



UNIVERSITÀ DEGLI STUDI
DI TORINO

DI TORINO
UNIVERSITÀ DEGLI STUDI

Politecnico di Milano
Dipartimento di Elettronica e Informazione

December 5, 2012
Milano, Italy

FastFlow: high-level programming patterns with non-blocking lock-free run-time support

Marco Aldinucci - Uni. of Torino, Italy

Coordinator of the research activities on "Parallel and High-Performance Computing"

Fabio Tordini (PhD), Claudia Misale (PhD), Irfan Uddin (PostDoc), Guilherme Peretti Pezzi (PostDoc), Maurizio Drocco (SW engineer)

&

Massimo Torquati and Marco Danelutto - Uni. Pisa, Italy

Massimiliano Meneghin - IBM Research, Ireland

Peter Kilpatrick - Queen's Uni. Belfast, U.K.



- * Concurrency and multi-core, the theoretical background

- ♦ a personal perspective

- * FastFlow

- ♦ A programming model (and a library) for multicore (& manycore)
- ♦ Fast core-to-core lock-free messaging

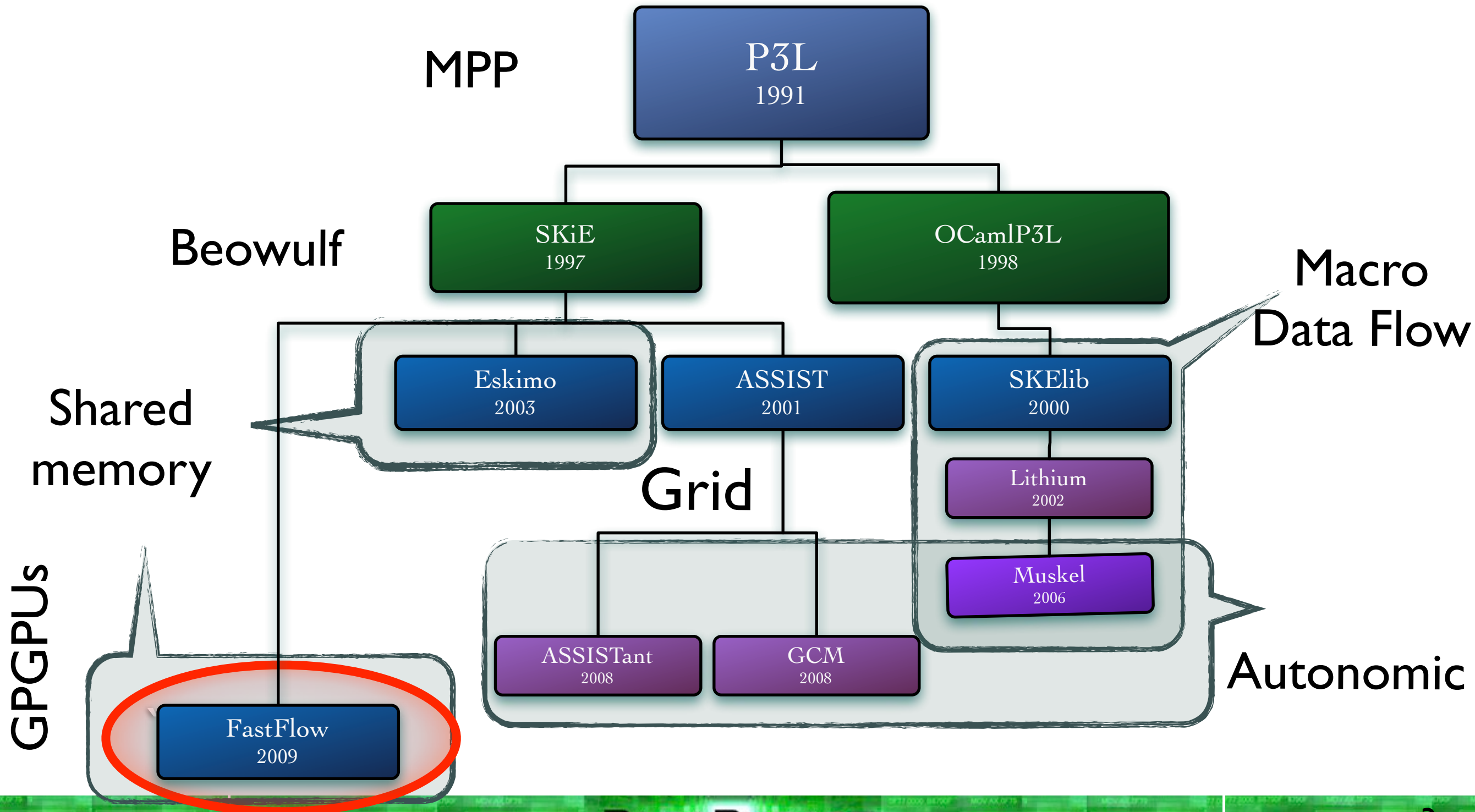
- * Applications

- * Discussion

Our tool perspective

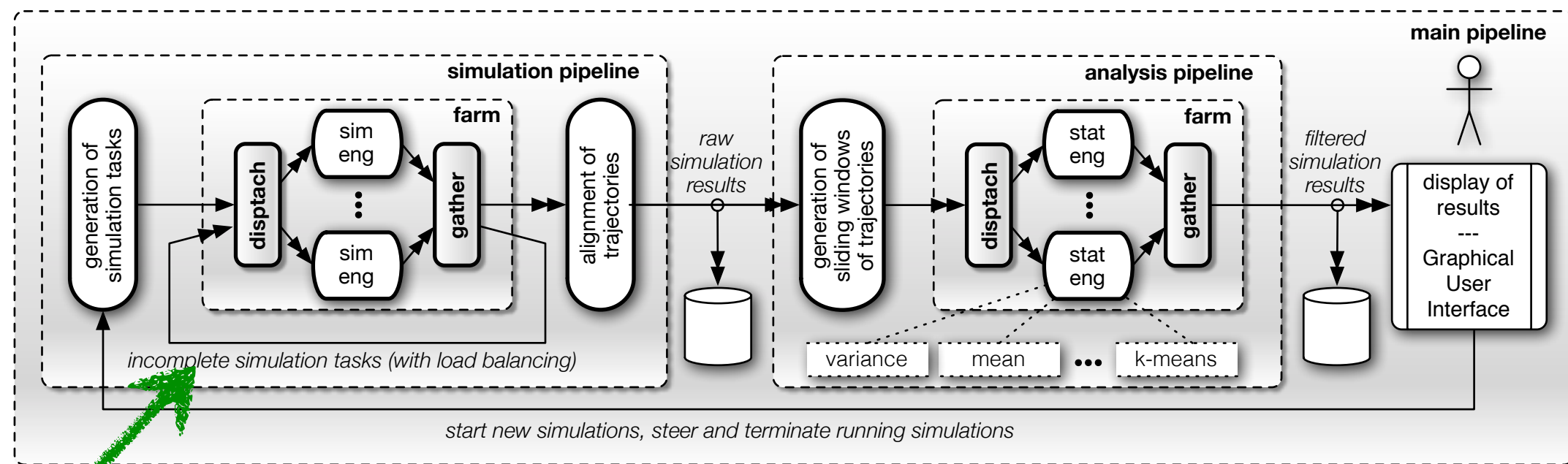


UNIVERSITÀ DEGLI STUDI
DI TORINO





Concurrency and multi-core theoretical background: a personal perspective



DSL task engine

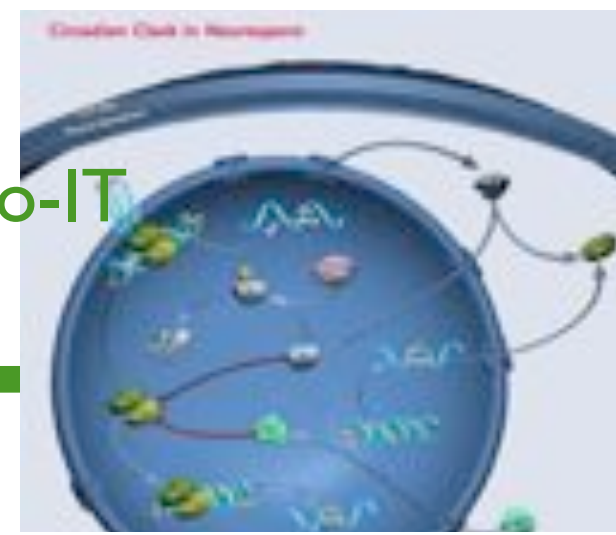
task size

Table 2 - Performance (Intel 32 core platform)

