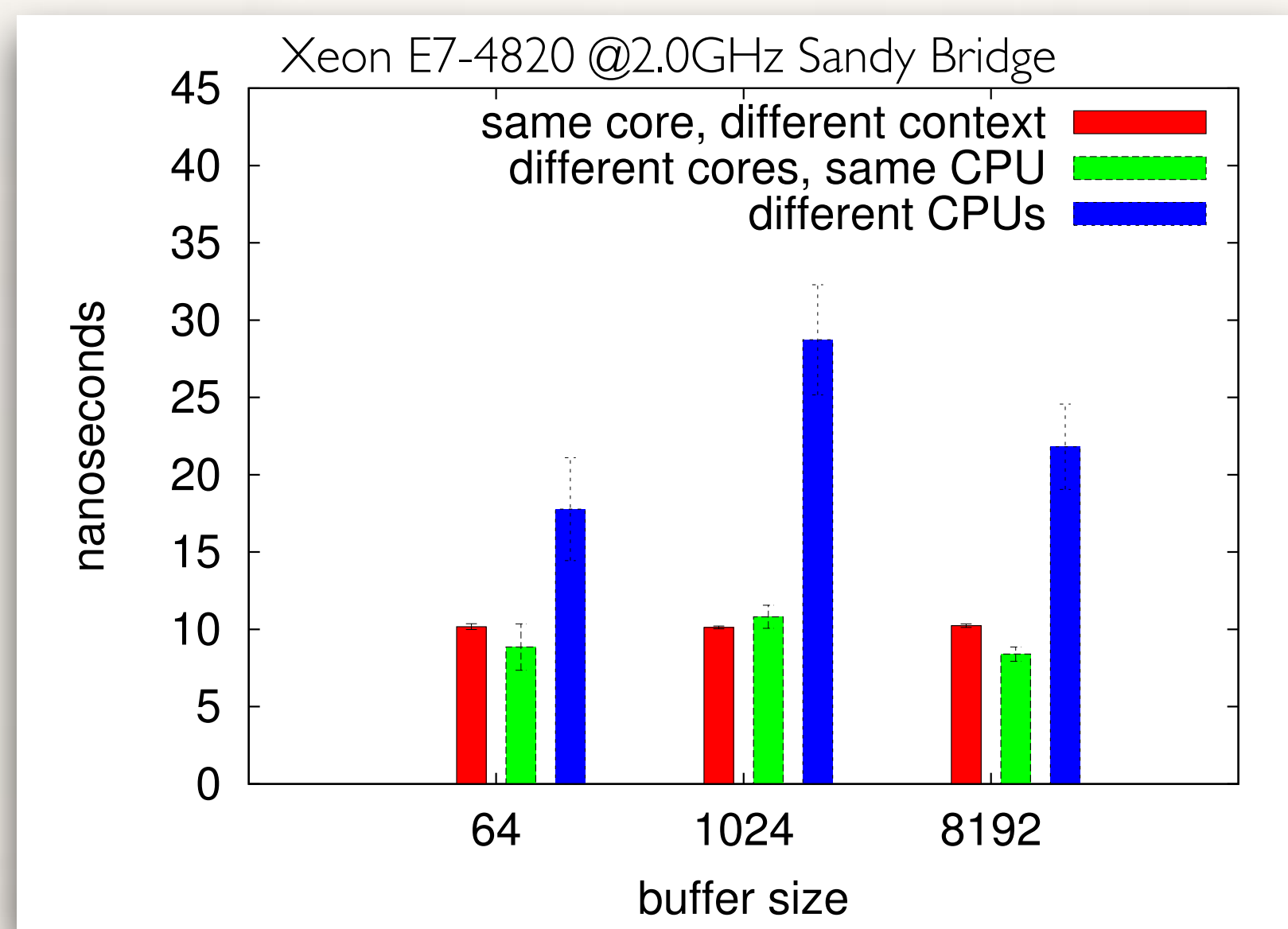
Distributed zero-copy channel
OMQ/TCP or native IB/OFED

shmem channels communicate
pointers in a message passing style

- ❖ Enough to support producer-consumer
- ❖ Inherently weaker w.r.t. mutual exclusion
- ❖ Weaker execution model requirements
 - ❖ Mutex not really possible on SIMT model (GPU)
 - ❖ Mutex requires memory-fences and leverages on (expensive) cache coherency on multicore
- ❖ Deadlock is cyclic networks avoided via unbound queue (wait-free)
 - ❖ *M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati. An efficient unbounded lock-free queue for multi-core systems. Euro-Par 2012, LNCN 7484, 2012.*

Queues: performance

Shared-memory



- ❖ MVAPICH (shmem) ~ 190ns
- ❖ Faster and more scalable than CAS/test-and-set implementation

Distributed

Message size (bytes)	ib_write_bw (Mb/s)	MPI (Mb/s)	FastFlow /IB (Mb/s)	FastFlow/ZMQ /IPoIB (Mb/s)
10	300	192	129	0.7
100	3,600	1,816	1,300	7.0
1,024	22,900	13,936	10,591	70.0
5,000	25,200	23,880	19,761	300.0
10,000	25,500	25,128	20,479	500.0
25,000	25,700	12,408	20,051	1,100.0
50,000	25,800	16,232	21,019	1,950.0
65,536	22,900	17,472	20,889	1,980.0
200,000	25,800	21,208	21,211	3,800.0
400,000	25,800	22,532	21,226	6,200.0

TABLE I: Comparing throughput of different implementations of the *unidirectional bandwidth* test for several message sizes.

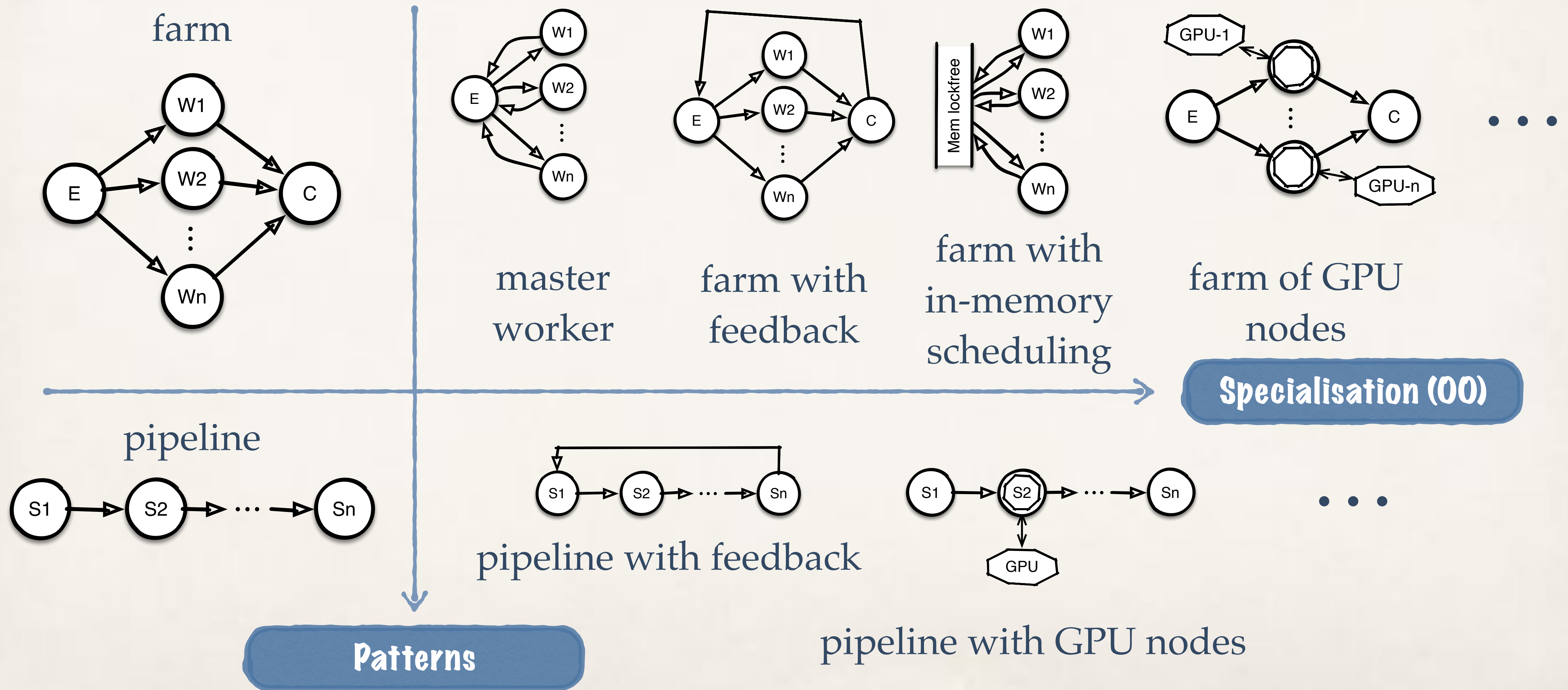
- ❖ Comparable with MVAPICH (distributed)

Core patterns

- ❖ Streaming patterns: fast data movements + computation-to-core pinning
 - ❖ Full control of locality and memory affinity
- ❖ Fully compositional and deadlock-free (by construction)
- ❖ Minimalistic run-time
 - ❖ Two patterns and one pattern-modifier: farm, pipeline and feedback
 - ❖ Configurable scheduling and gathering policies and implementations
- ❖ Full expressivity
 - ❖ Enough to build higher-level abstraction: Master-worker, D&C, Map, Reduce, Pool, Stencil, ...

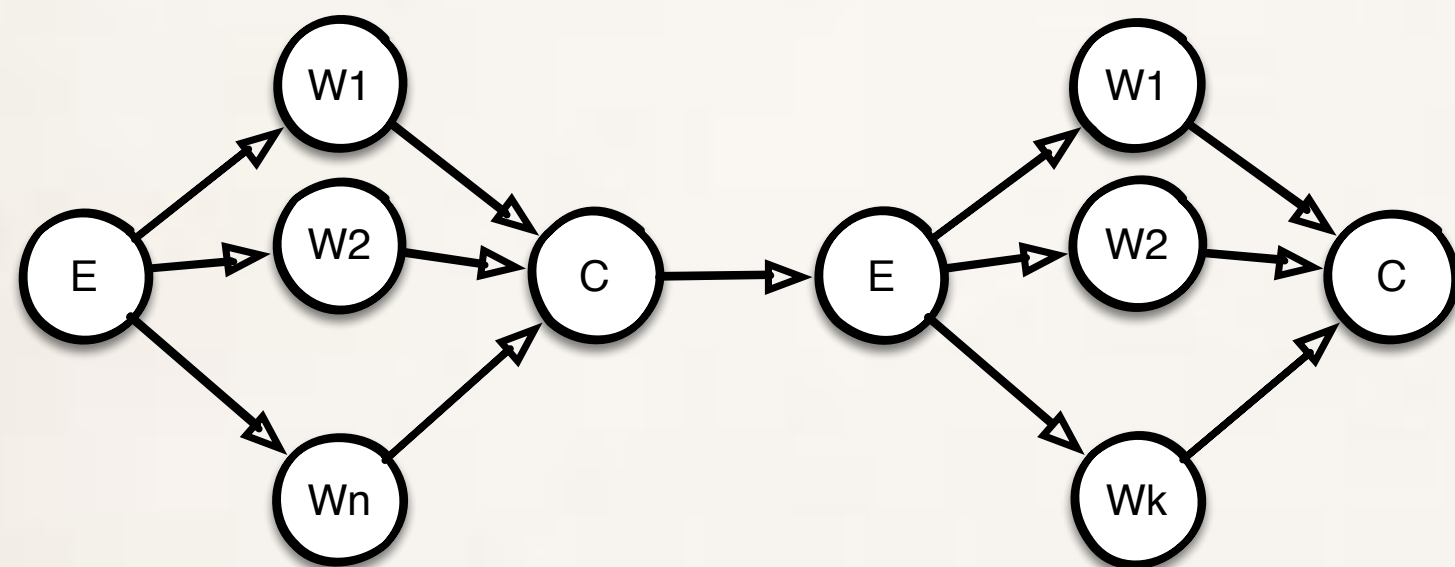
Core patterns: farm, pipeline, feedback

+ Nonblocking/blocking, dynamic/static scheduling, E/C policy configuration, core pinning