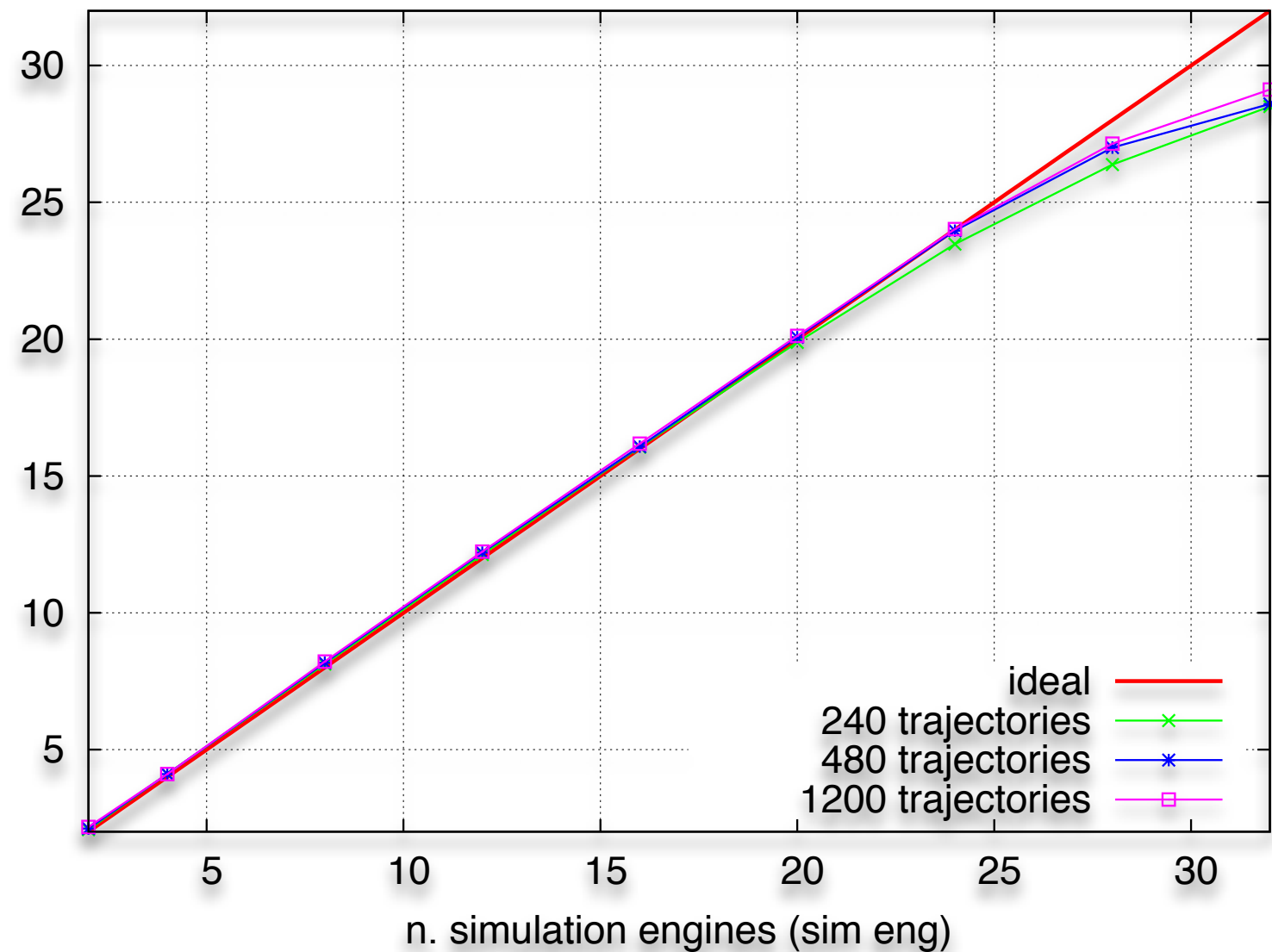
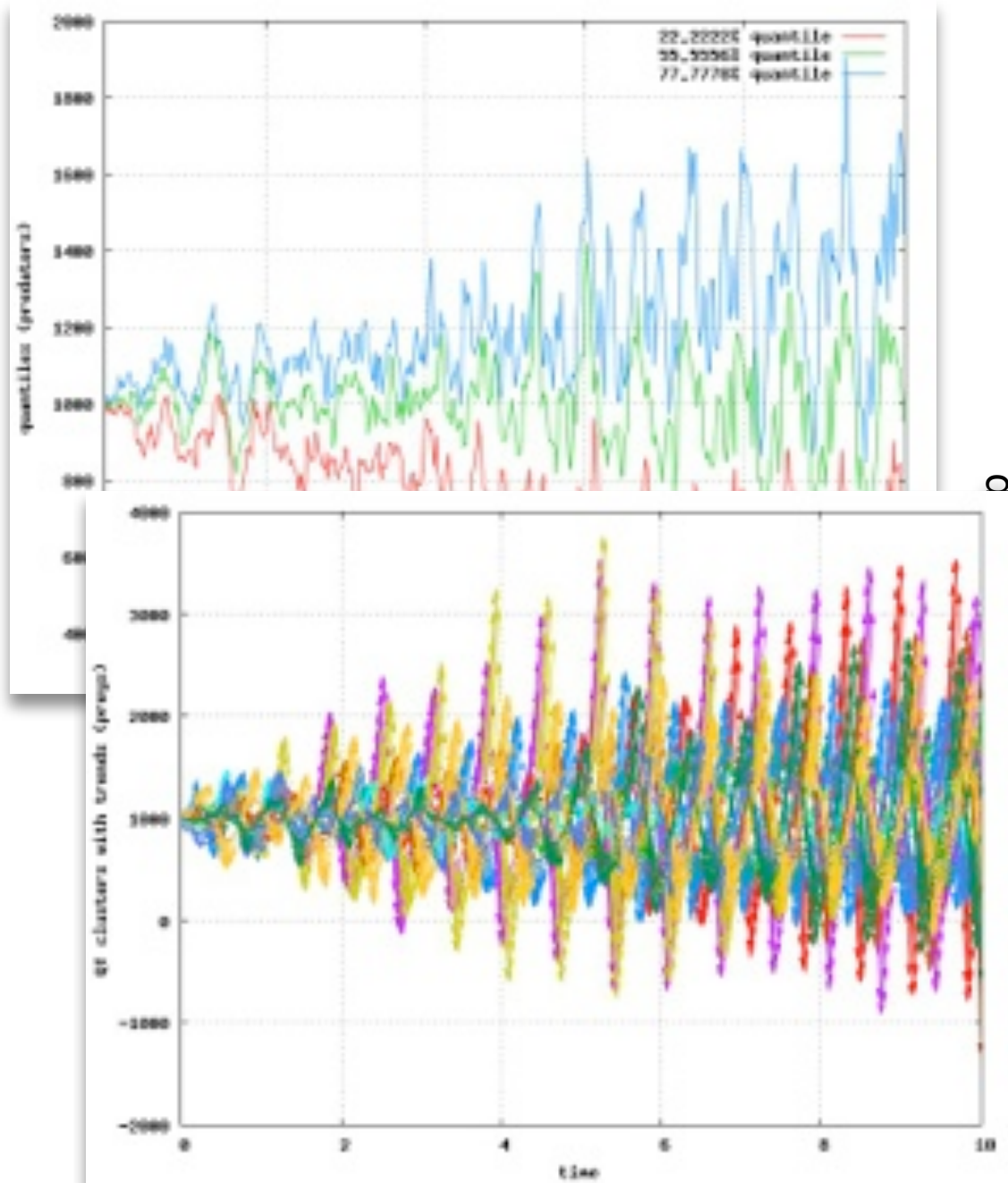
Model	Single trajectory information			Overall data (20 sim eng, 3 stat eng)		
	N. samples	Avg sim step	Sample time	Inter-arrival time	Throughput	Output size
Neurospora	10^4	7.80 μ s	517.24 μ s	25.86 μ s	11.87 MB/s	36.62 MB
Neurospora	10^5	8.37 μ s	55.51 μ s	2.78 μ s	11.98 MB/s	366.21 MB
Neurospora	10^6	75.63 μ s	4.65 μ s	232.68 ns	201.63 MB/s	3.58 GB
EColi	10^6	173.64 μ s	0.58 μ s	28.81 ns	257.66 MB/s	4.47 GB
Lotka-Volterra	10^6	22.86 μ s	0.69 μ s	34.68 ns	147.11 MB/s	2.68 GB

Parallel stochastic sim for system biology

IEEE PDP 2011, HiBB 2011, Briefings in Bioinformatics (invited), Bio-IT world (invited), IEEE PDP 2013 (submitted), BMC Bioinformatics



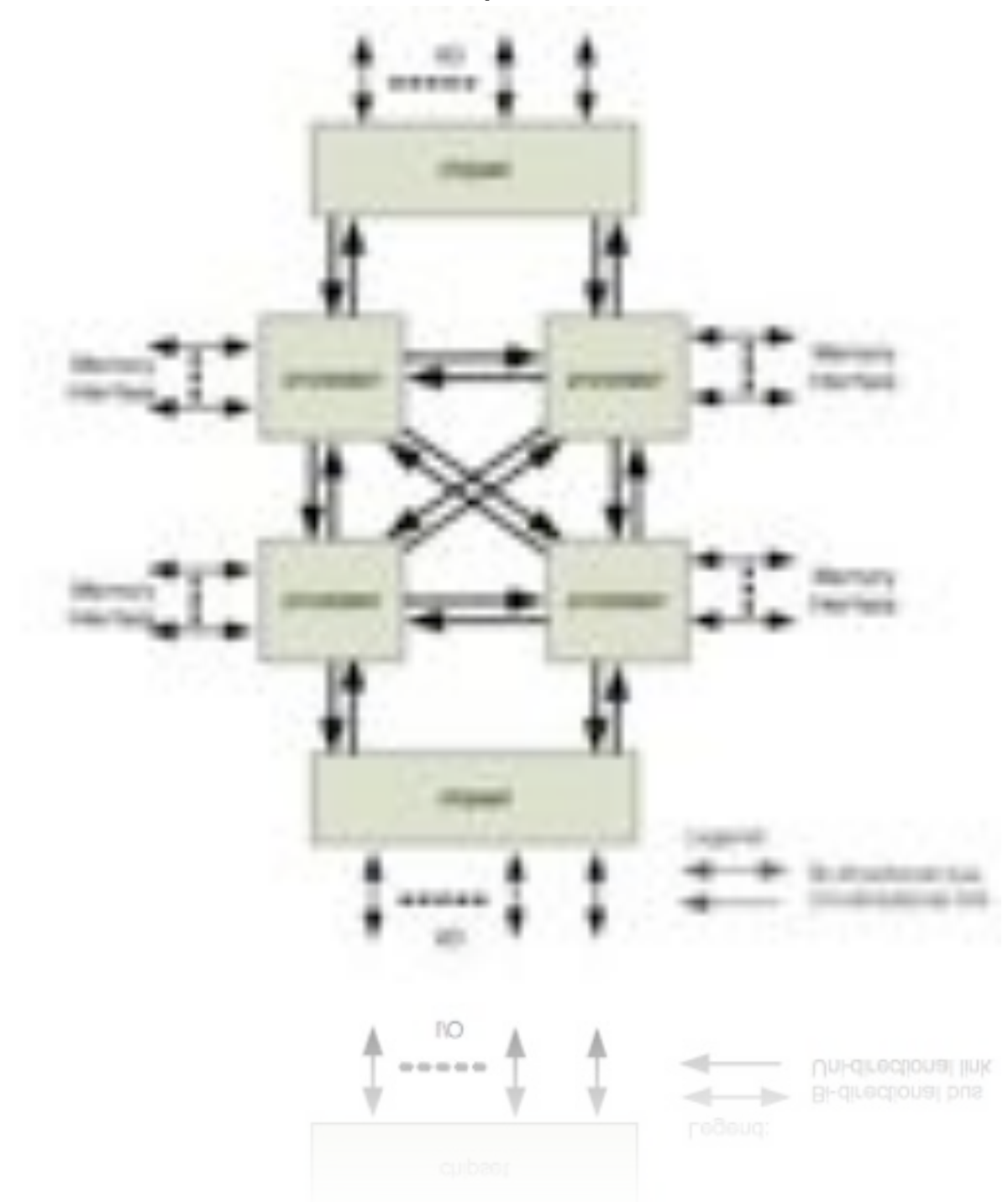
Simulation of transcriptional regulation in Neurospora



Nowadays

- * E.g. Intel Ivybridge, Haswell
 - * cache-coherent
 - * 12 or more core per socket (20 contexts)
 - * **cc-NUMA** (as matter of a fact)
- * NVidia/AMD GPGPU/Hybrid
 - * SIMD, no global synch
 - * performance only with proper and **not fully automatic** memory hierarchy management
- * Intel MIC CPU/GPGPU
 - * ring-based interconnection, variable coherency
 - * apparently even more NUMA
- * ~~IBM powerEN~~
 - * general purpose cores
 - * specialised cores, soft cores?

Quickpath



From programming/tuning viewpoint ... the simplest is already too complex ...

* Exploit cache coherence

- ◆ Memory fences are expensive, increasing core count will make it worse

* Fine-grained parallelism is hard to achieve

- ◆ I/O bound problems, High-throughput, Streaming, Irregular DP problems
- ◆ Automatic and assisted parallelisation solves uniform&easy cases

* SIMD/GPGPU worsen the scenario

- ◆ Atomic ops in memory (i.e. fences) are still needed
- ◆ Not everything can be described with do independent (a.k.a. map)

Micro-benchmarks: farm of tasks



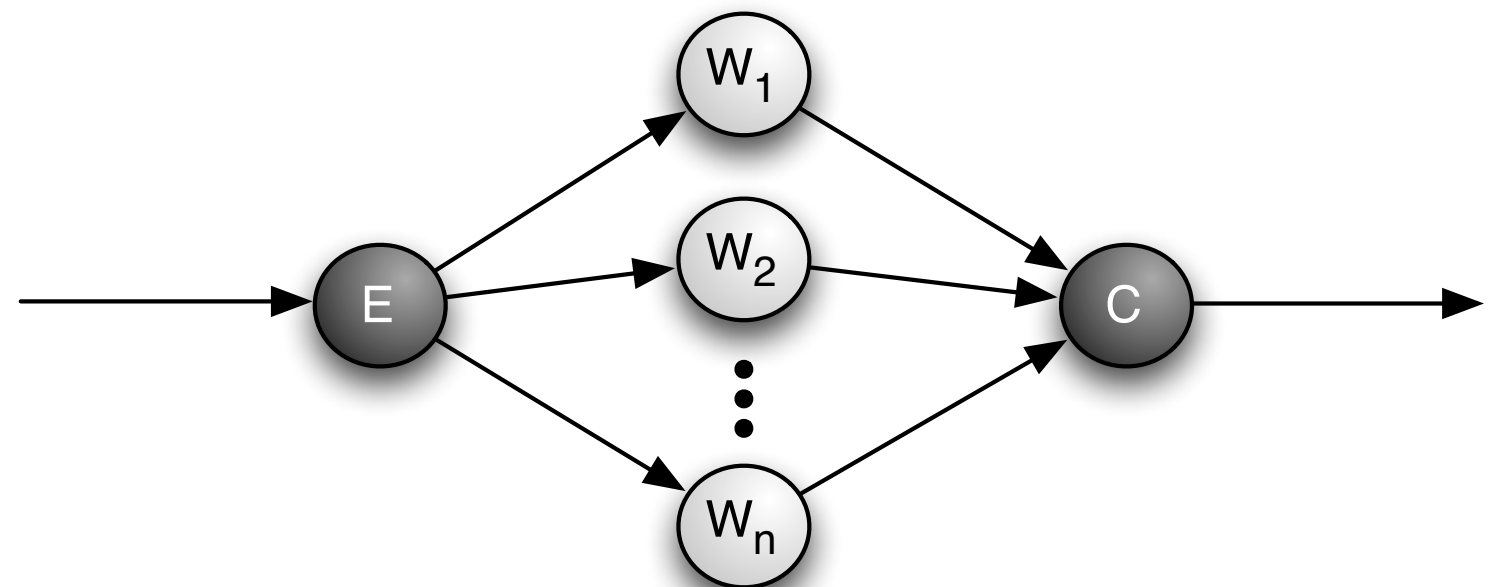
UNIVERSITÀ DEGLI STUDI
DI TORINO

Used to implement: parameter sweeping, master-worker, etc.

```
void Emitter () {  
    for ( i =0; i <streamLen;++i){  
        task = create_task ();  
        queue=SELECT_WORKER_QUEUE();  
        queue ->PUSH(task);  
    }  
}
```

```
void Worker() {  
    while (!end_of_stream){  
        myqueue ->POP(&task);  
        do_work(task) ;  
    }  
}
```

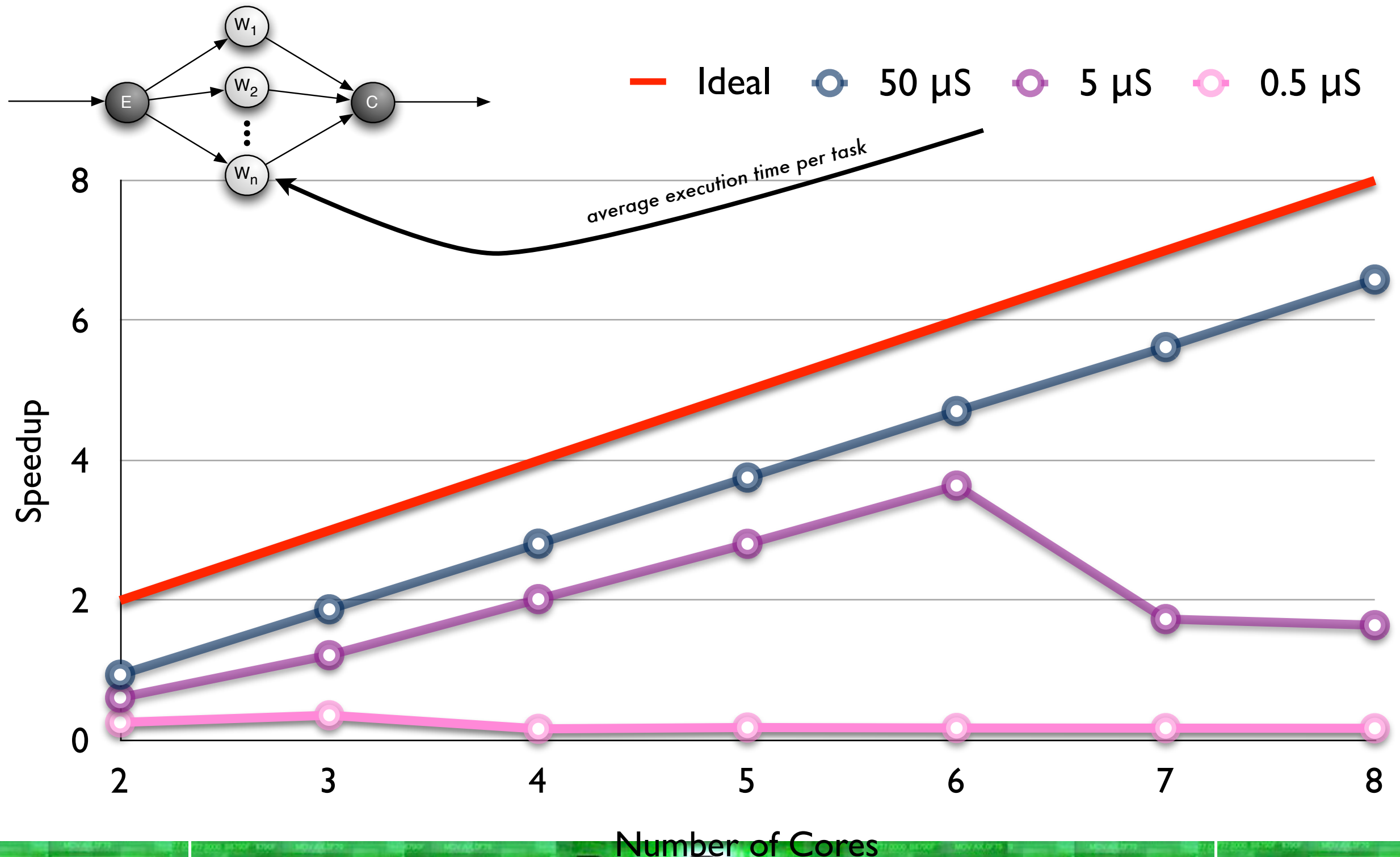
```
int main () {  
    spawn_thread( Emitter ) ;  
    for ( i =0; i <nworkers;++i){  
        spawn_thread(Worker);  
    }  
    wait_end () ;  
}
```



Task farm with POSIX lock/unlock



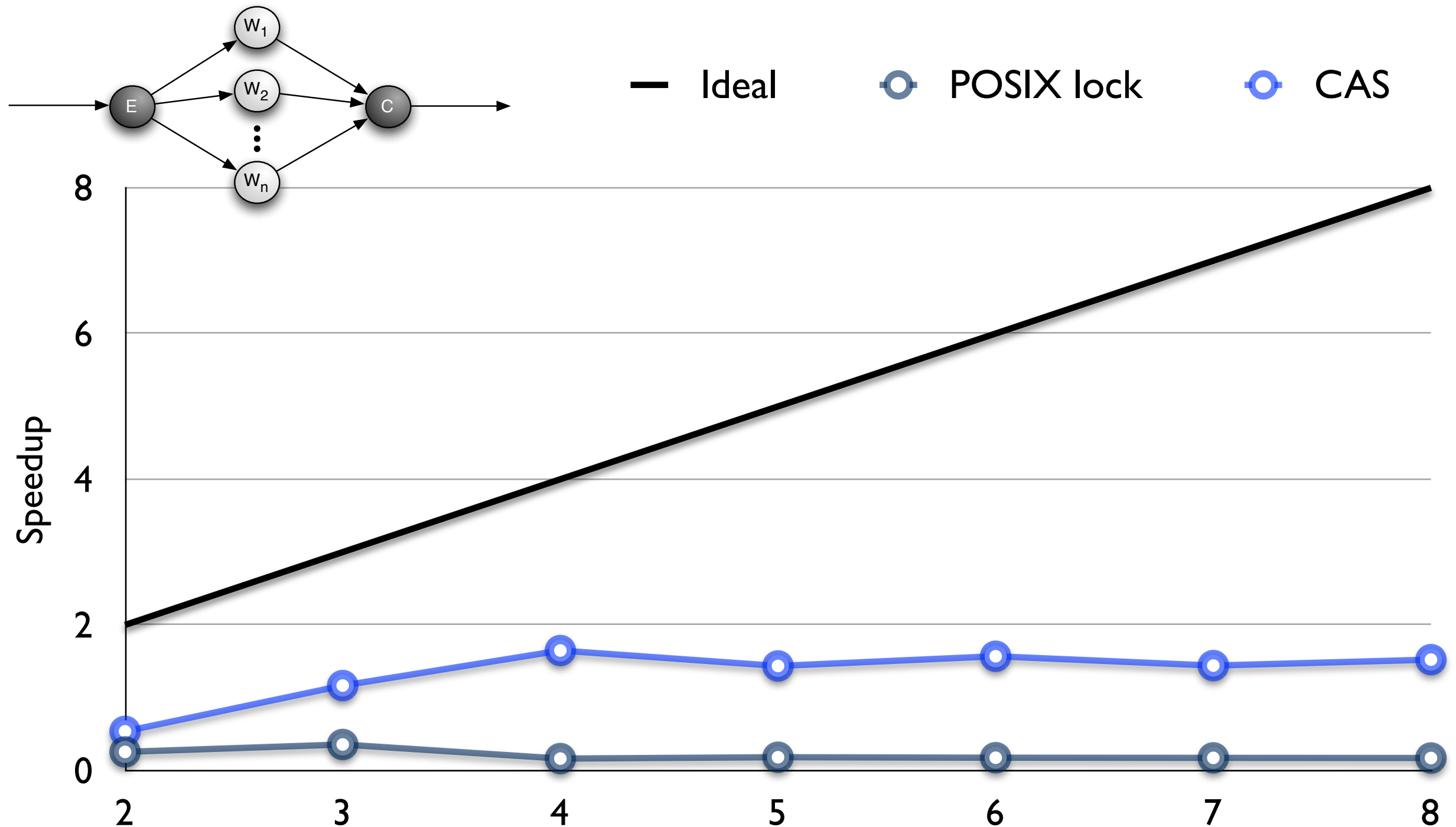
UNIVERSITÀ DEGLI STUDI
DI TORINO



Can we avoid locks?

- * Under relaxed memory models, using CAS/RW-ops
 - ◆ nonblocking algorithms
 - ◆ they perform better than lock-based
 - ◆ they fence the memory and pay cache coherency reconciliation overhead
 - ◆ in GPUs ...
 - CAS/atomic ... you have to go to the global memory

Lock vs Nonblocking CAS (fine grain 0.5 μ S)

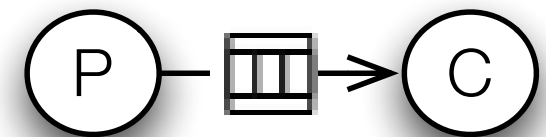


Re-starting from the basics

* Reducing the problem to the bare bones

- ◆ Producer-Consumer model (streaming)
- ◆ Directly control thread blocking using non-blocking synchronisations
- ◆ Directly design the “data channel”
 - Having clear how data move in the whole memory hierarchy

* Restarting from the FIFO queue



Producer-Consumer

* Producer-Consumer queues

- ♦ fundamental data structures in concurrent systems
 - data/message channels synchronization, task scheduling, ...
 - work-stealing mechanisms (e.g. for OpenMP runtime)

* Producer-Consumer vs Mutual Exclusion

- ♦ Mutex is inherently more complex (requires deadlock-freedom)
 - require interlocked ops (CAS, ...), that induces memory fences, thus cache invalidation
 - Dekker and Bakery algorithms requires Sequential Consistency
 - Producer Consumer is a cooperative (non cyclic) process

* Producer-Consumer vs Transactional Memories (?)

- ♦ To be tested extensively, interesting to understand what happens when data is moved to another core (get an invalidation?)
- ♦ Transactions happens at cache line level (IBM/BlueGene) or blocking decode unit (IBM/PPC x86_64/ring0 wait on reservation)

Concurrent queues

* Concurrency level

- ◆ SPSC, SPMC, MCSP, MPMC

* Internal data structures

- ◆ Array-based, List-based

* Size

- ◆ Bounded, Unbounded

* Progress guarantees

- ◆ No guarantee (blocking), Obstruction freedom, Lock freedom, Wait freedom

Blocking vs non-blocking

* What are the performance implications of the progress properties ?