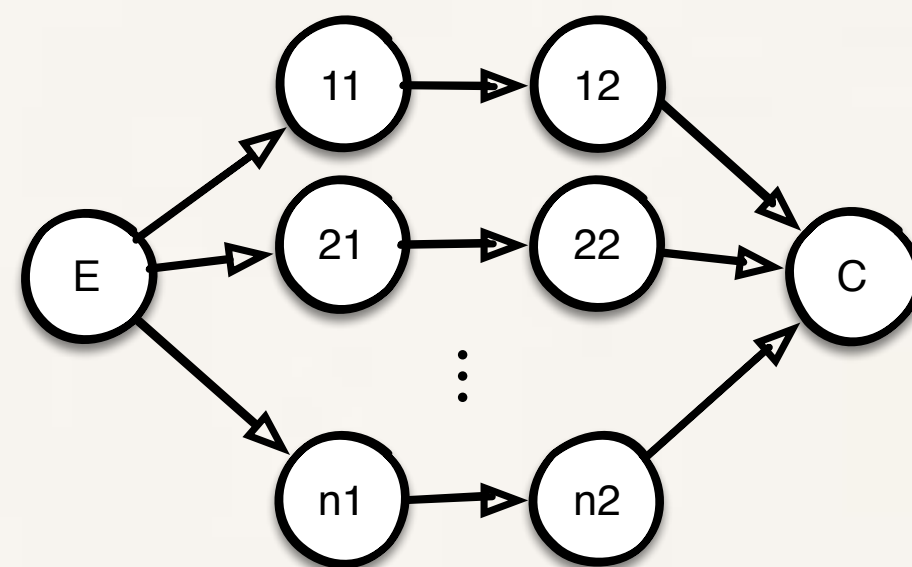


Core patterns can be arbitrarily composed

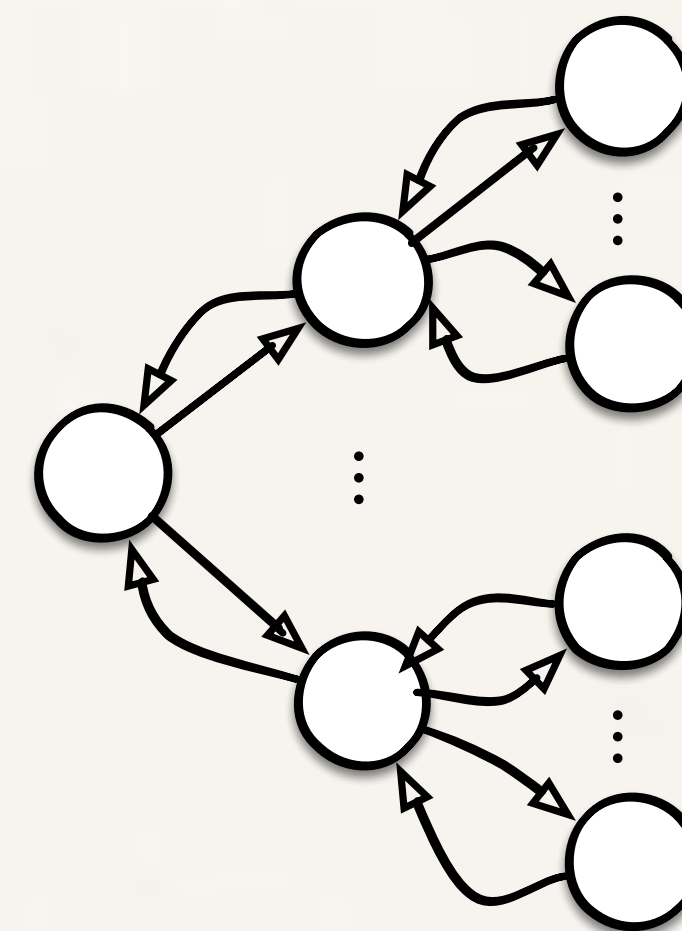
pipe(farm(s),farm(s))



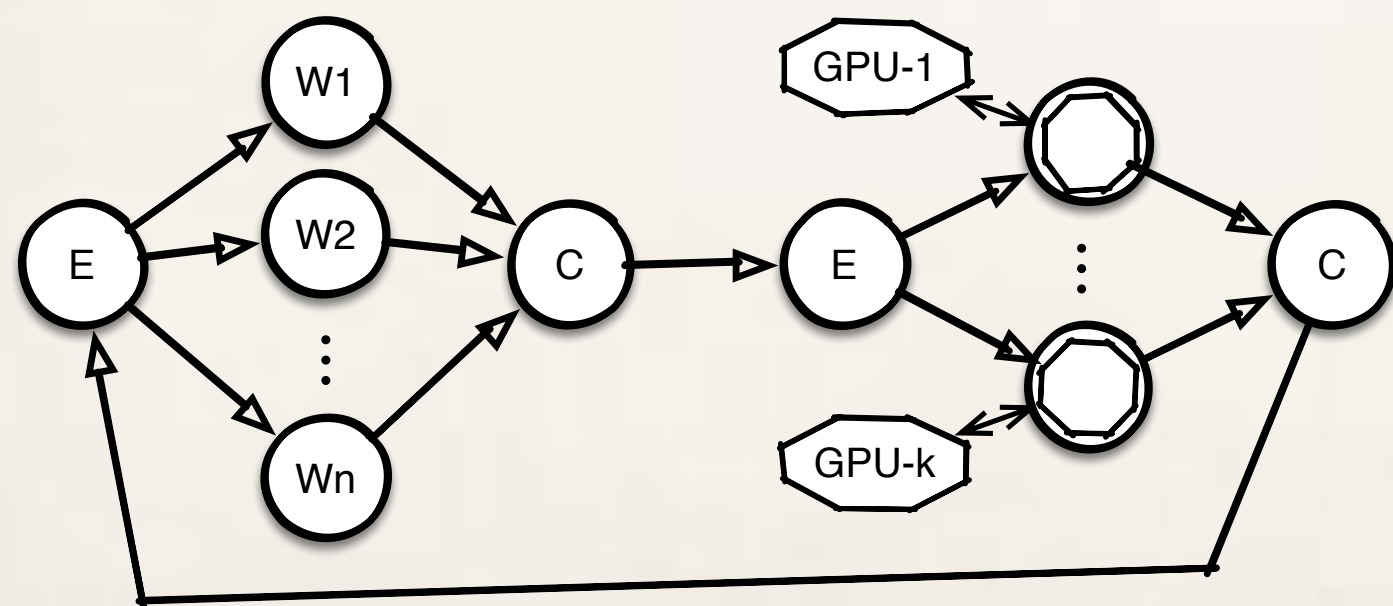
farm(pipe(s1,s2))



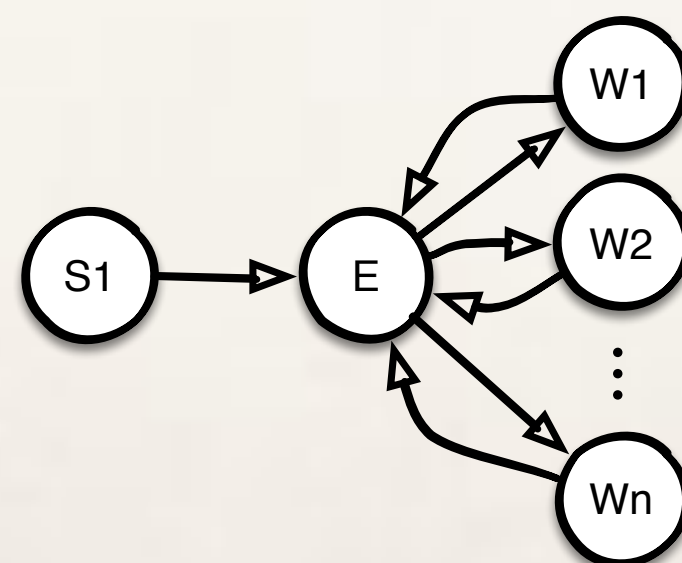
farm(farm(s))



pipe(farm(s),farm(s),loopback)

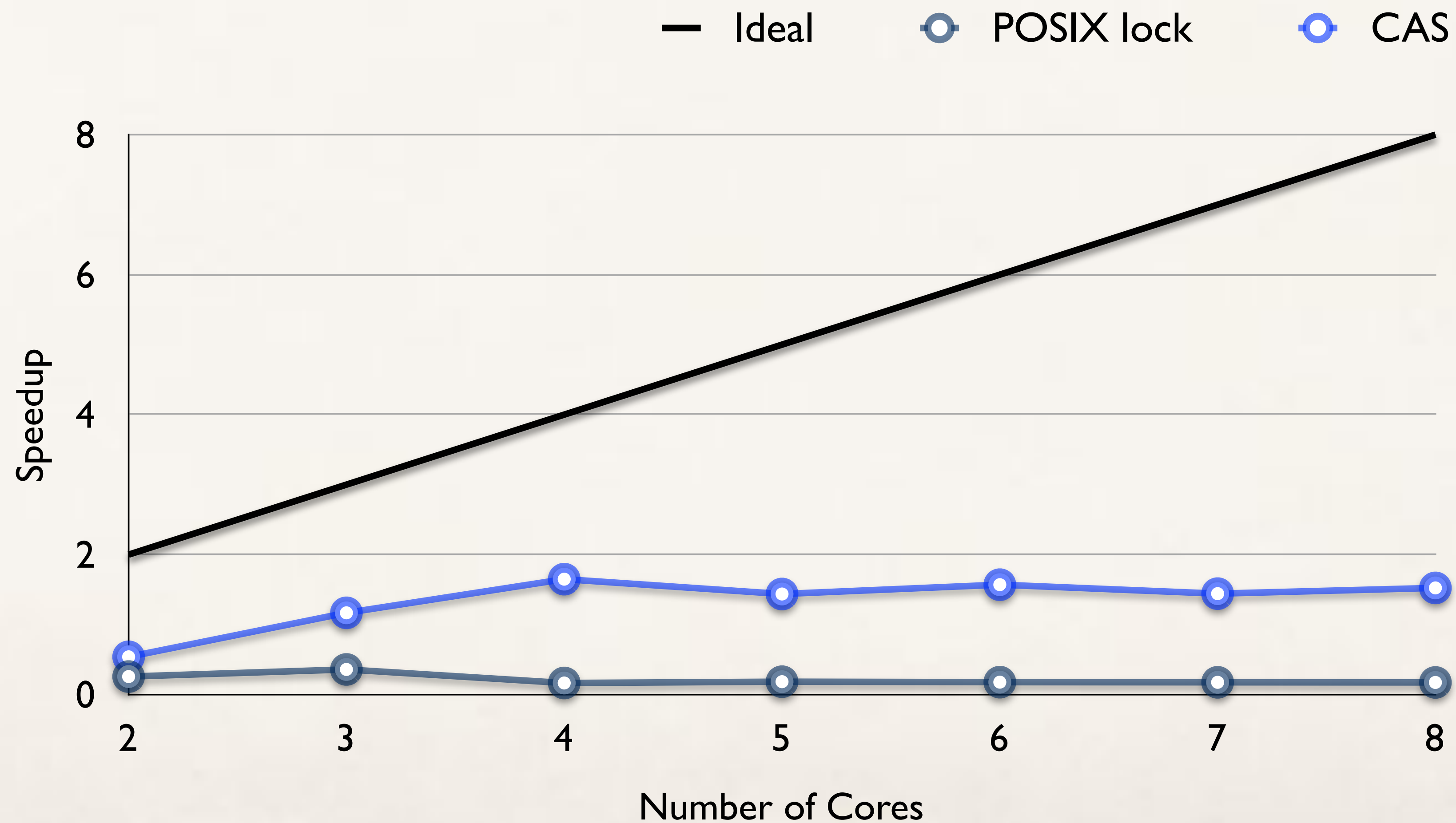
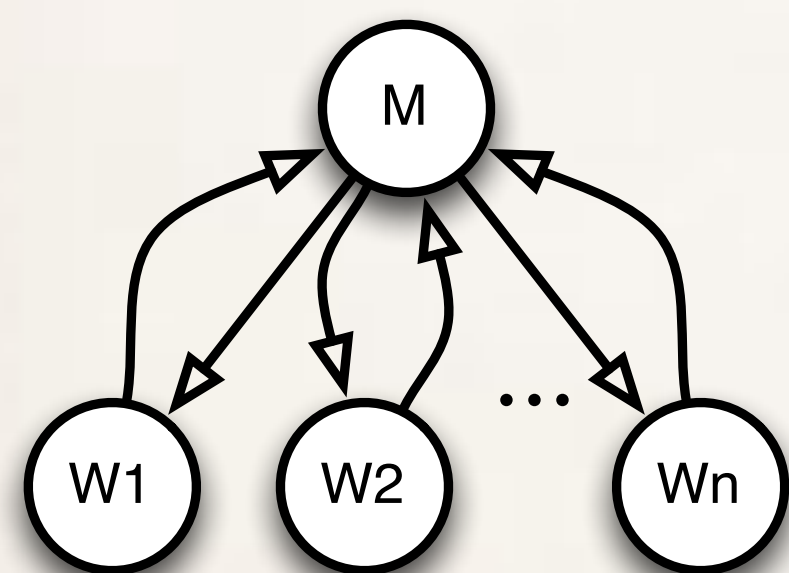


pipe(s1,farm(s2))