* For medium/coarse grain applications:

♦ Blocking **faster** than Non-Blocking

several task-based approaches are here

* For fine grain applications:

♦ Non-Blocking **faster** than Blocking

♦ Obstruction-Free **faster** than Lock-Free **faster** than Wait-Free

* In the general case:

I'm focusing here

♦ Stronger properties are harder to maintain

Related Work: Lock-free, CAS-free, wait-free

* Single-Producer-Single-Consumer FIFO queues

- ♦ Lamport et al. 1983 Trans. PLS (Sequential consistency only - in memory)
- ♦ Higham and Kavalsh. 1997 ISPAN (PICI - TSO + proof - in memory)
- ♦ Giacomoni et al. 2008 PPOPP (TSO + cache slipping - in memory)
- ♦ BatchQueue & MCRingBuffer (TSO, double/multiple-buffering - in memory)

* Multiple-Producers-Multiple-Consumers FIFO queues

- ♦ Blocking 2-locks - Michael and Scott
- ♦ Nonblocking with CAS - list-based - Michael and Scott (PODC96)
 - Requires deferred reclamation/hazard pointers to avoid ABA problem
- ♦ Nonblocking with CAS - array-based - Tsigas and Zhang (PAA01)
- ♦ Nonblocking without CAS - in memory \Rightarrow Cannot be done
- ♦ Nonblocking without CAS - with mediator thread \Rightarrow FastFlow

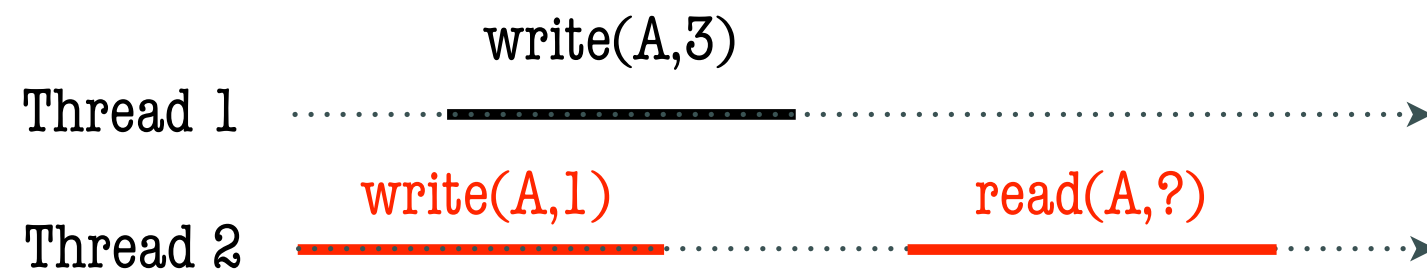
Recap: coherence and consistency



UNIVERSITÀ DEGLI STUDI
DI TORINO

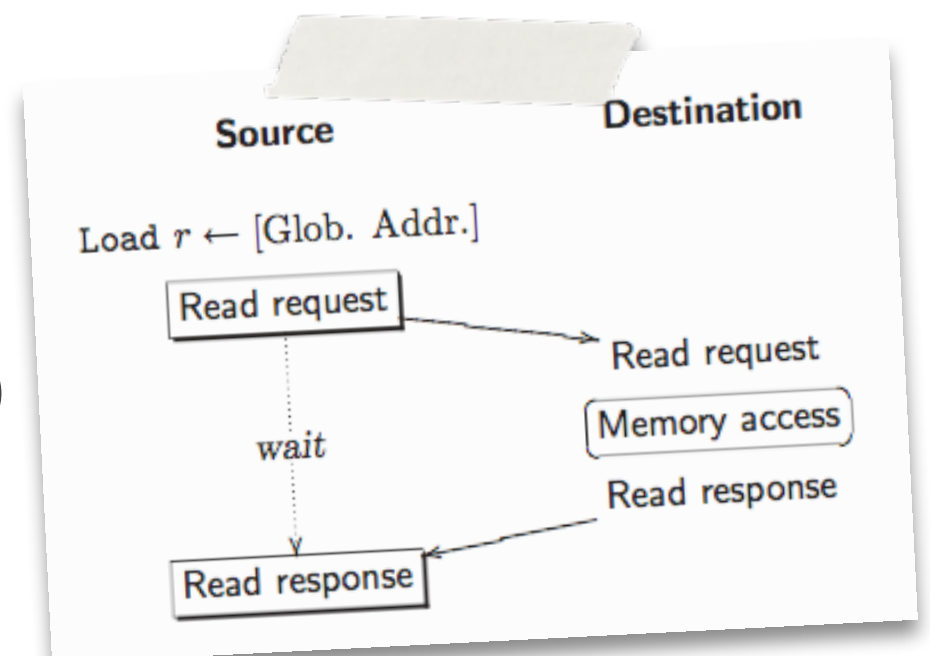
* Memory/Cache Coherence

- ◆ Deal with multiple replicas of the same location in different caches



* Memory Consistency

- ◆ Deal with the ordering in which writes and reads at different locations take effect in memory (issued by either the same or different processors/cores)



FastFlow SPSC queues

```
push_nonblocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}
```

```
pop_nonblocking(data) {  
  
    if (head == tail) {  
        return EWOULDBLOCK;  
    }  
    data = buffer[tail];  
    tail = NEXT(tail);  
    return 0;  
}
```

Lamport FIFO - 1983

```
push_nonblocking(data) {
    if (NULL != buffer[head]) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}
```

For any model weaker than TS0

```
pop_nonblocking(data) {
    data = buffer[tail];
    if (NULL == data) {
        return EWOULDBLOCK;
    }
    buffer[tail] = NULL;
    tail = NEXT(tail);
    return 0;
}
```

FastFlow FIFO

derived from PIC1 (Higham and Kavalsh, ISPAN 1997)
& FastForward (Giacomoni et al, PPOPP 2008)

FastFlow SPSC queues

```
push_nonblocking(data) {  
    if (NEXT(head) == tail) {  
        return EWOULDBLOCK;  
    }  
    buffer[head] = data;  
    head = NEXT(head);  
    return 0;  
}  
  
pop_nonblocking(data) {  
    if (head == tail) {  
        return EWOULDBLOCK;  
    }  
    data = buffer[tail];  
    tail = NEXT(tail);  
    return 0;  
}
```

head and tail are
mutually invalidated by
producer and consumer
| cache miss every push
and pop (at least)

Lamport FIFO - 1983

```
push_nonblocking(data) {  
    if (NULL != buffer[head]) {  
        return EWOULDBLOCK;  
    }  
    buffer[head] = data; (WMB)  
    head = NEXT(head);  
    return 0;  
}  
  
pop_nonblocking(data)  
data = buffer[tail];  
if (NULL == data)  
    return EWOULDBLOCK;  
buffer[tail] = NULL;  
tail = NEXT(tail);  
return 0;  
}
```

producer read/write head
consumer read/write tail
no misses

excluding "true" deps

extended domain
on void *

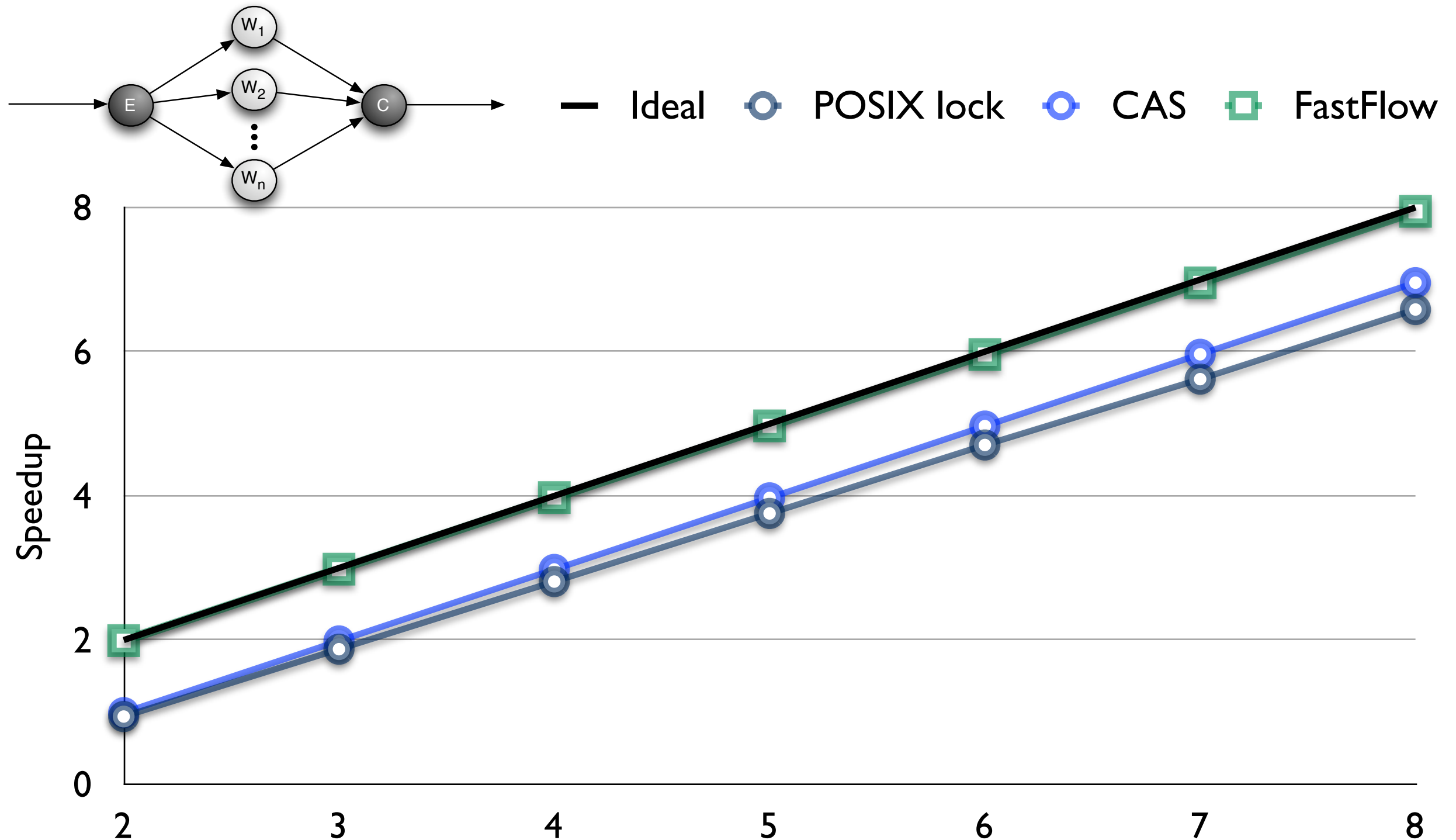
FastFlow FIFO

derived from PICI (Higham and Kavalsh, ISPAN 1997)
& FastForward (Giacomoni et al, PPOPP 2008)

Lock vs CAS vs SPSC FastFlow (50 μ S)