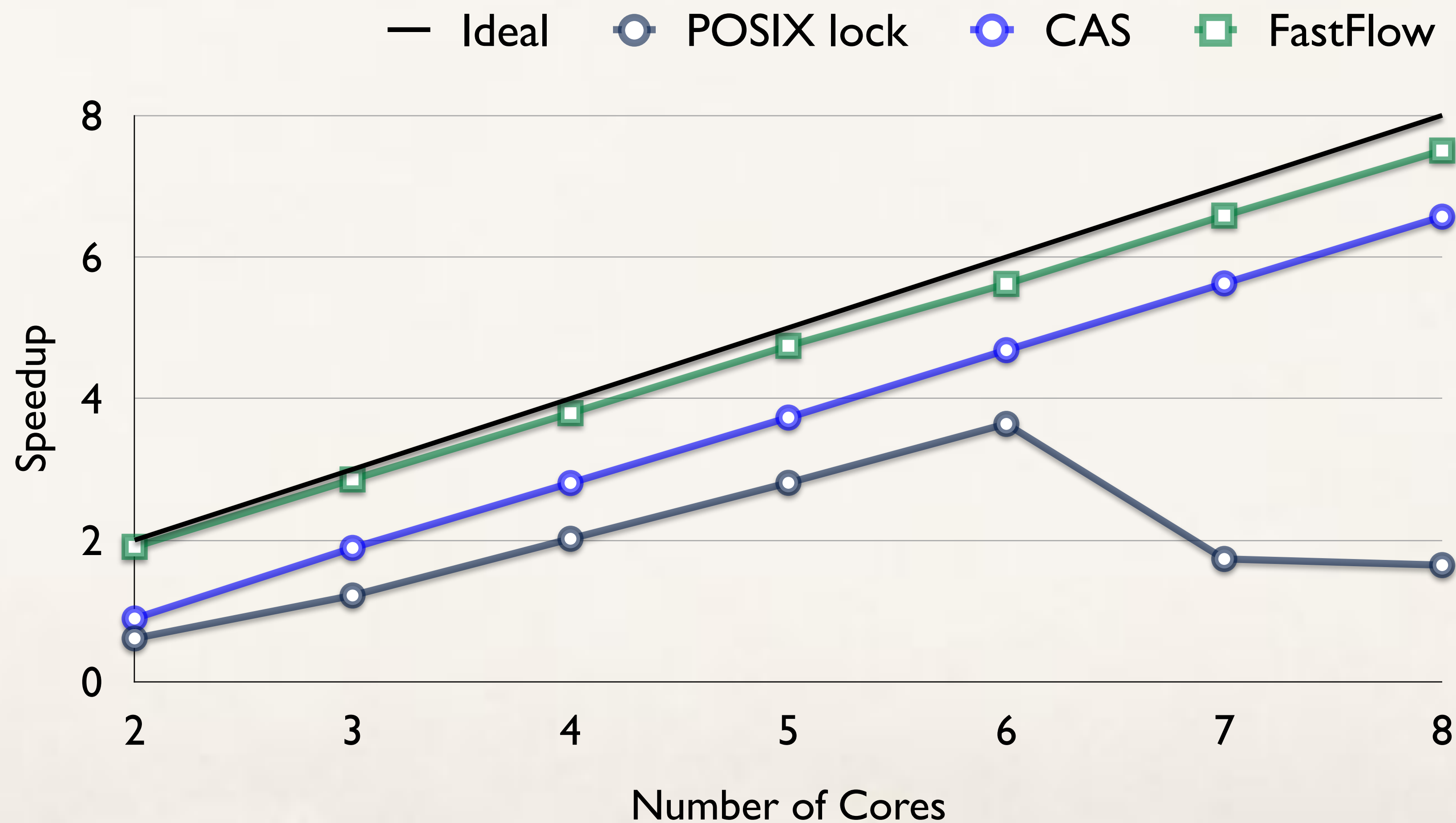
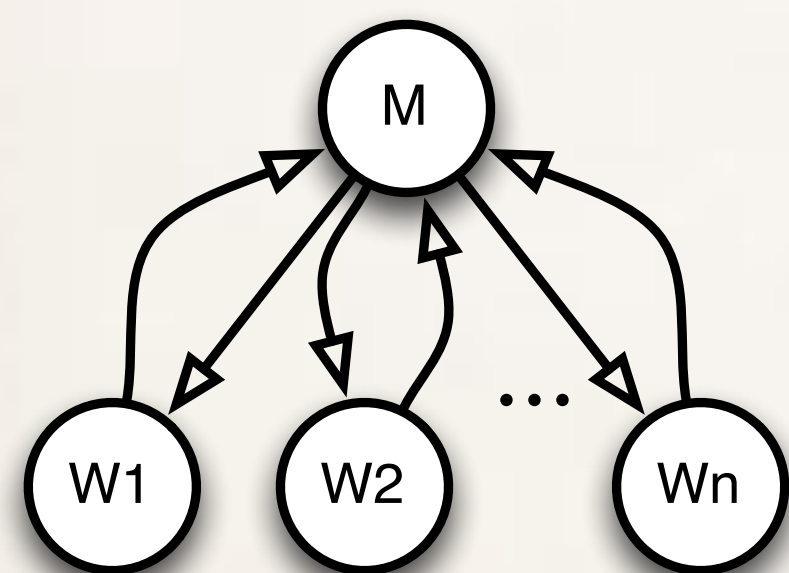


...

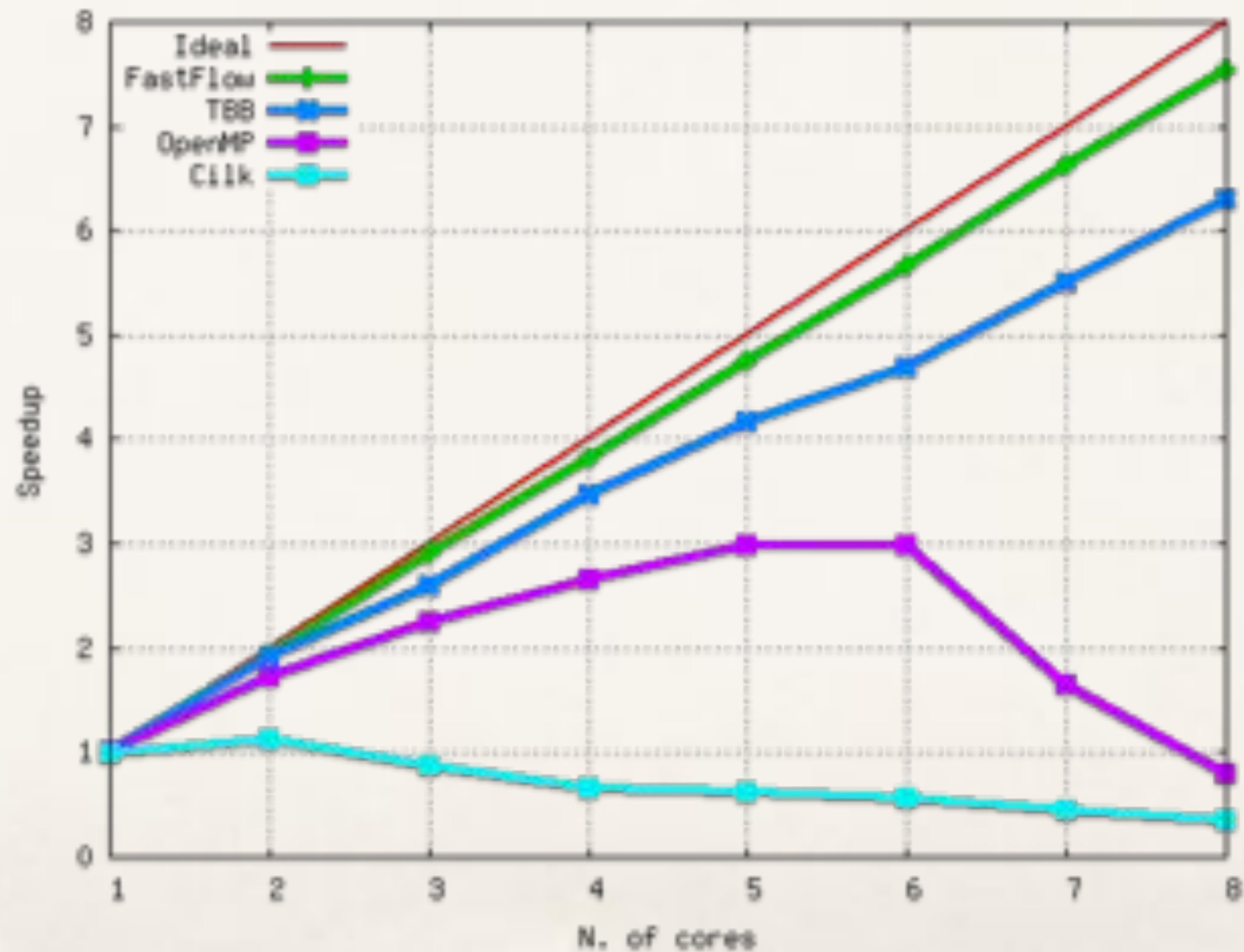
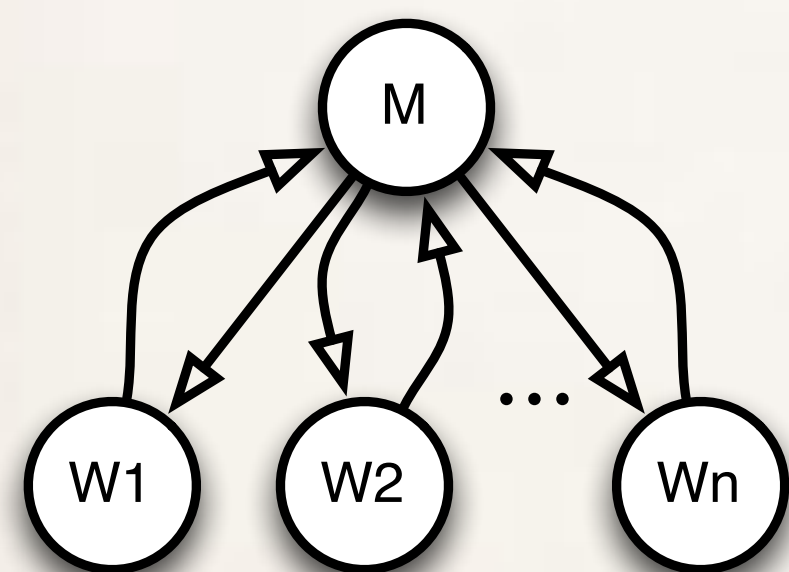
Master-worker ($0.5 \mu\text{S}$ workload)



Master-worker ($5 \mu\text{S}$ workload)



Master-worker ($5 \mu\text{S}$ workload)



Core patterns: a programming model

- ❖ A low / medium-level **data-centric programming model**
 - ❖ Concurrent computation modelled as a (cyclic) graph
 - ❖ Nodes are parallel activities. Edges are true data dependencies
 - ❖ Synchronisation are messages, data can be moved as messages or shared memory
 - ❖ Can be realised with or without coherency, in shared-memory, distributed, PGAS, ...
- ❖ Not exactly a Kahn's net, more a CSP-actor hybrid model
 - ❖ Processes are named and the data paths between processes are identified

High-level patterns

- ❖ Address application needs
 - ❖ Loop parallelism (OpenMP fashion)
 - ❖ Parallel-For, Parallel-For-Reduce
 - ❖ Data Parallelism
 - ❖ Stencil, Stencil-Reduce, Map, MapReduce (pipelined)
 - ❖ Task & Stream
 - ❖ Pool (e.g. genetic alg.), Macro-Data-Flow (e.g. linear algebra, dynamic programming, ...)
 - ❖ Farm, Pipeline
- ❖ Implementation-wise, just OO extensions of composition of core patterns

High-level patterns

- ❖ Think in parallel & high-level
 - ❖ Efficiency, portability, time-to-market
- ❖ High-level parallel patterns
 - ❖ Describing collective behaviour
 - ❖ Can be: expressive, efficient, compositional
 - ❖ Multicore, GPGPUs, distributed with an unifying vision
- ❖ Thanks to clear semantics, can be autonomically reconfigured
 - ❖ not discussed in this talk - see ParaPhrase EU STREP FP7 project

Autonomic management of non-functional concerns in distributed & parallel application programming

Marco Aldinucci
Dept. Computer Science
University of Torino
Torino – Italy
aldinuc@di.unito.it

Marco Danelutto
Dept. Computer Science
University of Pisa
Pisa – Italy
marcod@di.unipi.it

Peter Kilpatrick
Dept. Computer Science
Queen's University of Belfast
Belfast – UK
p.kilpatrick@qub.ac.uk

Abstract

An approach to the management of non-functional concerns in massively parallel and/or distributed architectures that marries parallel programming patterns with autonomic computing is presented. The necessity and suitability of the adoption of autonomic techniques are evidenced. Issues arising in the implementation of autonomic managers taking care of multiple concerns and of coordination among hierarchies of such autonomic managers are discussed. Experimental results are presented that demonstrate the feasibility of the approach.

Keywords: autonomic management, algorithmic skeletons

functional concerns are those concerns not directly related to the result computed by an application, but rather to the way this result is computed. Examples of non-functional concerns include performance, security, fault tolerance. Management of such concerns usually requires extensive knowledge of the target execution environment and appropriate interaction with the functional code of the application. Non-functional concern management becomes increasingly complex as the target architecture becomes more and more dynamic and heterogeneous, and the features of the target execution environment are progressively hidden from the application programmer (moving from clusters to “invisible grids” [6] and from grids to clouds). This sug-


M. Aldinucci, M. Danelutto, P. Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. IPDPS, 2009. IEEE.



Example: parallel for

```
// FastFlow (--std=c++11)
ff::ParallelFor pf;
pf.parallel_for(0L,N,[&A](const long i) {
    A[i]+=1;
},nworkers);
```

```
// OpenMP (-fopenmp)
#pragma omp parallel for num_threads(nworkers)
for(long i=0;i<N;++i) {
    A[i]+=1;
};
```

- ❖ Currently a method call on a C++11 Lambda function (loop body)
 - ❖ All other high-level patterns in the same style
- ❖ Moving to C++11 generalised attribute syntax (N2761)
 - ❖ Within REPARA EU-FP7 STREP project  REPARA

```
[[ff::target(ff::cpu,ff::gpu), ff::input(A), ff::output(A), ...]]
for( ; ; ; ) { ... }
```