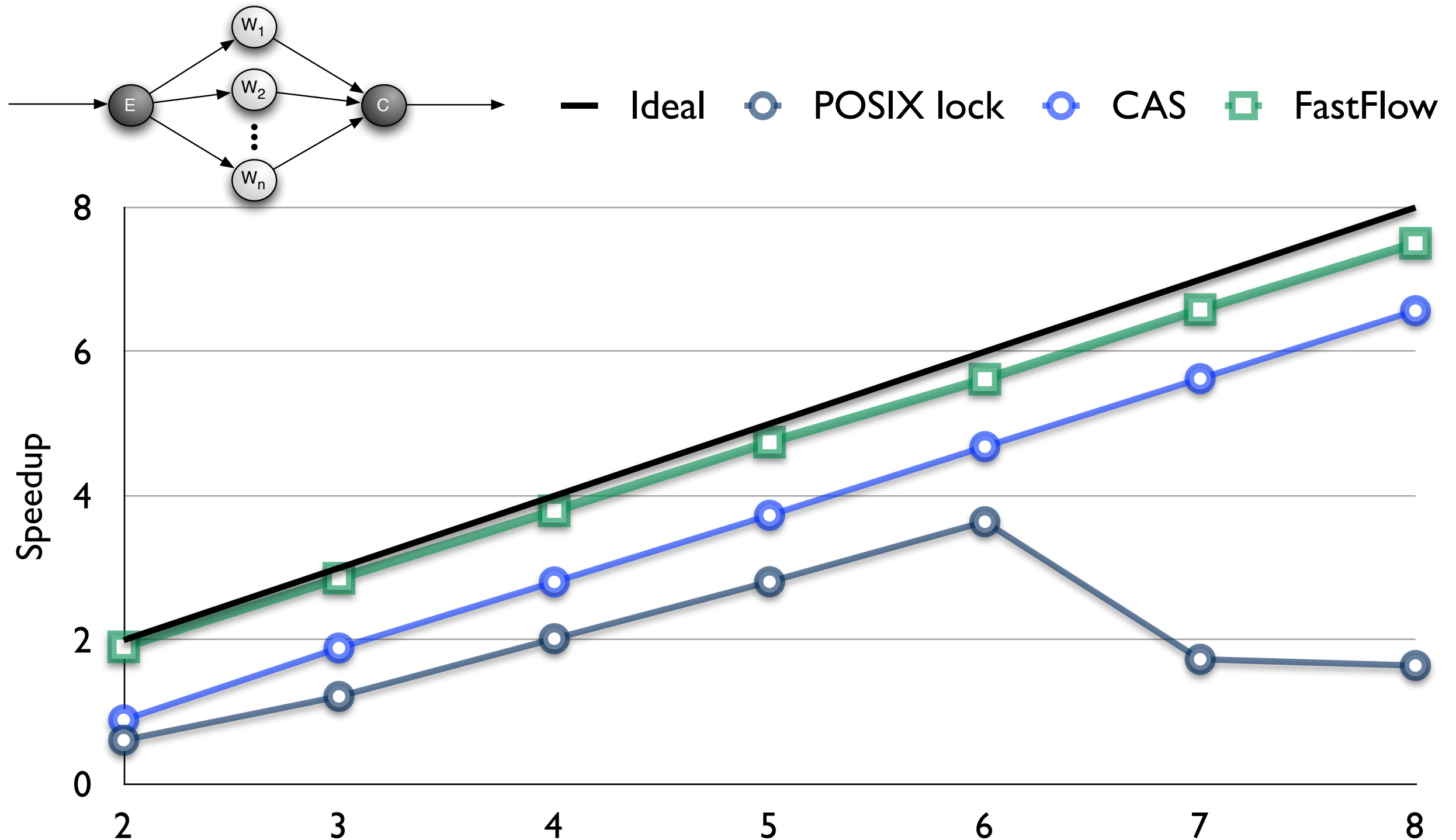
UNIVERSITÀ DEGLI STUDI
DI TORINO



Lock vs CAS vs SPSC FastFlow (5 μ S)



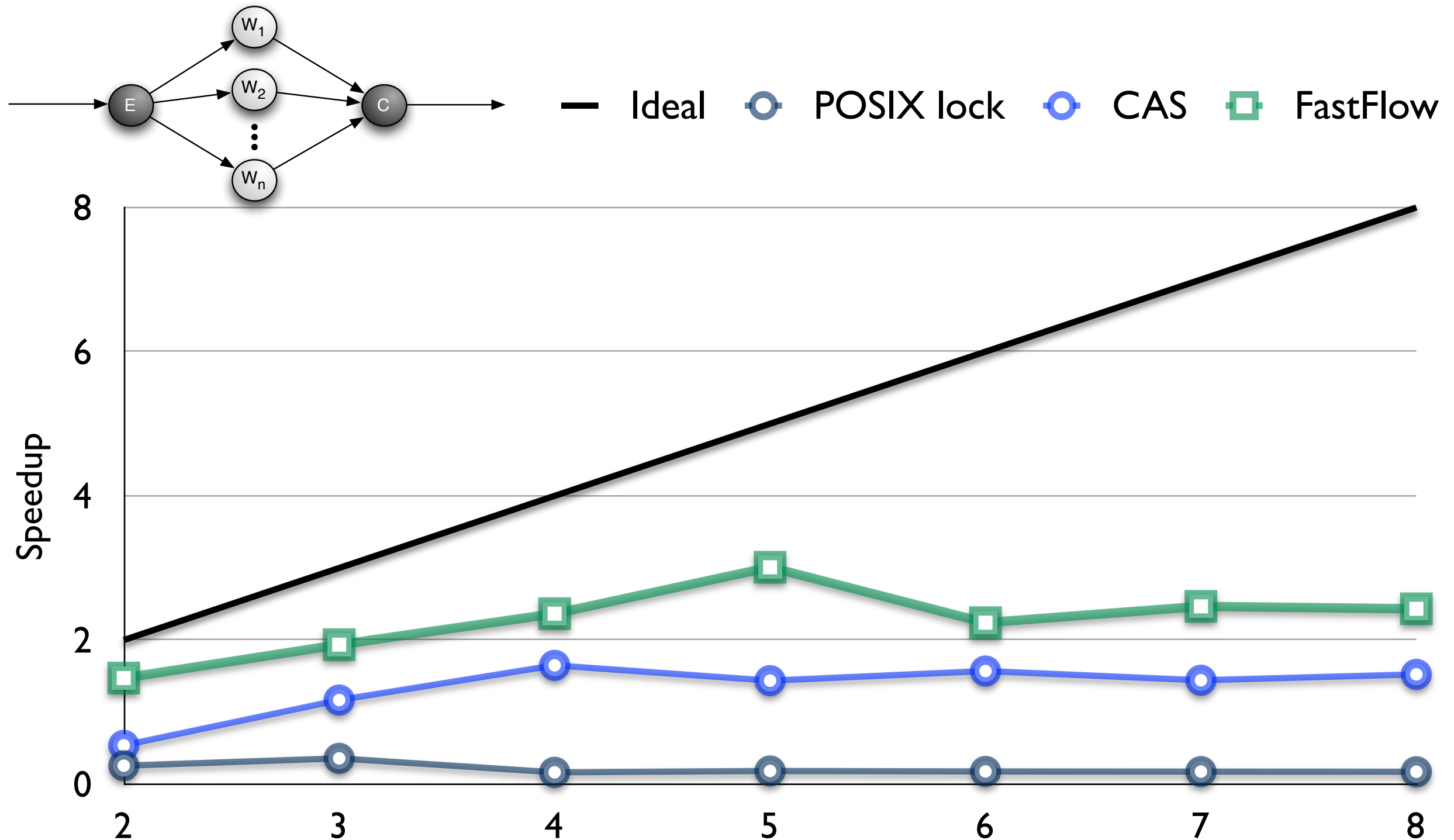
UNIVERSITÀ DEGLI STUDI
DI TORINO



Lock vs CAS vs SPSC FastFlow (0.5 μ S)



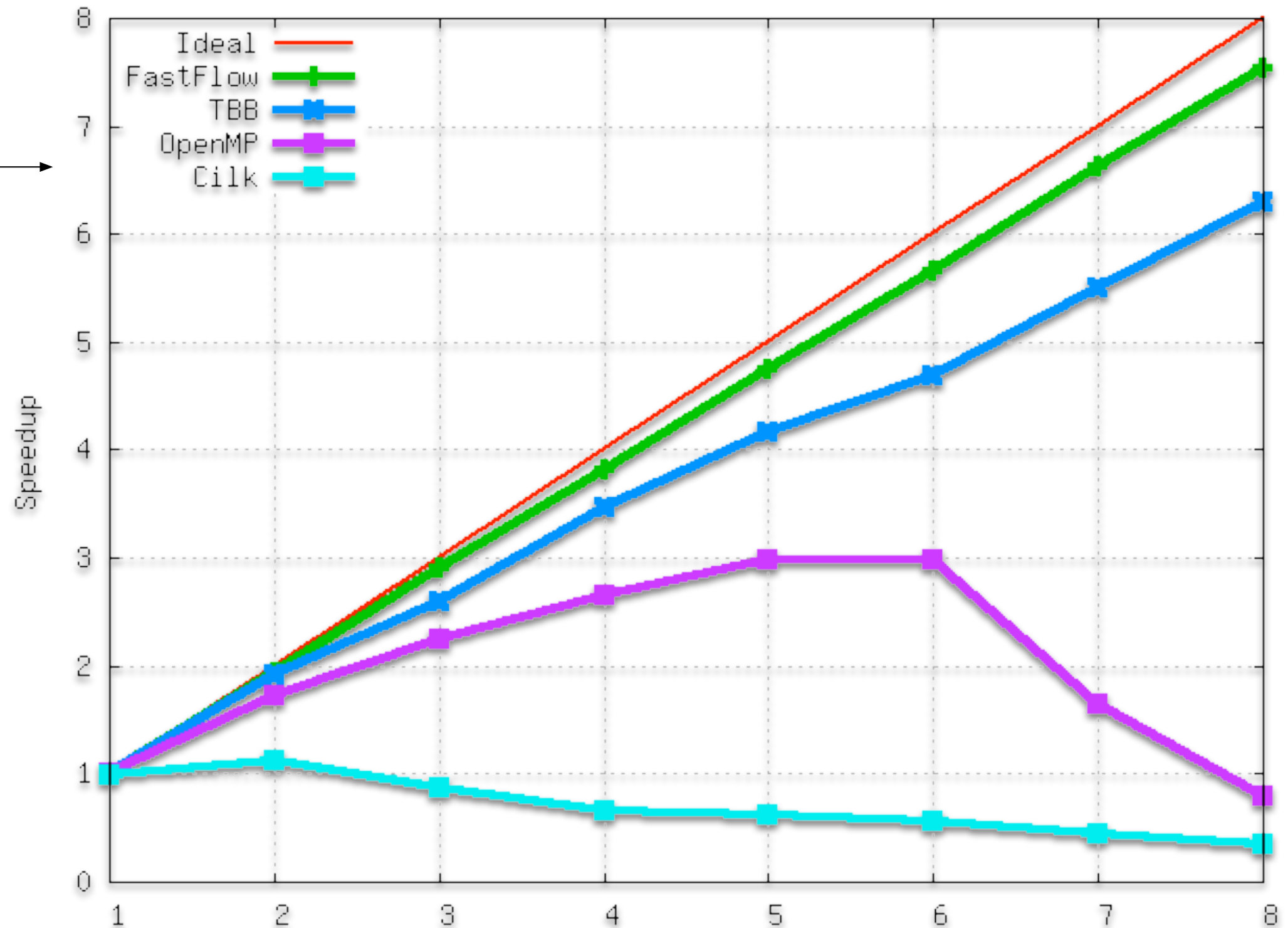
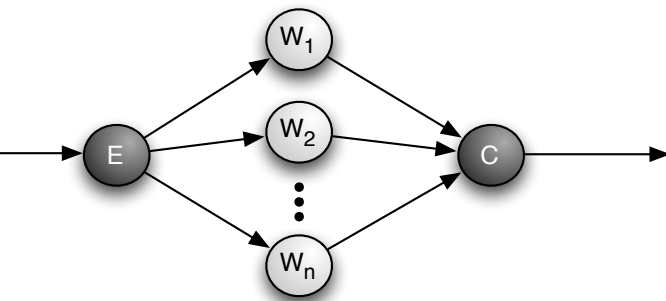
UNIVERSITÀ DEGLI STUDI
DI TORINO



Medium grain (5 μ S workload)