Applications

Stream: Sequence Alignment Tools

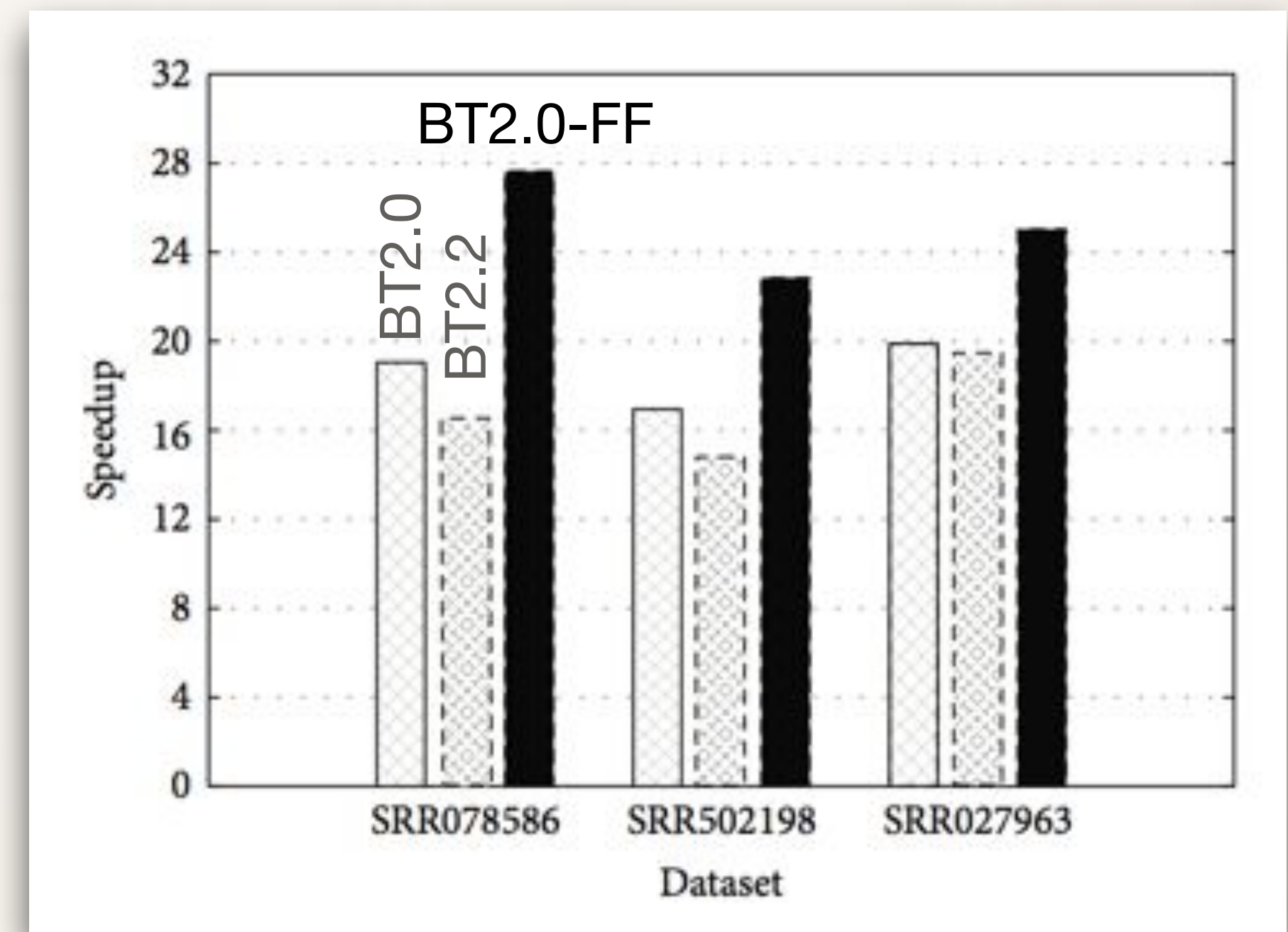
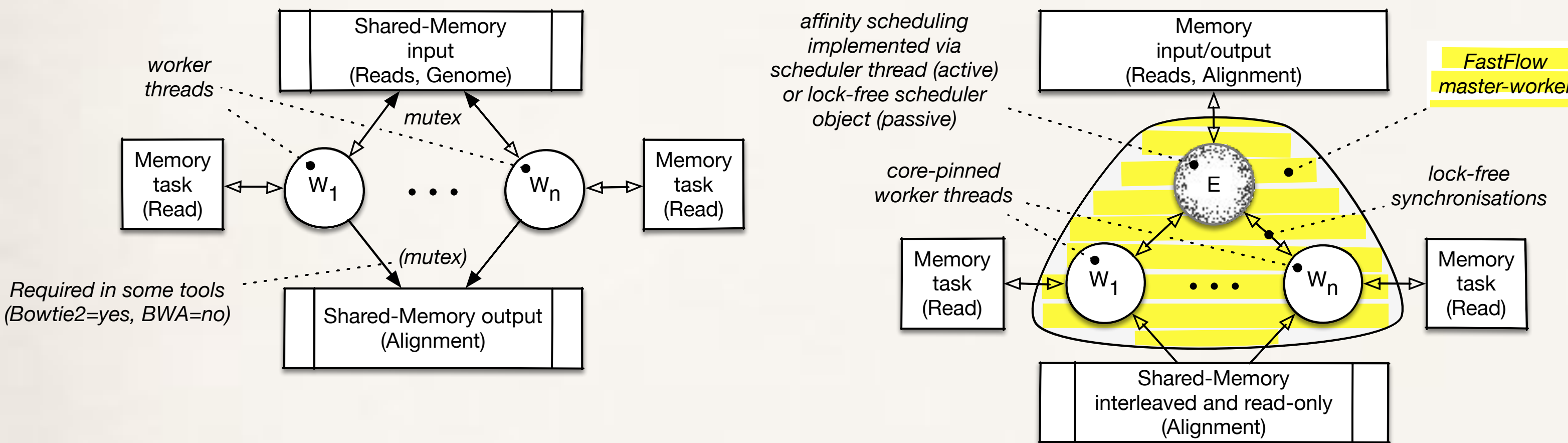
Tasks: Linear Algebra

Data: Image Filtering

System programmers should use the techniques they
advocate: memory allocation



Stream: Bowtie (BT) and BWA sequence alignment tools



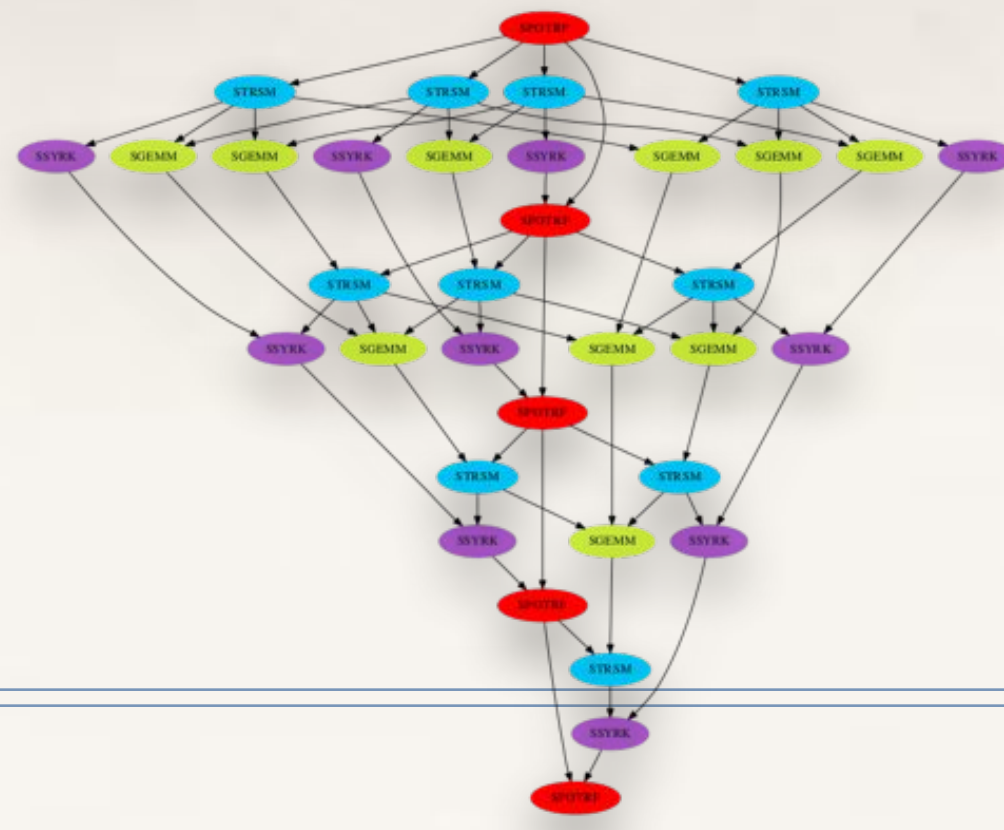
C. Misale, G. Ferrero, M. Torquati, M. Aldinucci.
Sequence alignment tools: one parallel pattern
to rule them all?
BioMed Research International, 2014.

- ❖ Top tools for parallel DNA alignment
- ❖ Hand-tuned C/C++/SSE2 code
- ❖ Spin locks + Pthreads
- ❖ Sequences (reads) are streamed from MMIO files to workers
- ❖ Memory bound

- ❖ FastFlow master-worker
- ❖ Memory affinity, pinning, affinity scheduling (embedded in the pattern)
- ❖ BT: up to 200% speedup
BWA: up to 10% speedup over originals

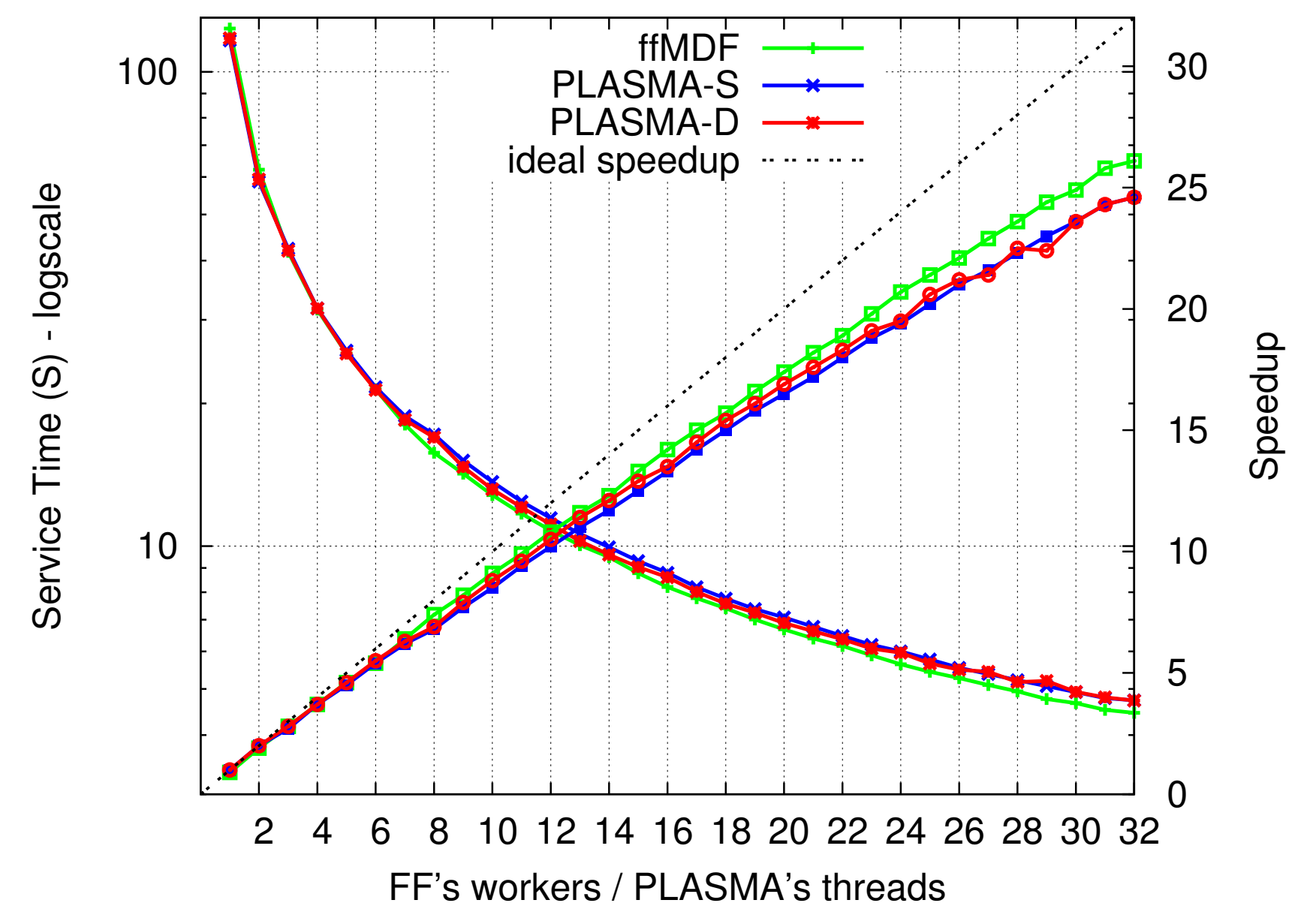
Graphs of tasks

LU and Cholesky

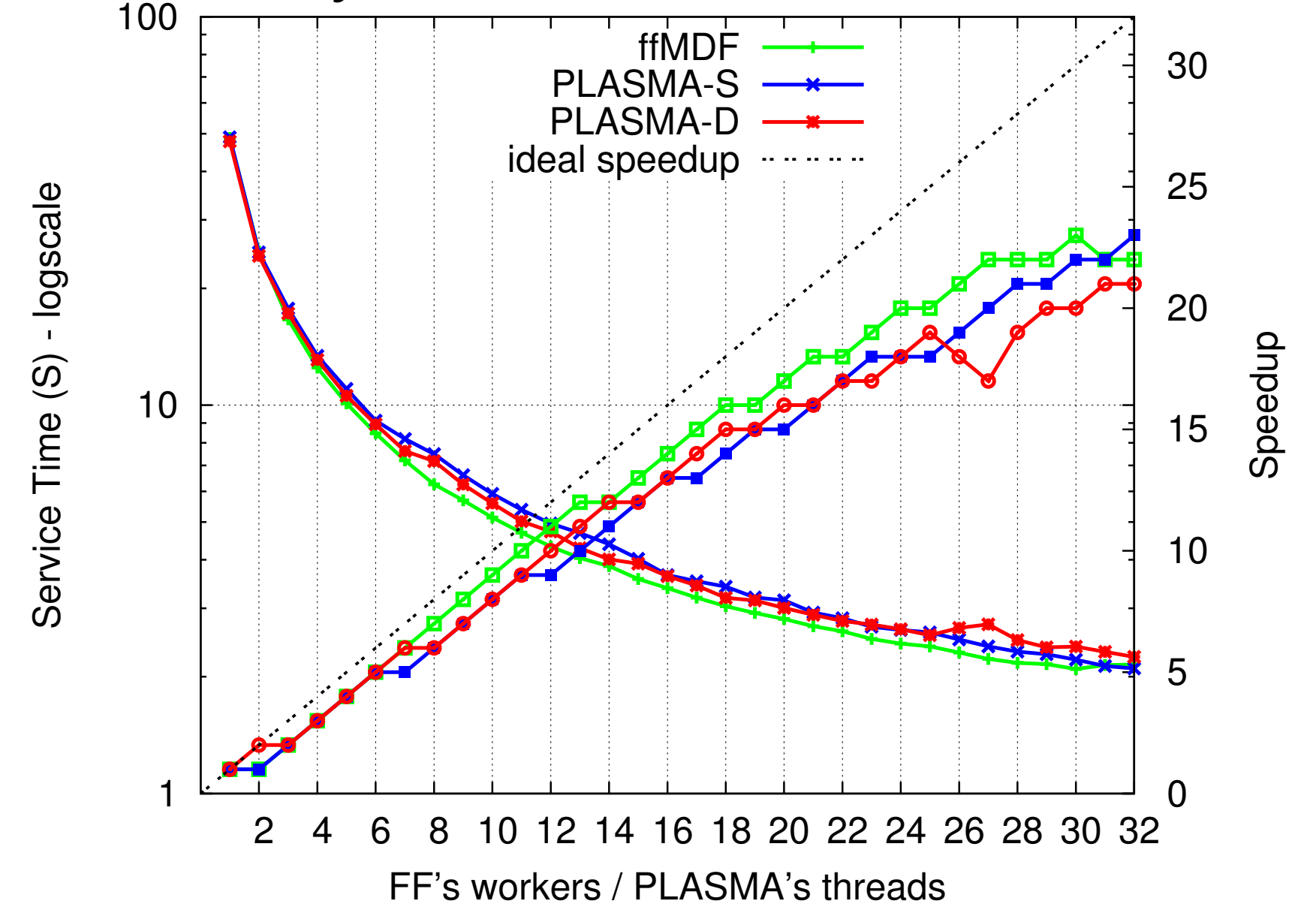


- ❖ Macro-Data-Flow (MDF) pattern encoding dependence DAG
 - ❖ **pipeline**(TaskGen, **farm**(TaskExec))
 - ❖ Configurable scheduling, affinity, ...
- ❖ Dynamic generation of the DAG
- ❖ Comparable or faster than specialised linear algebra frameworks (e.g. PLASMA)
 - ❖ MDF is general, can be used for a wide range of applications, Dynamic Programming, Divide&Conquer, ...

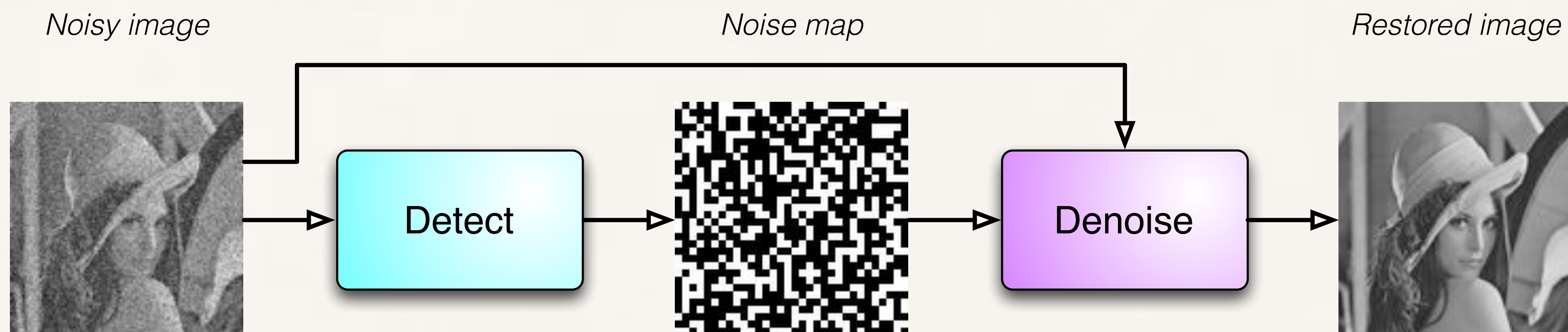
LU size=8192, Nehalem 32 cores



Cholesky size=8192, Nehalem 32 cores

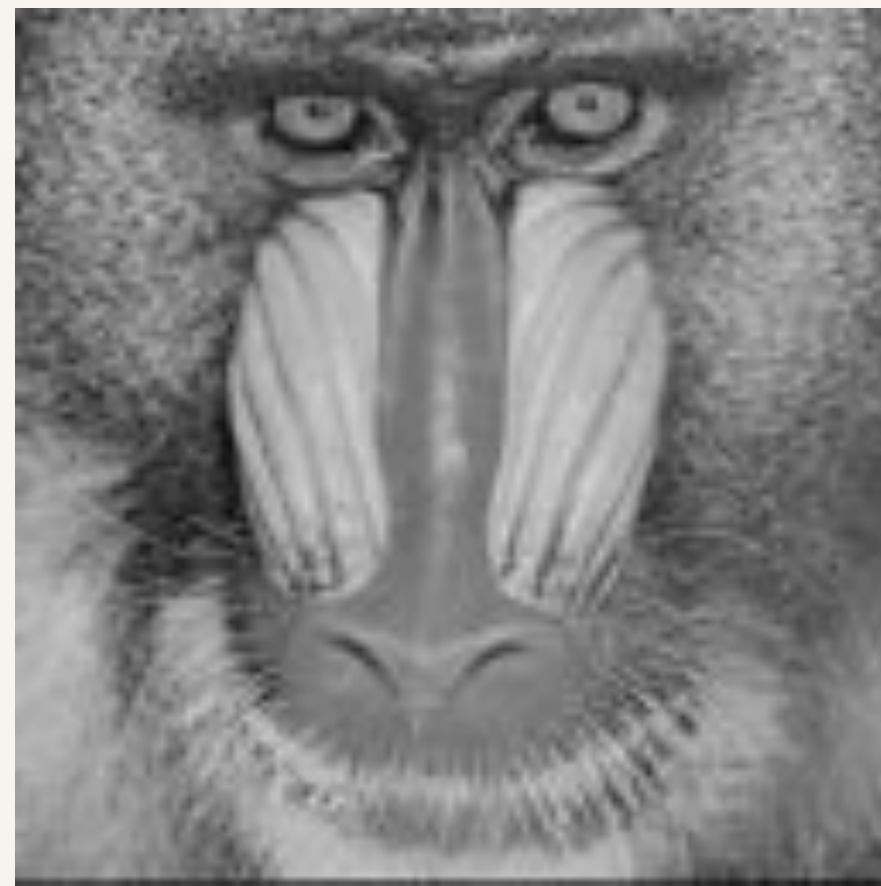


Data Parallelism: Two-stage restoring

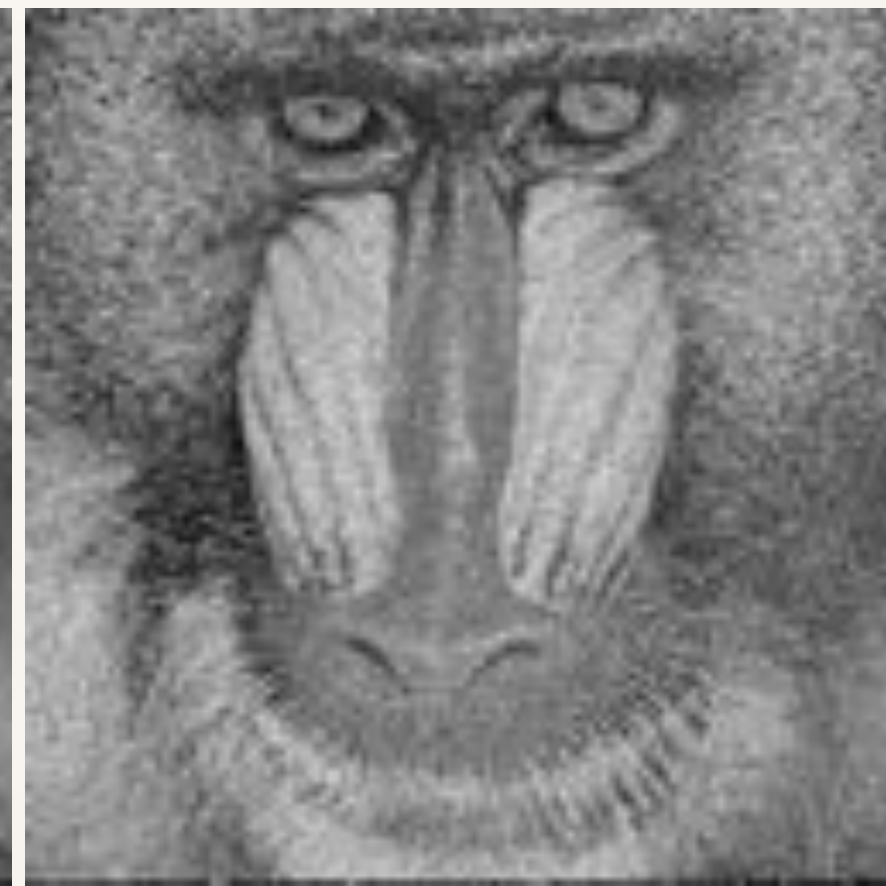


- ❖ **progressive-switching/adaptive median**
 - ❖ neural/bayesian networks, fuzzy, ...
 - ❖ **variational**
 - ❖ statistic
- ❖ **Statistic detection + variational restoration**
 - ❖ High quality, edge-preserving filtering
 - ❖ Much more computational demanding, not really viable without parallelism
 - ❖ Matlab on a single 256x256 image with 50% of noise requires dozen of minutes
 - ❖ **Stages can be pipelined**

Effective noise filtering (variational) Salt-andPepper, Gaussian, ...



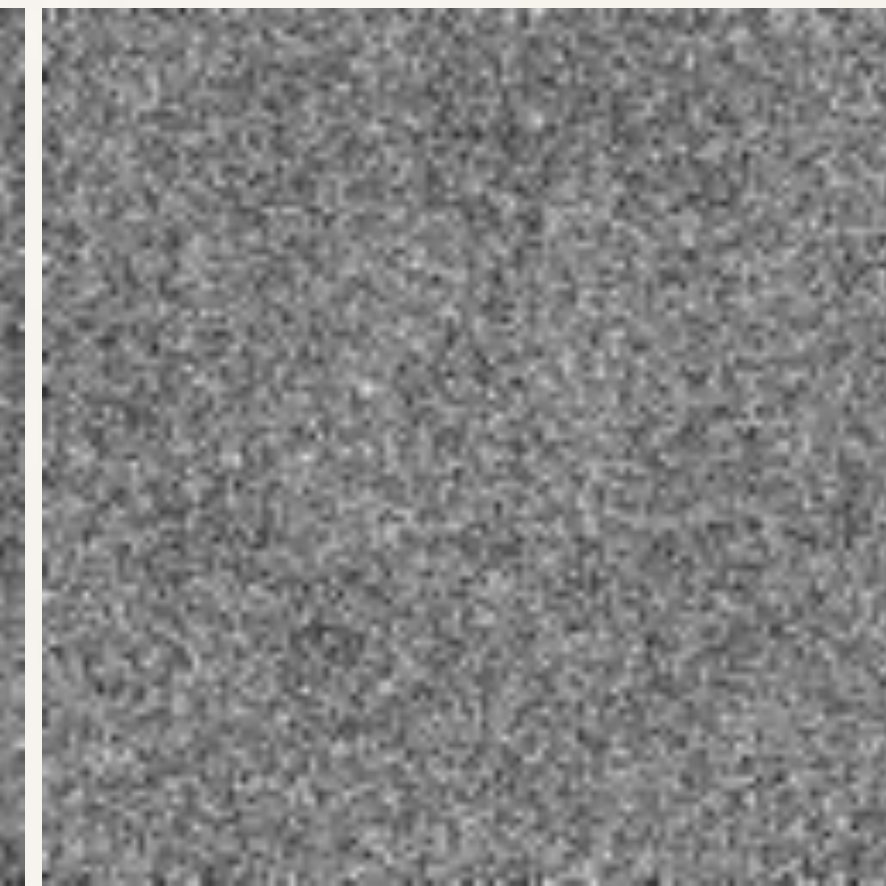
Original
Baboon standard
test image
1024x1024