UNIVERSITÀ DEGLI STUDI
DI TORINO



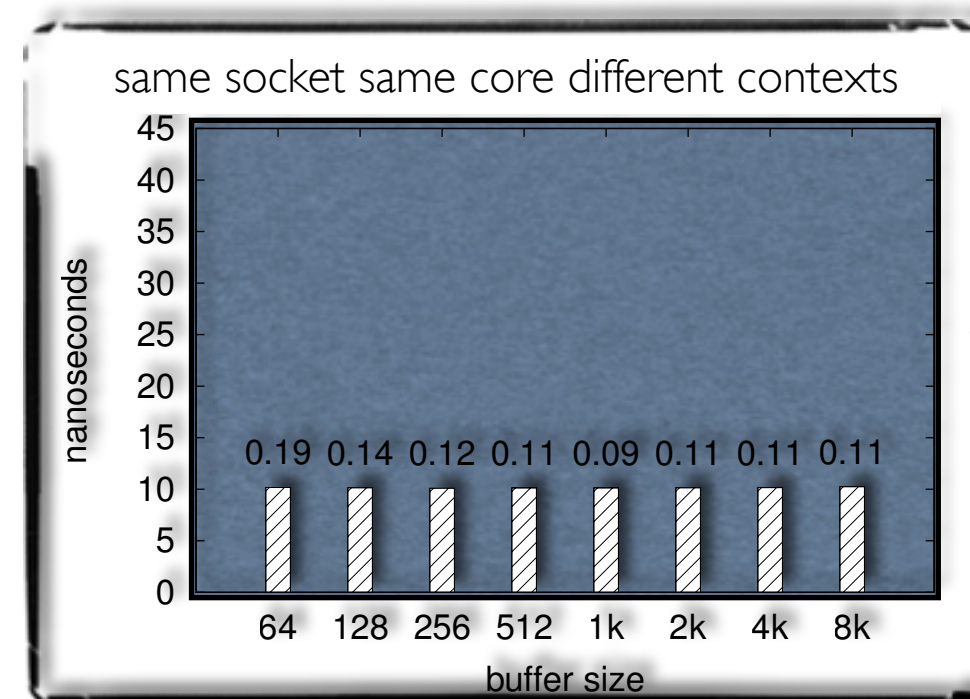
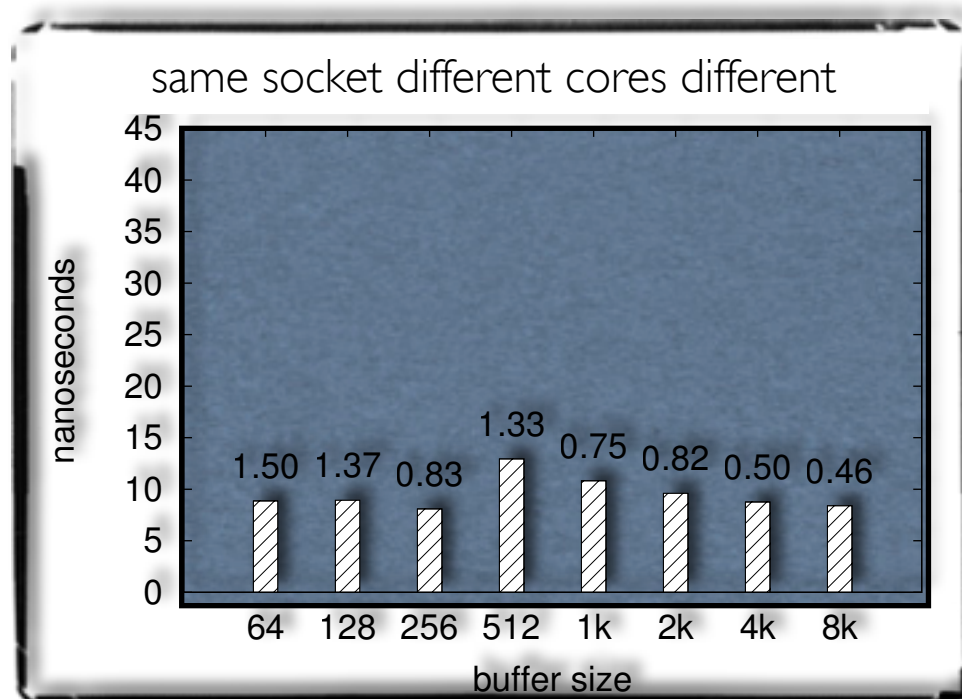
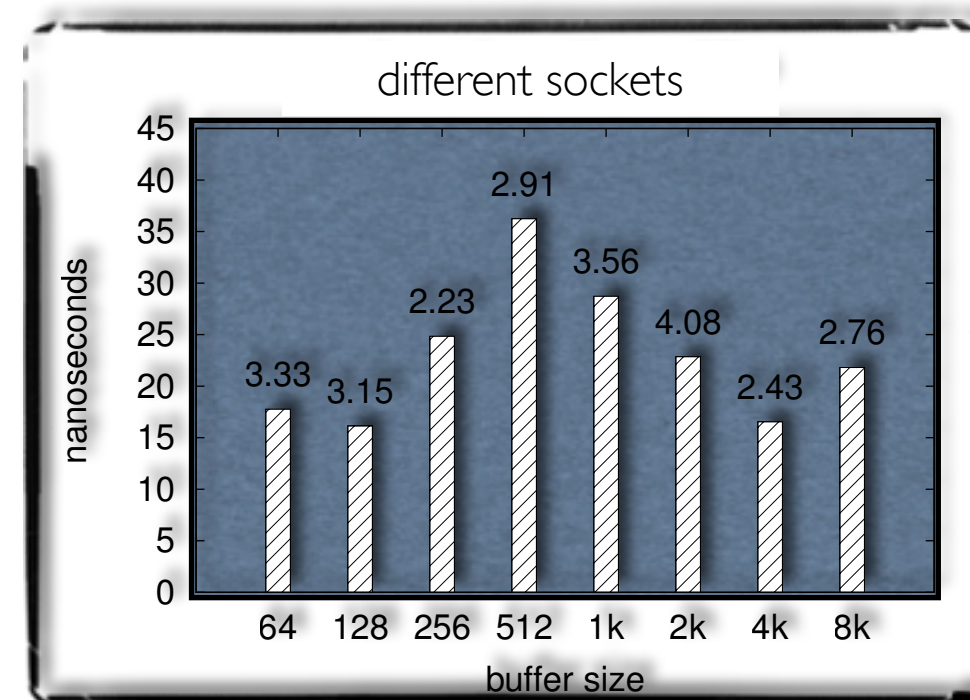
Layer I: Simple streaming networks



UNIVERSITÀ DEGLI STUDI
DI TORINO

4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2



Layer 1: Simple streaming networks

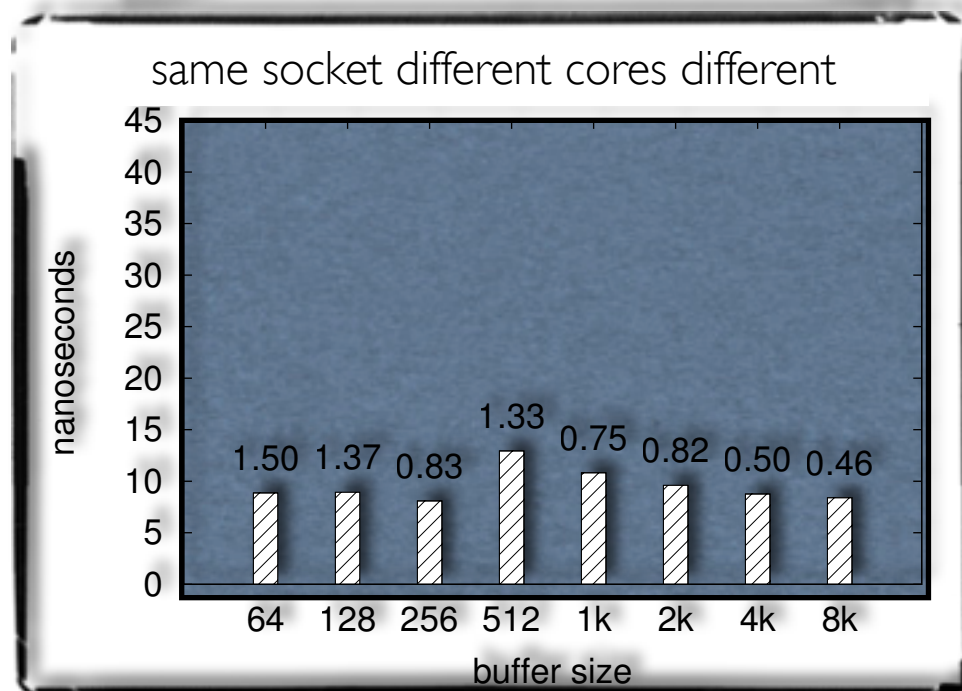
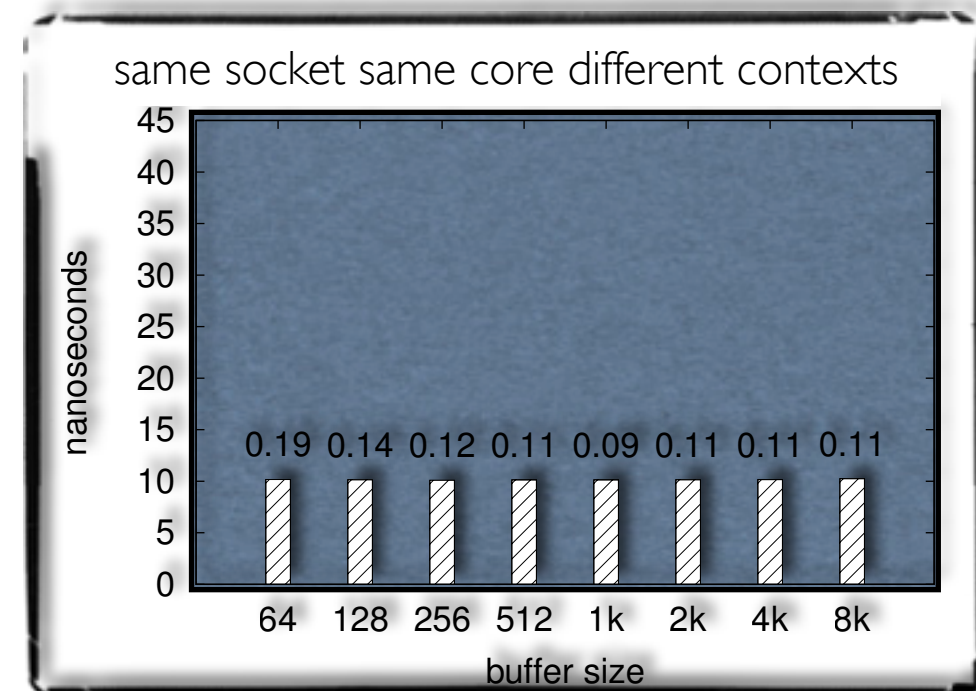
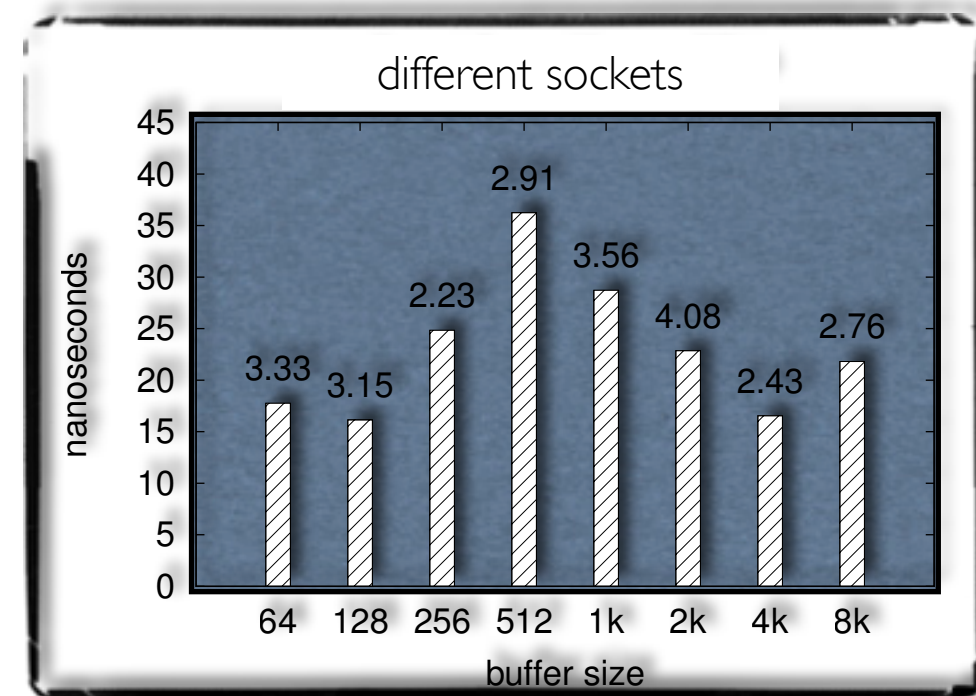