10% impulsive noise



50% impulsive noise

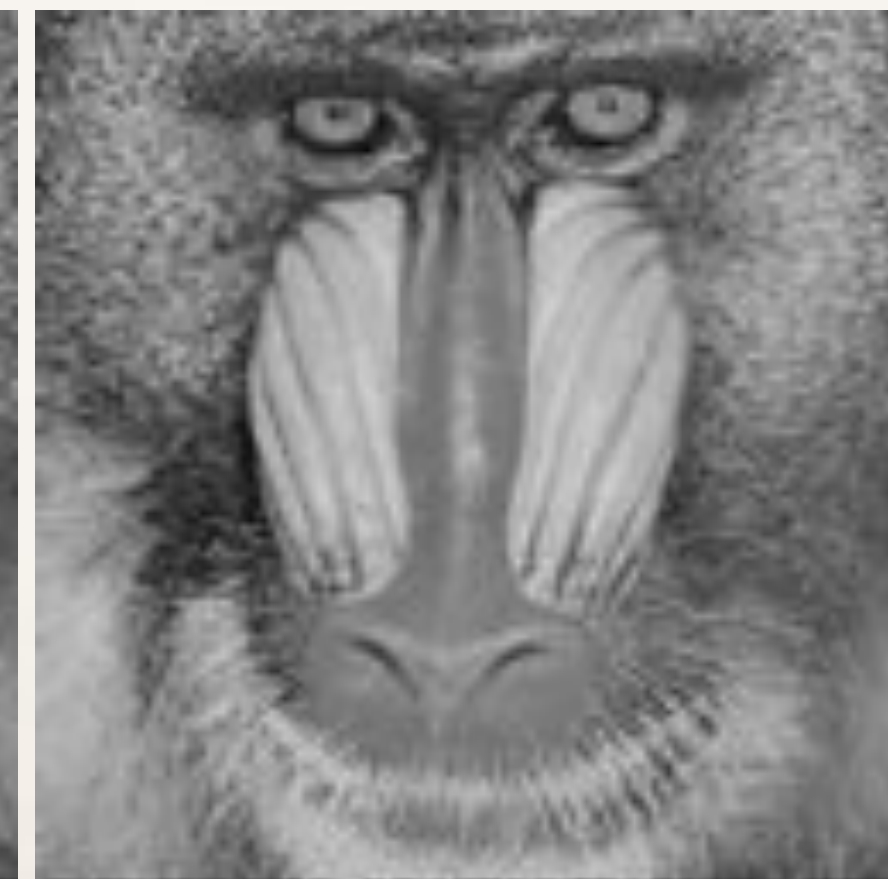


90% impulsive noise

Restored



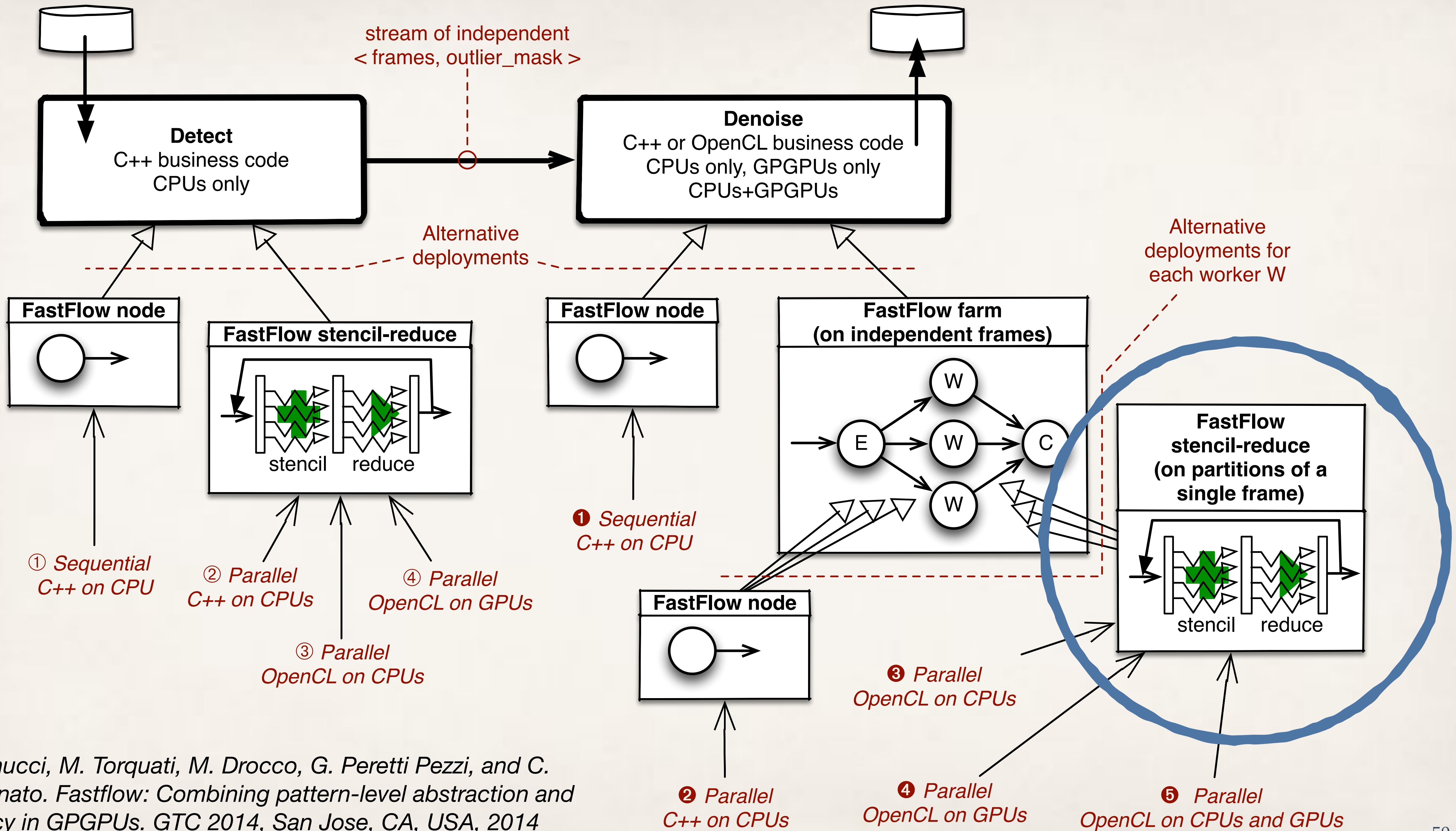
PNSR 43.29dB MAE 0.35



PNSR 32.75dB MAE 2.67



PNSR 23.4 MAE 11.21



M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato. Fastflow: Combining pattern-level abstraction and efficiency in GPGPUs. GTC 2014, San Jose, CA, USA, 2014

```

using namespace ff;

template<typename DenoiserCUDataTaskType, typename DenoiserCUDataMapF>
class cudaDenoiserAUTO: public Denoiser, public FFSTENCILREDUCECUDA(DenoiserCUDataTaskType, DenoiserCUDataMapF, reduceF) {
public:
    cudaDenoiserAUTO(void *kernel_params_, unsigned int height, unsigned int width, bool fixed_cycles, unsigned int max_cycles,
bool trace_time) :
kernel_params(kernel_params_),
    FFSTENCILREDUCECUDA(DenoiserCUDataTaskType, DenoiserCUDataMapF, reduceF)(max_cycles),
    Denoiser(height, width, fixed_cycles, max_cycles, trace_time) {
}

void *svc(void *t) {
    ((denoise_task *)t)->kernel_params = kernel_params;
    ((denoise_task *)t)->fixed_cycles = fixed_cycles;
    return Denoiser::svc(t);
}

void svc_end() {}

unsigned int restore(unsigned char *in, unsigned char *out, int *noisymap, unsigned int *noisy, unsigned int n_noisy, void
*task) {
    unsigned int height = this->height;
    unsigned int width = this->width;
    memcpy(out, in, height * width * sizeof(unsigned char));
    FFSTENCILREDUCECUDA(DenoiserCUDataTaskType, DenoiserCUDataMapF, reduceF)::svc(task);
    return this->getIter();
}

private:
    void *kernel_params;
};
#endif /* CUDADENOISERAUTO_HPP_ */

```

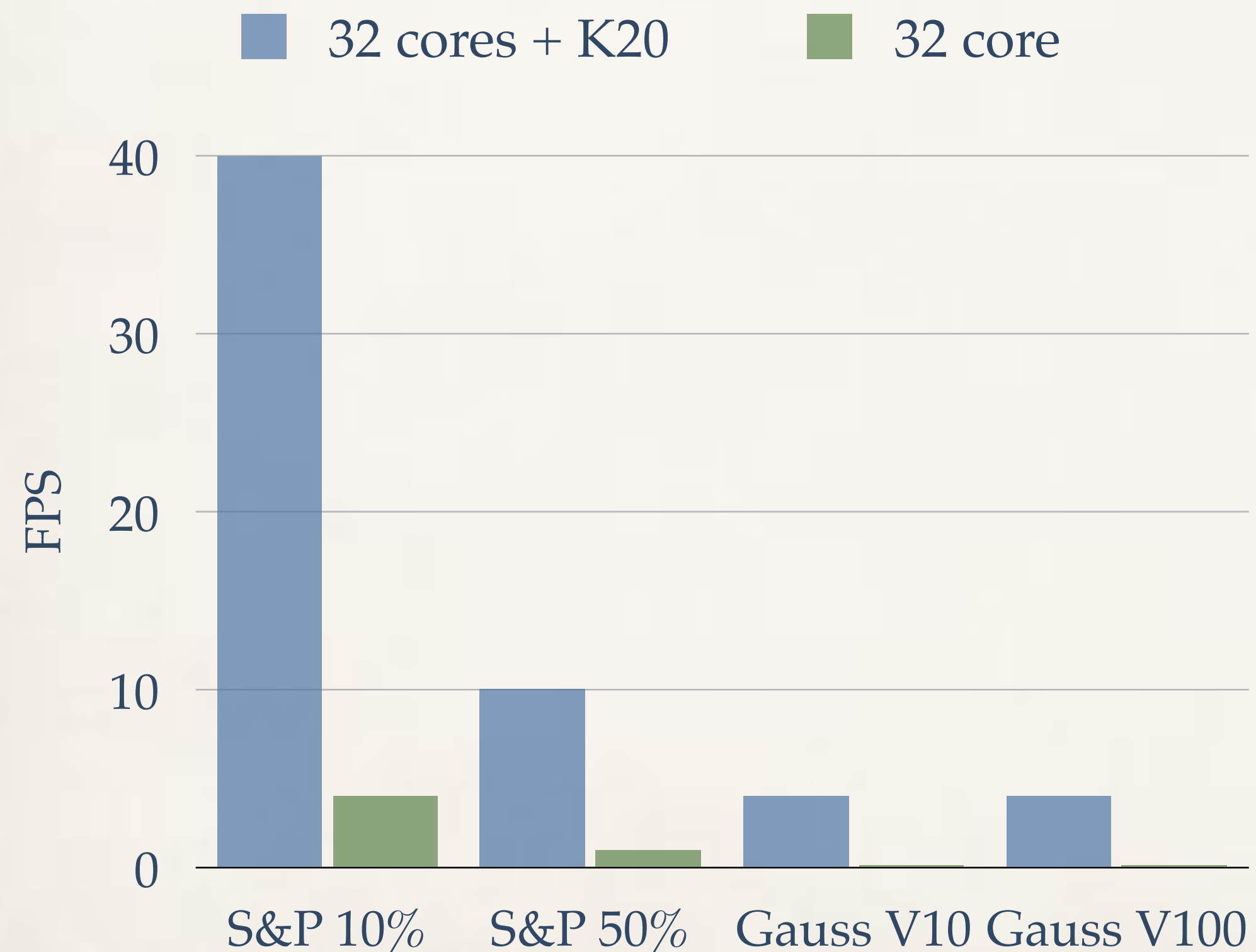
Stencil-Reduce
pattern init

Stencil-Reduce
pattern run

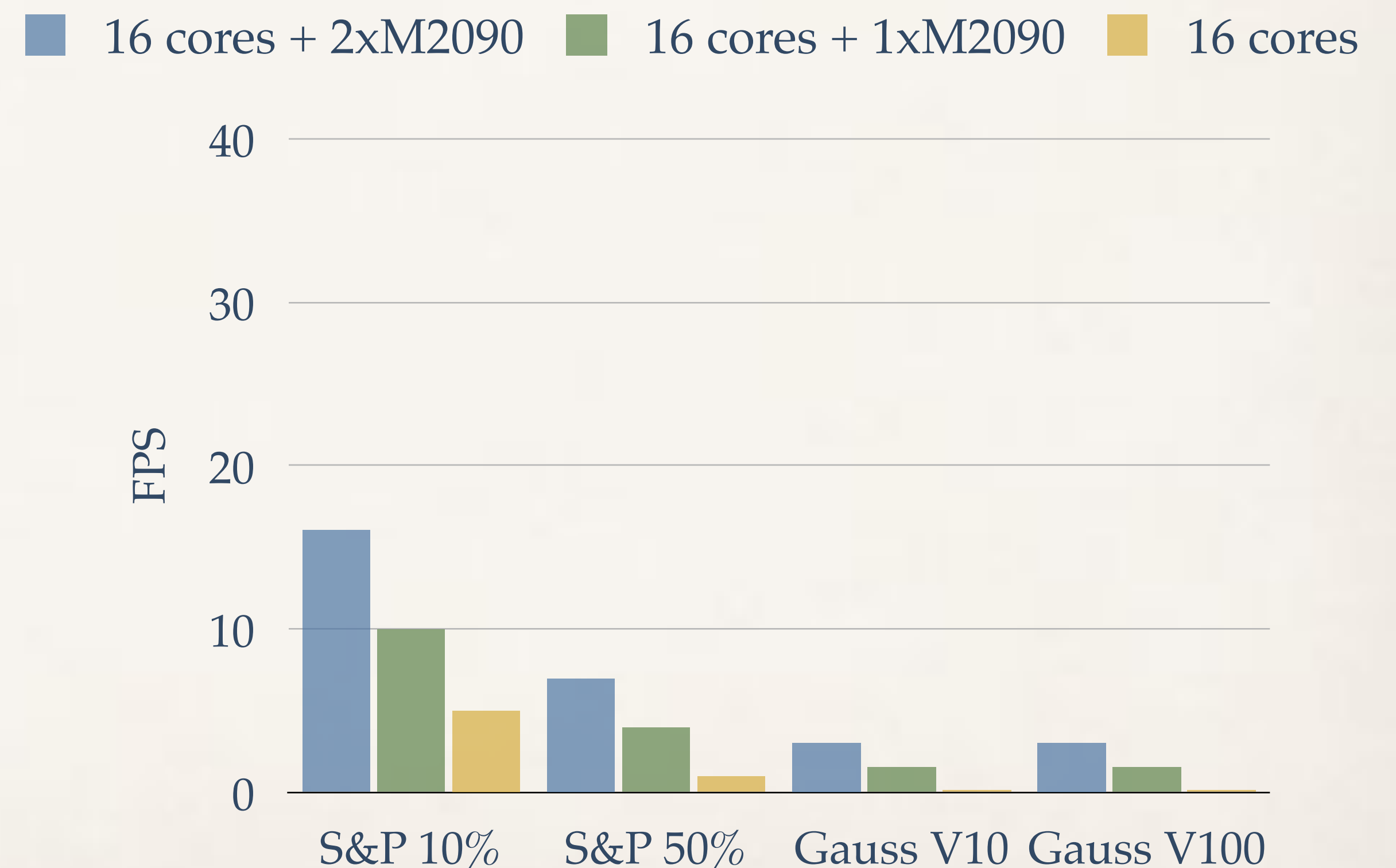
Performance (CPUs + GPGPUs)