UNIVERSITÀ DEGLI STUDI
DI TORINO

4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

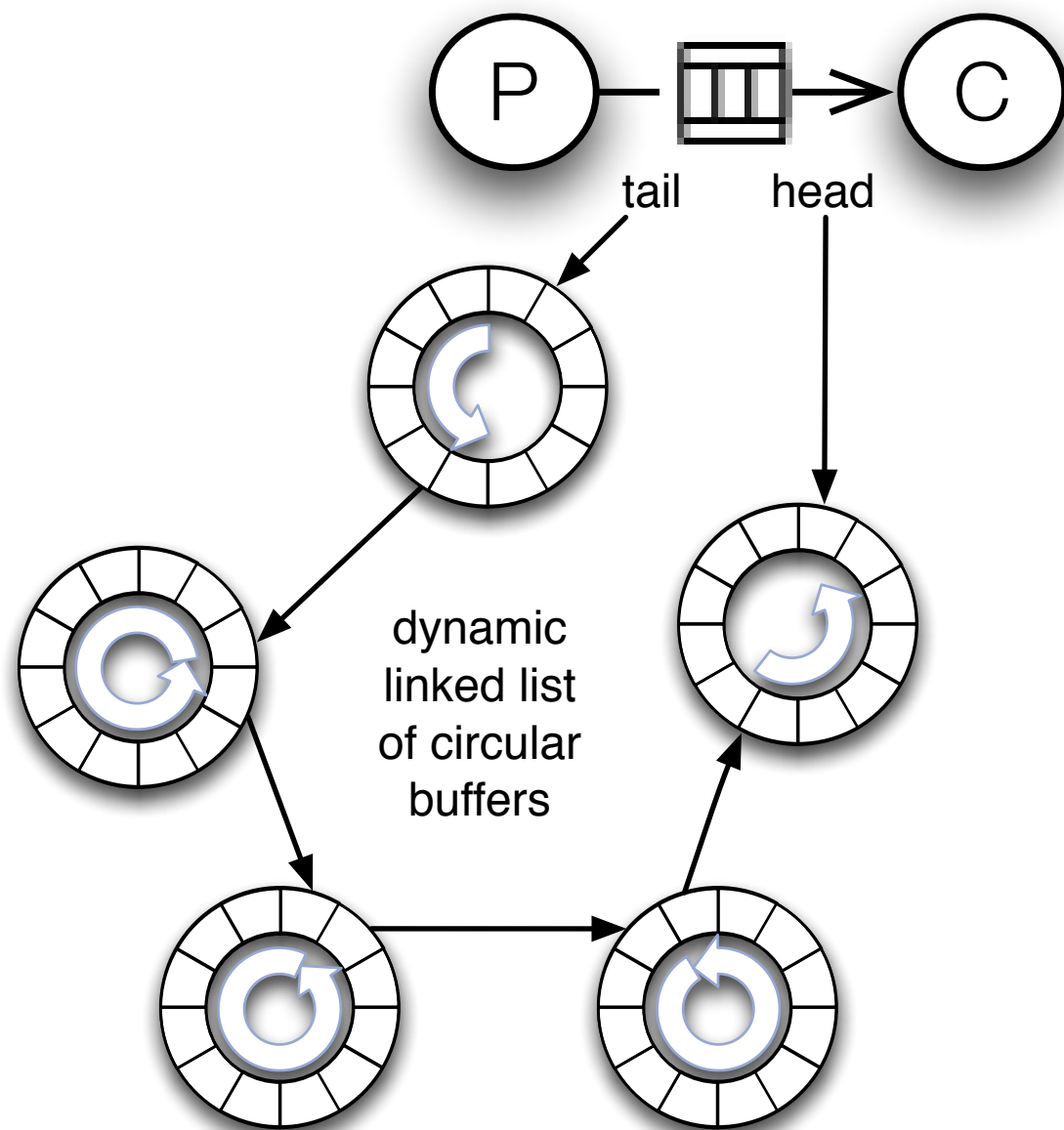
MPI shmem impl
is ~190 ns at best
(D.K. Panda)



Layer I: Simple streaming networks



UNIVERSITÀ DEGLI STUDI
DI TORINO



```

1 int size = N; //SPSC size
2 bool push(void* data) {
3     if (buf_w->full()) {
4         SPSC* t = pool.next_w();
5         if (!t) return false;
6         buf_w = t;
7     }
8     buf_w->push(data);
9     return true;
10 }
11 bool pop(void** data) {
12     if (buf_r->empty()) {
13         if (buf_r == buf_w) return false;
14         if (buf_r->empty()) {
15             SPSC* tmp = pool.next_r();
16             if (tmp) {
17                 pool.release(buf_r);
18                 buf_r = tmp;
19             }
20         }
21     }
22     return buf_r->pop(data);
23 }

```

```

25 struct Pool {
26     dSPSC inuse;
27     SPSC cache;
28
29     SPSC* next_w() {
30         SPSC* buf;
31         if (!cache.pop(&buf))
32             buf = allocateSPSC(size);
33         inuse.push(buf);
34         return buf;
35     }
36     SPSC* next_r() {
37         SPSC* buf;
38         return (inuse.pop(&buf)? buf : NULL);
39     }
40     void release(SPSC* buf) {
41         buf->reset(); // reset pread and pwrite
42         if (!cache.push(buf))
43             deallocateSPSC(buf);
44     }
45 }

```

Fig. 3: Unbounded wait-free uSPSC queue implementation.

M. Aldinucci, S. Campa, M. Danelutto, M. Torquati. An Efficient Synchronisation Mechanism for Multi-Core Systems.

EuroPar 2012.

Wed 29 Aug - B3 multicore 14.30-16.00

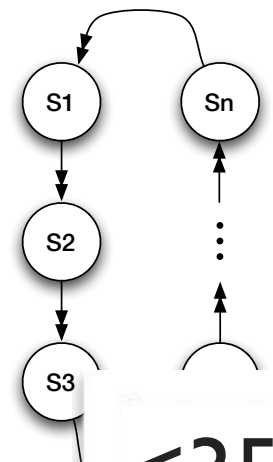
PARAPHRASE

Layer I: Simple streaming networks

<http://www.1024cores.net/home/technologies/fastflow>



UNIVERSITÀ DEGLI STUDI
DI TORINO



4 sockets x 8 core x 2 contexts

Xeon E7-4820 @2.0GHz Sandy Bridge
18MB L3 shared cache, 256K L2

**<35 ns irrespectively
of the mapping**

Speedup

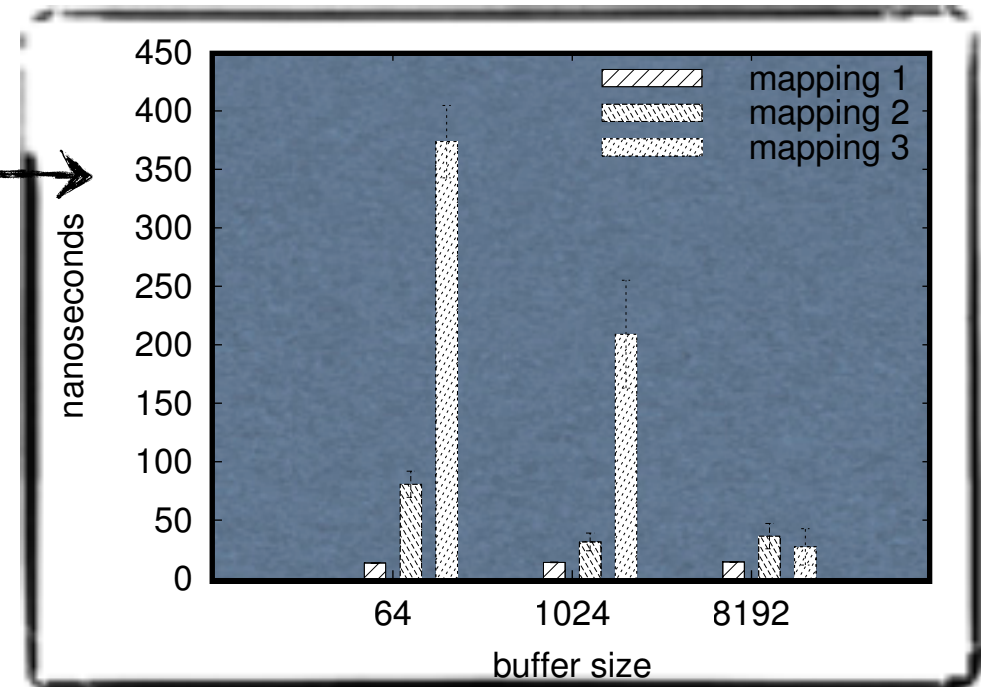
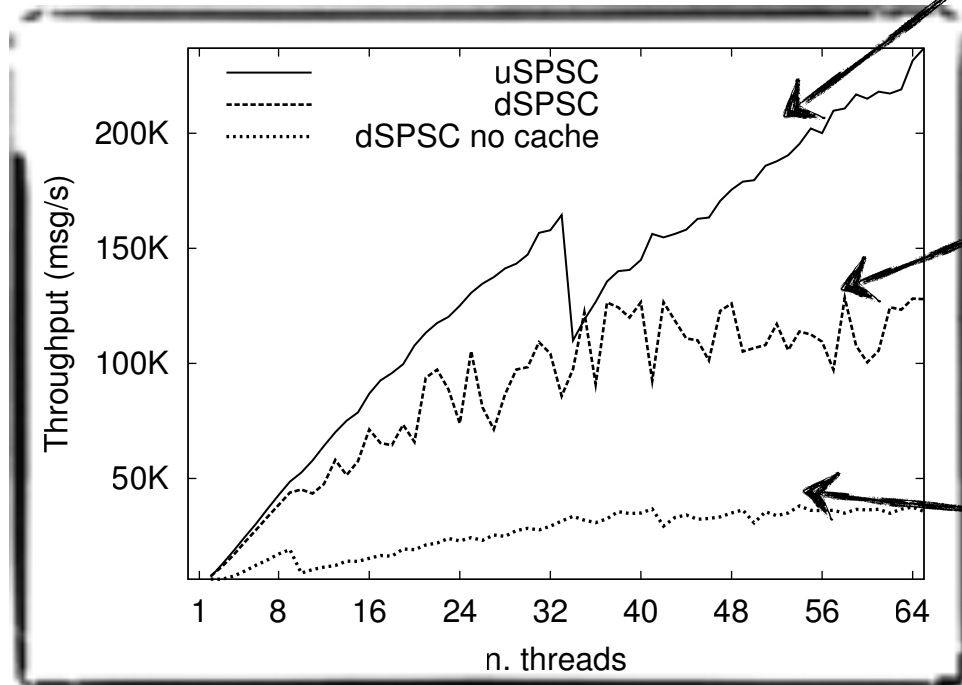
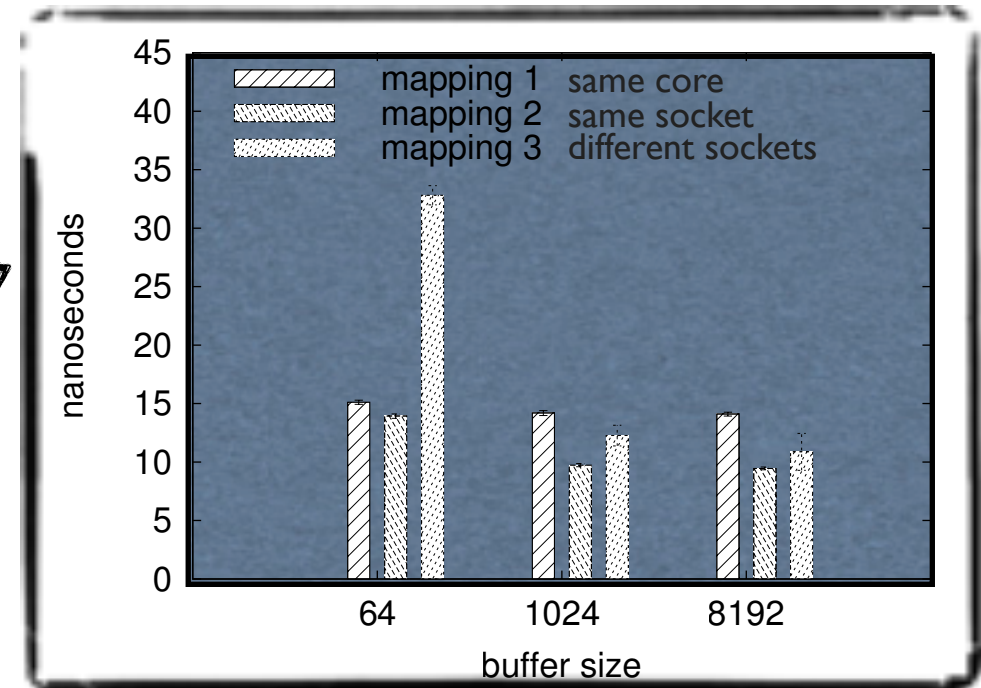
Linked list + circular buffer
FastFlow queue
(our result)