Video frames 768x512

Nehalem 32 cores + 1 K20

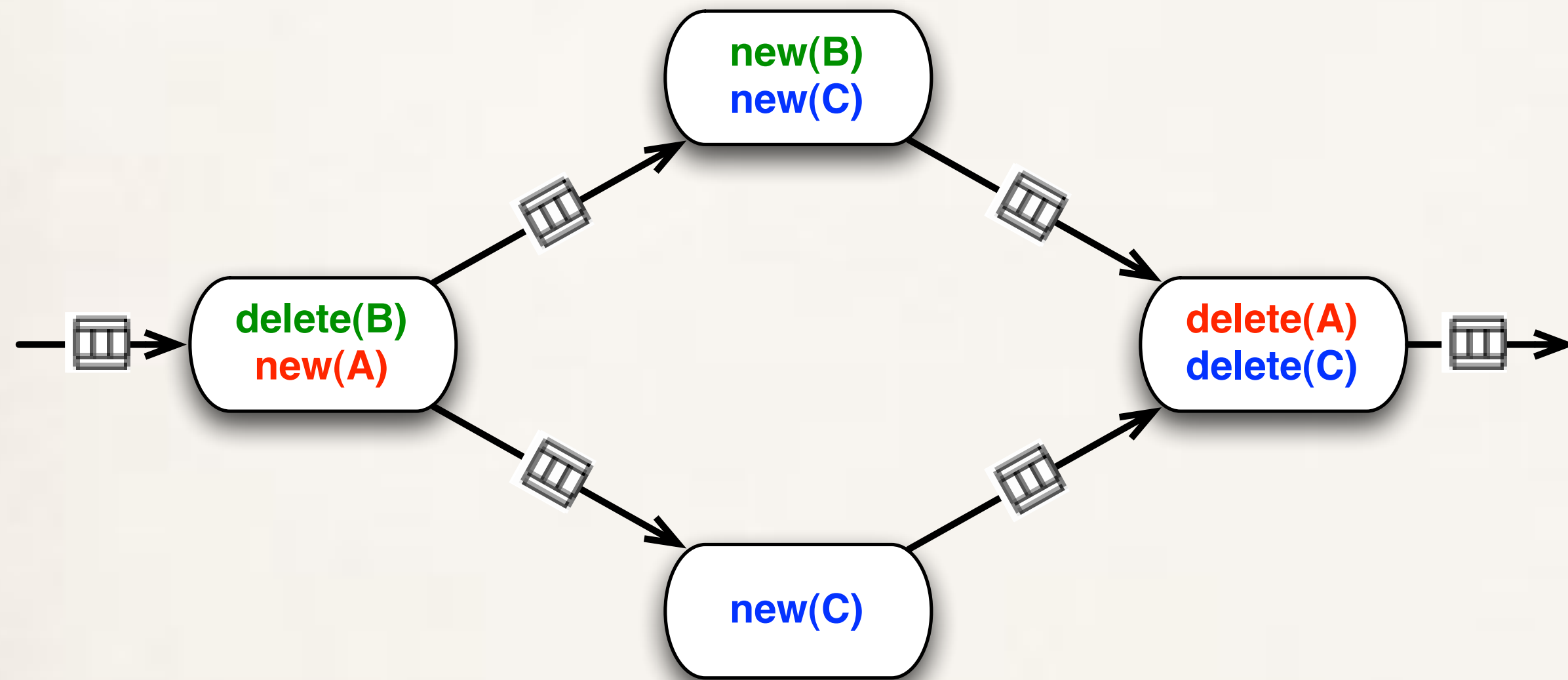


SandyBridge 16 cores + 2 Tesla M2090

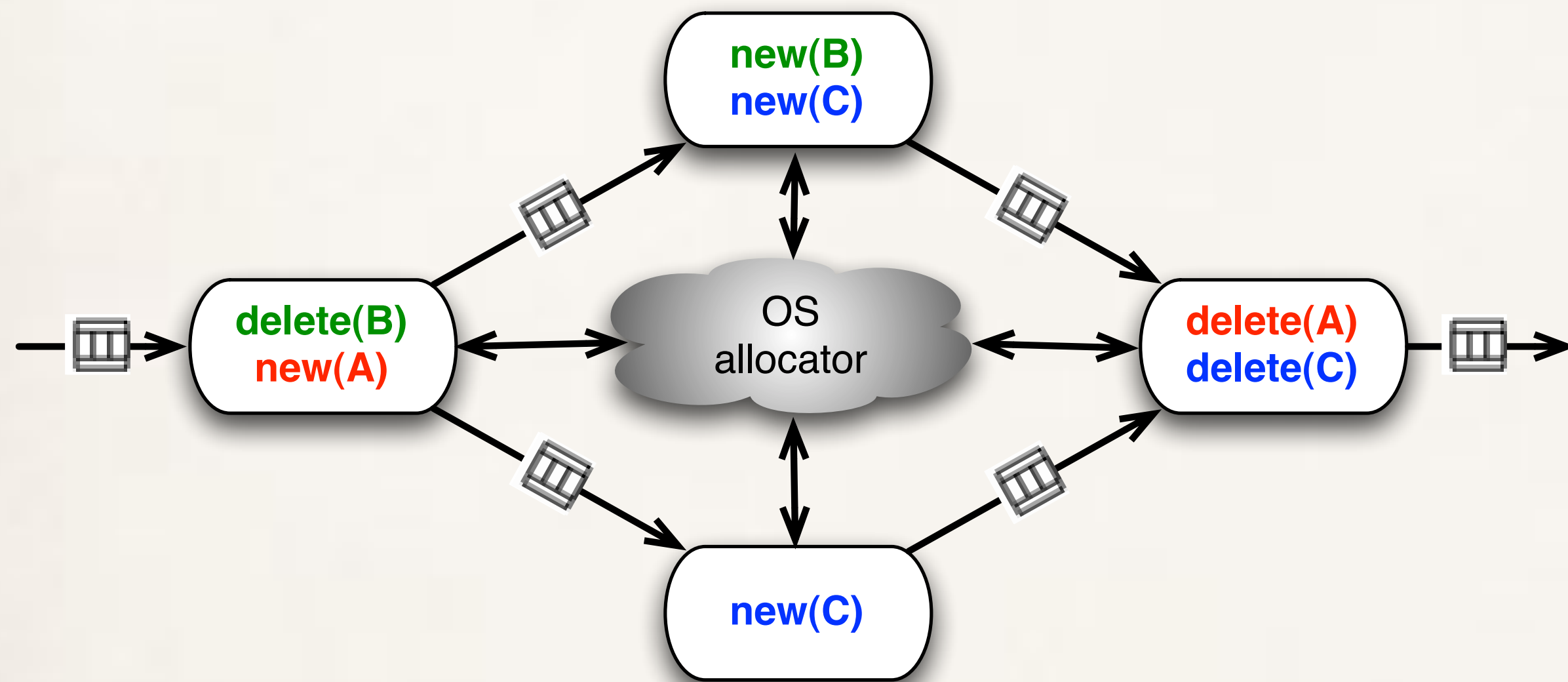


no difference w.r.t. hand-written CUDA code

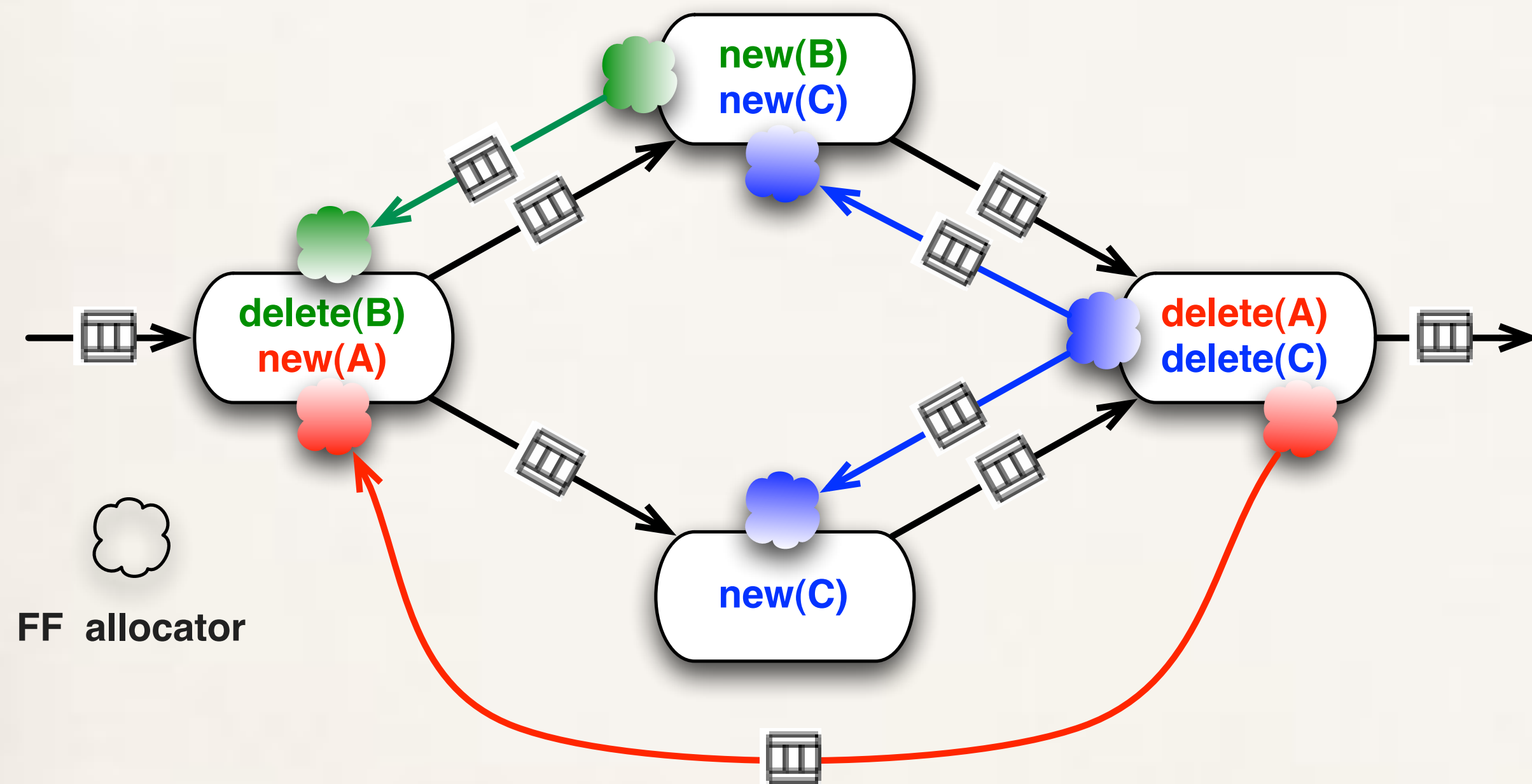
Parallel memory allocation (lock-free)



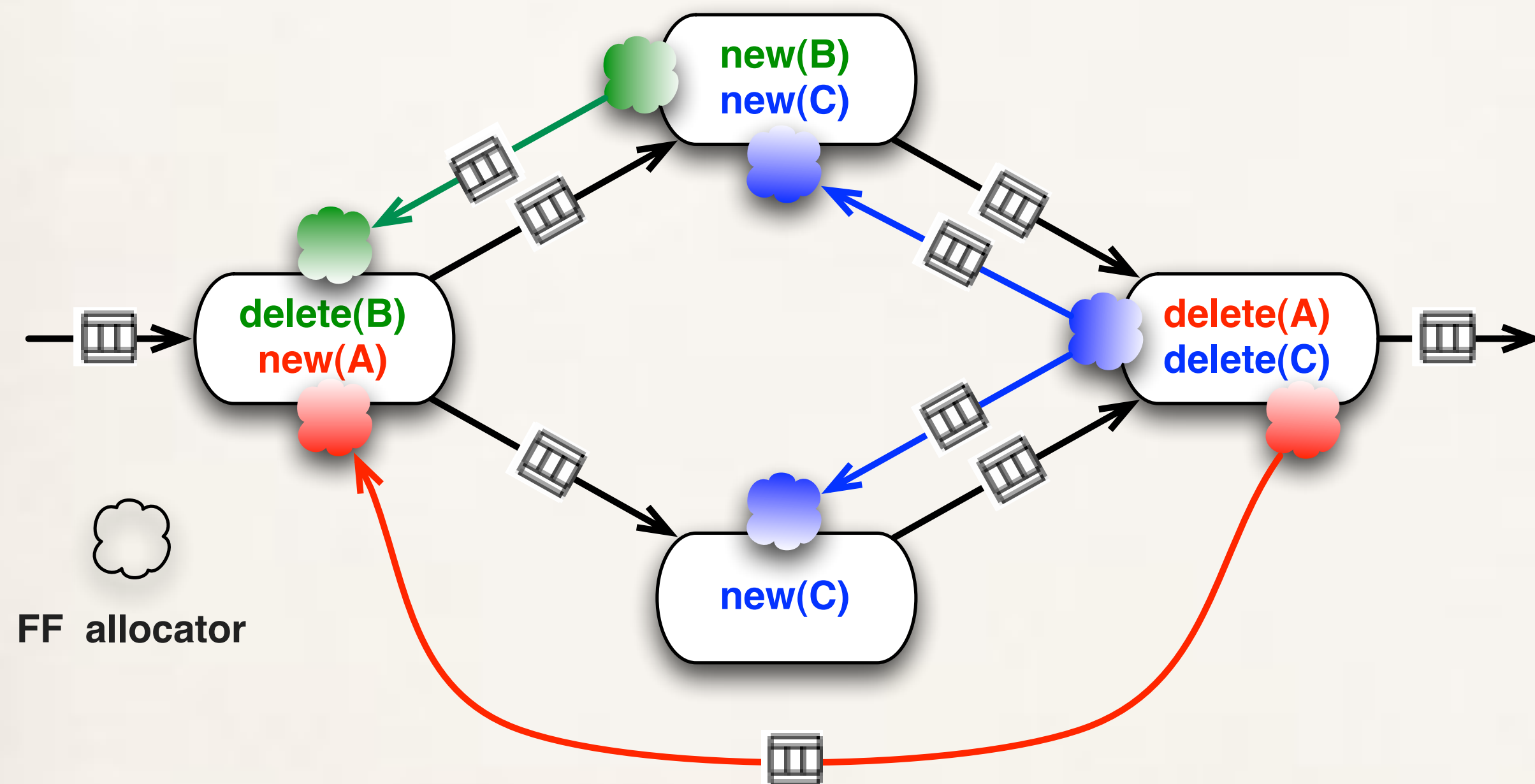
Parallel memory allocation (lock-free)



Parallel memory allocation (lock-free)



Parallel memory allocation (lock-free)



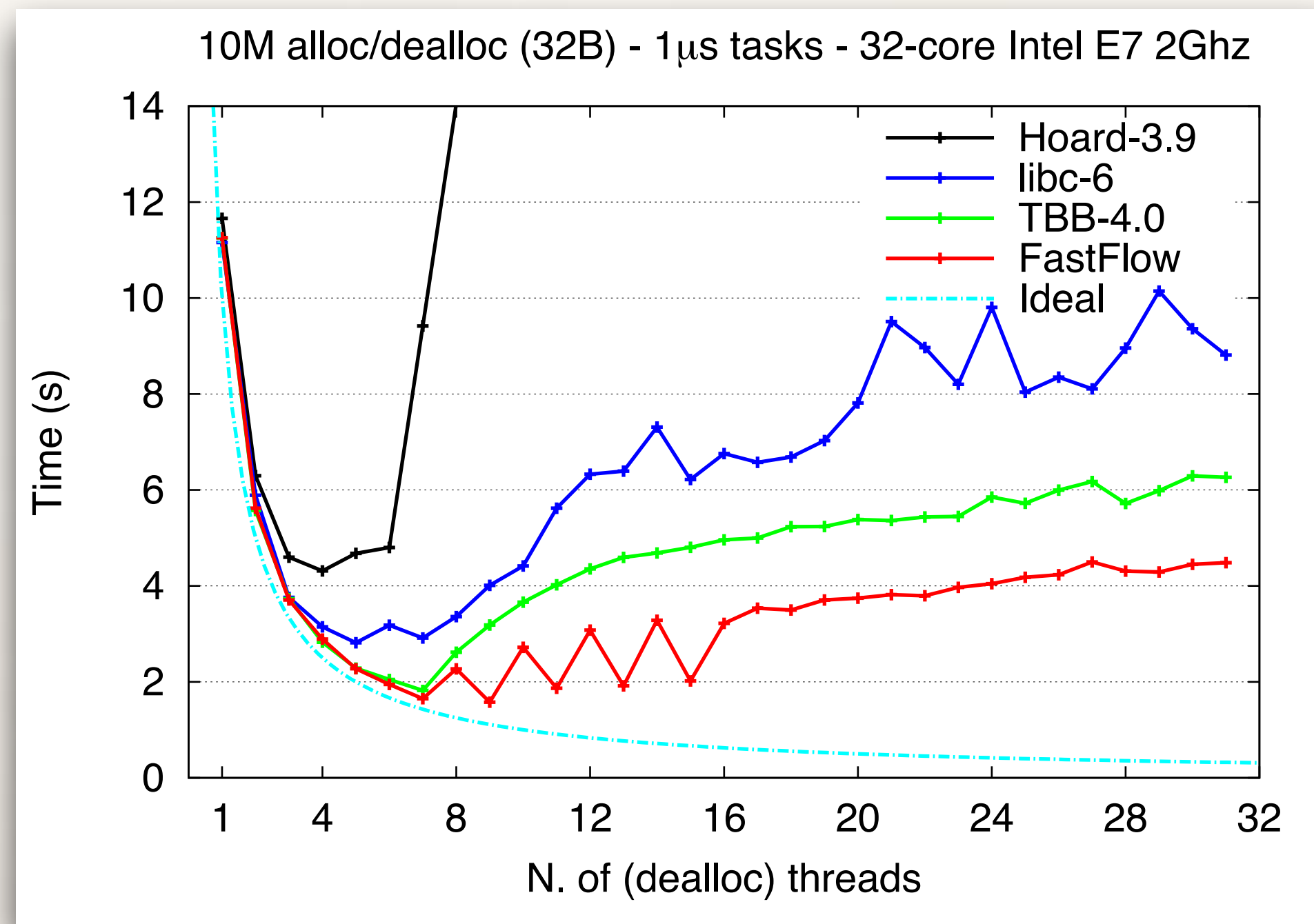
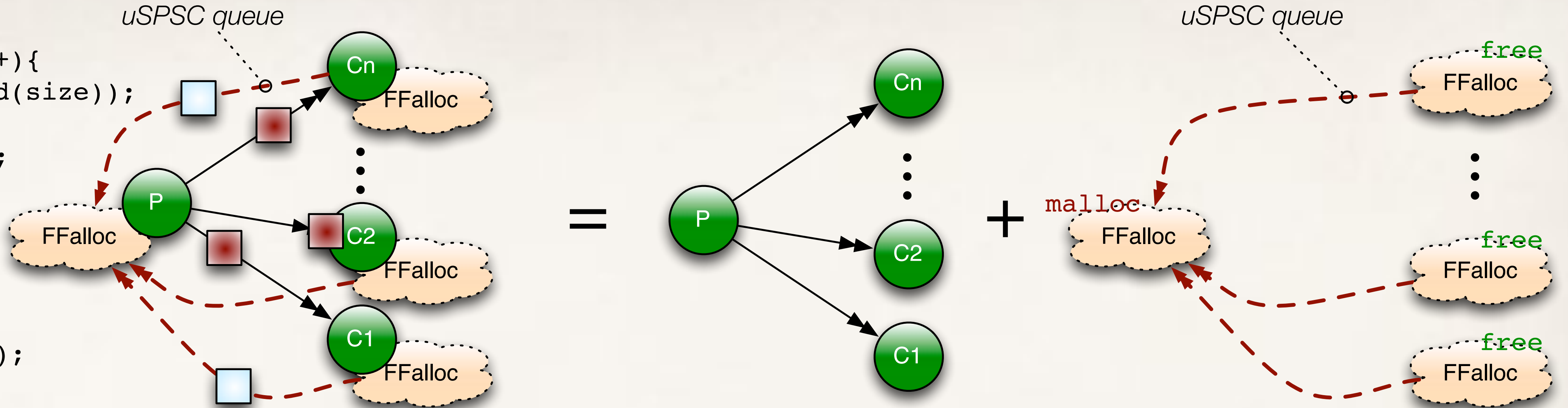
- ❖ A network that connects data allocations-deallocations paths
- ❖ Faster than posix, often faster than hoard and TBB
 - ❖ unpublished, code available on sourceforge
- ❖ Implements deferred deallocation to avoid ABA problem

Producer P:

```
for(i=0;i<10M;i++){
  pi = malloc(rnd(size));
  *pi=...;
  dispatch_RR pi;
}
```

Consumer Ci:

```
while (pi=get())
  do_work(1μs,pi);
  free(pi);
}
```



Conclusion

- ❖ Low-level approach for a better performance / scalability
 - ❖ It is already a myth, a medium scale. Does someone still believe assembler is faster than C++?
 - ❖ Not proved, but we believe it will be even more evident at the large scale (exa-scale)
- ❖ FastFlow: header-only C++11 library
 - ❖ Research framework, portable everywhere exists a C++11 compiler, tiny codebase
 - ❖ Efficient, scalable
- ❖ A data-centric parallel programming model is paramount
 - ❖ High-level with a clear parallel semantics, compositional, enhancing locality and fast data movements

Think different.

Stay foolish, play with cars

Stay hungry, keep looking to new toys



Thanks



University of Turin



M. Aldinucci



G. Peretti



A. Secco



F. Tordini



M. Drocco



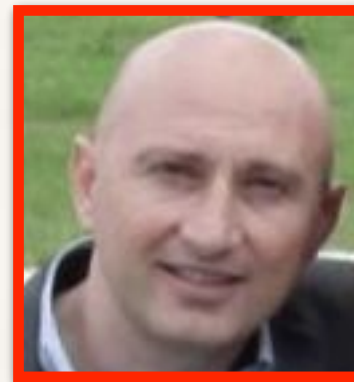
C. Misale



University of Pisa



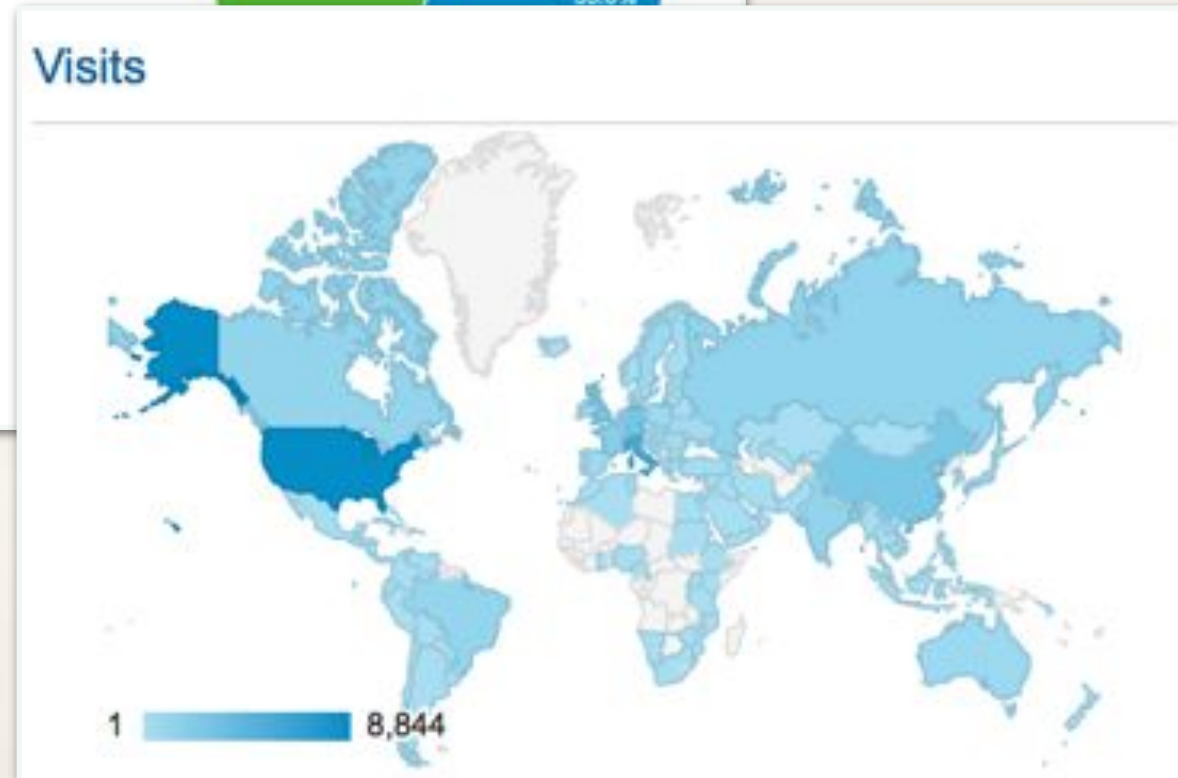
M. Danelutto



M. Torquati



P. Kilpatrick



EU-FP7 - 3.5M€



Unito - 440K€



EU-FP7 - 3.7M€



EU-FP7 Network of Excellence

<https://sourceforge.net/projects/mc-fastflow/>