12x faster

Linked list with pooling
Opt Michael-Scott queue
(our result)

20x faster

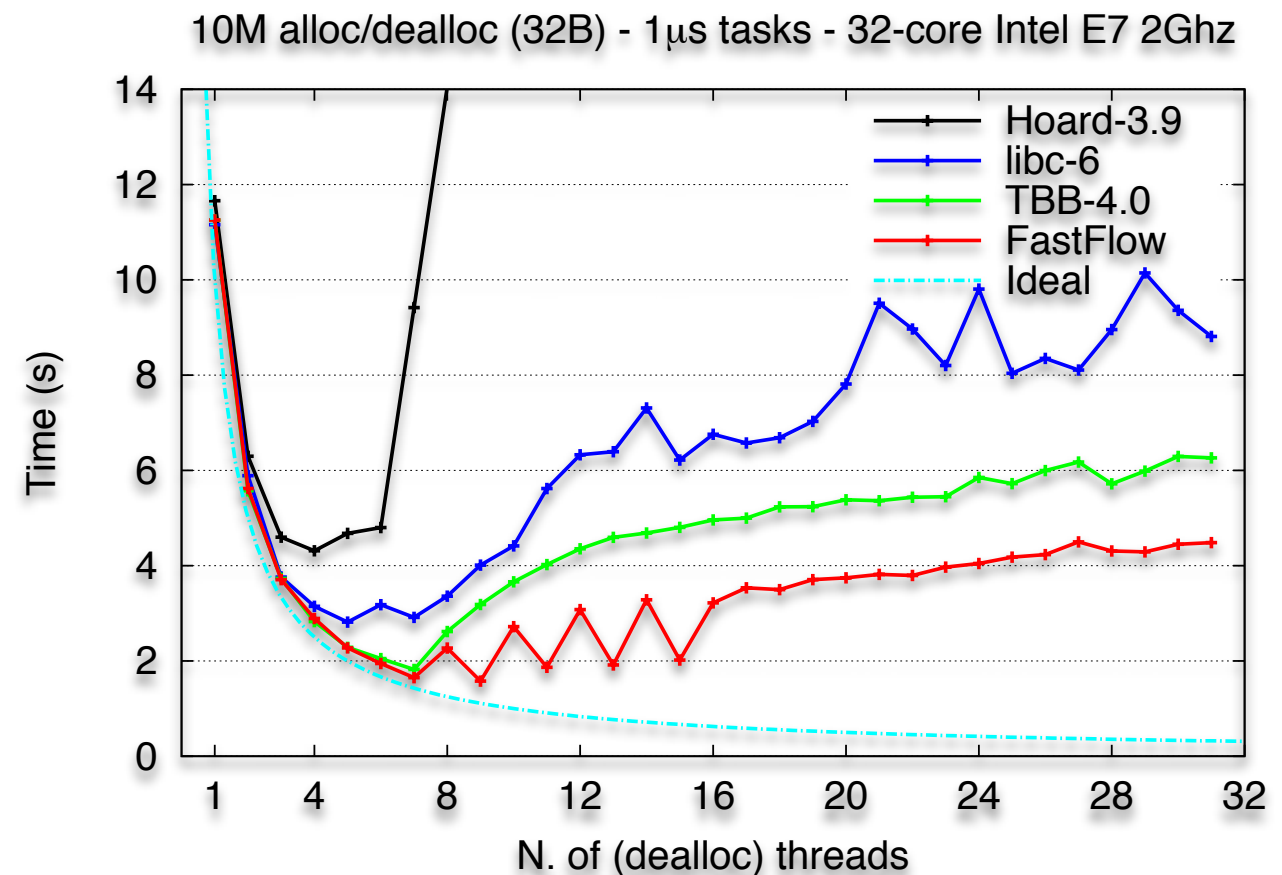
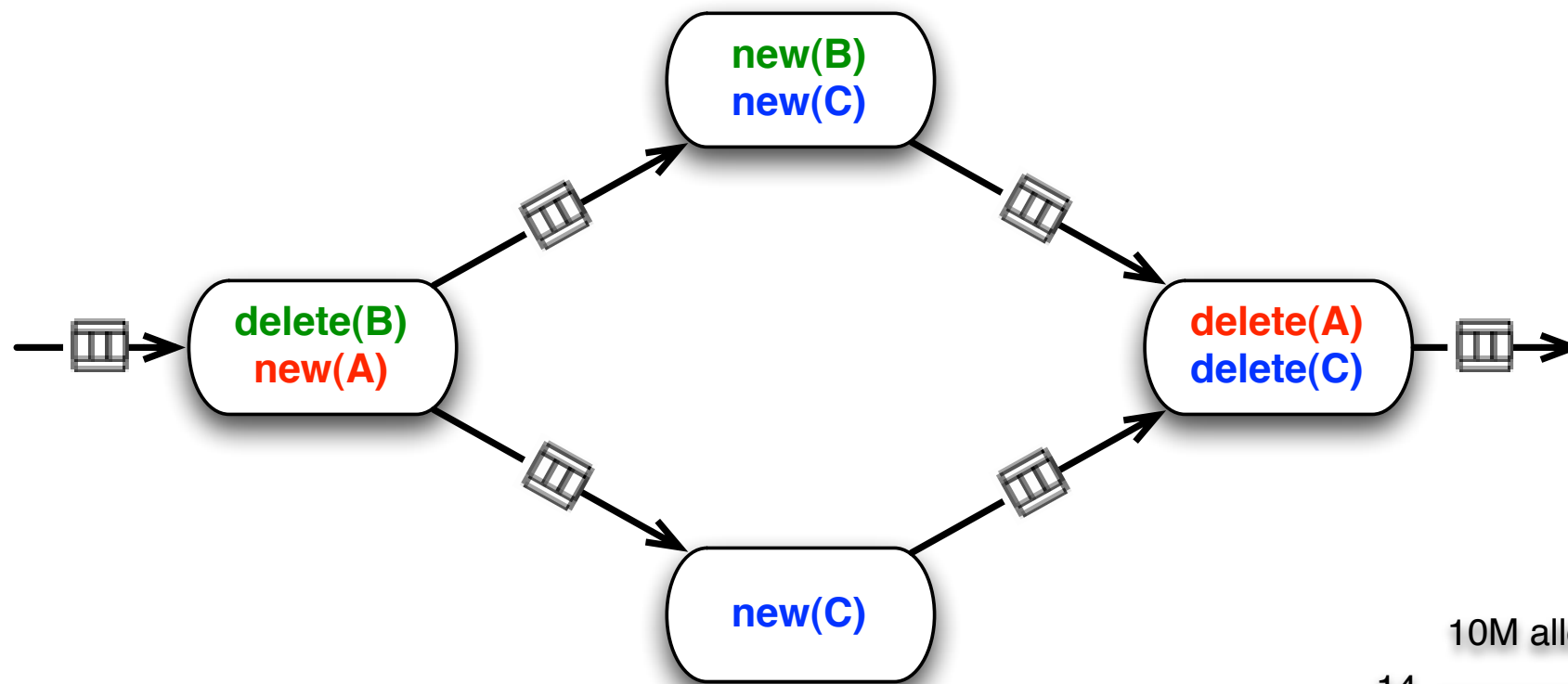
Linked list w/ dyn alloc
Michael-Scott queue
well-known ~ 400 citations



Unbound queues are useful



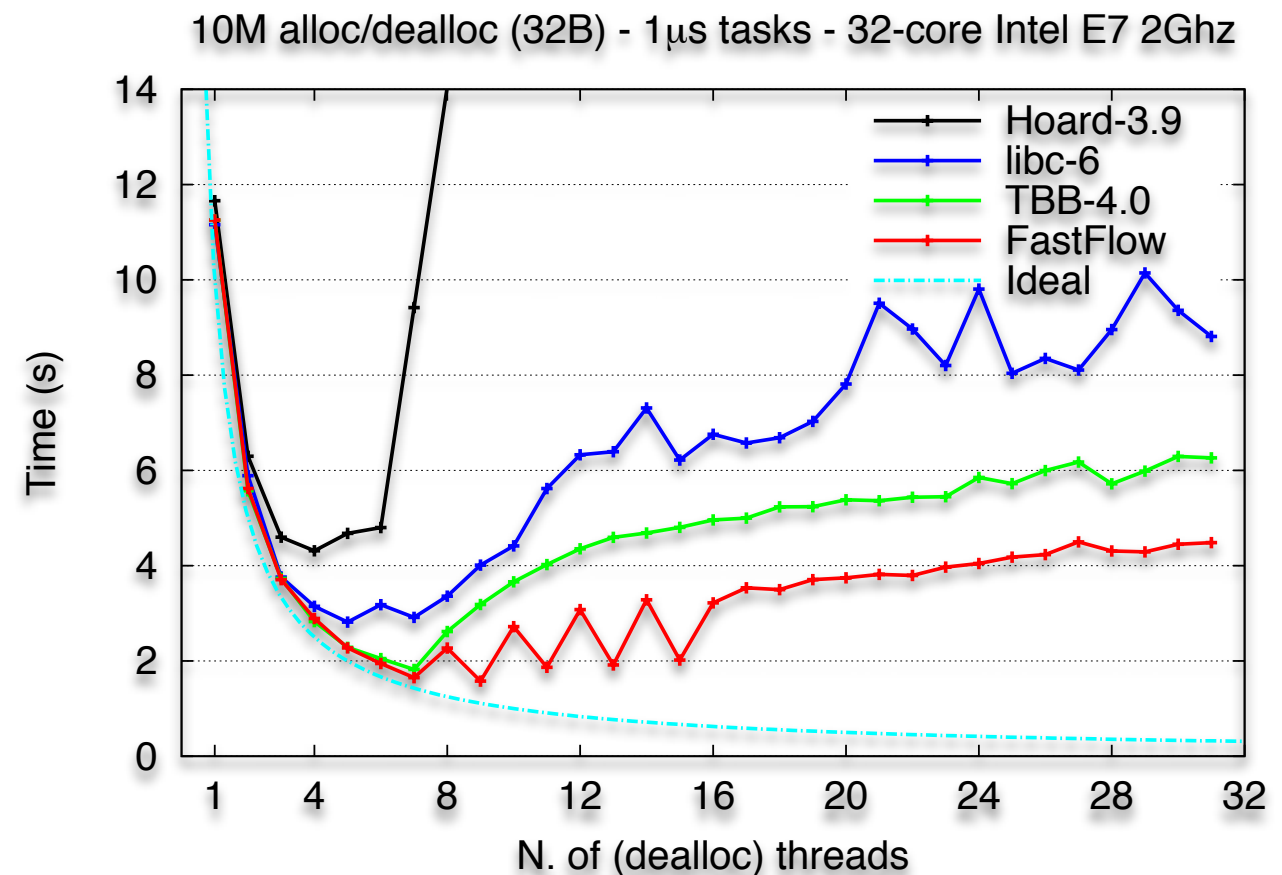
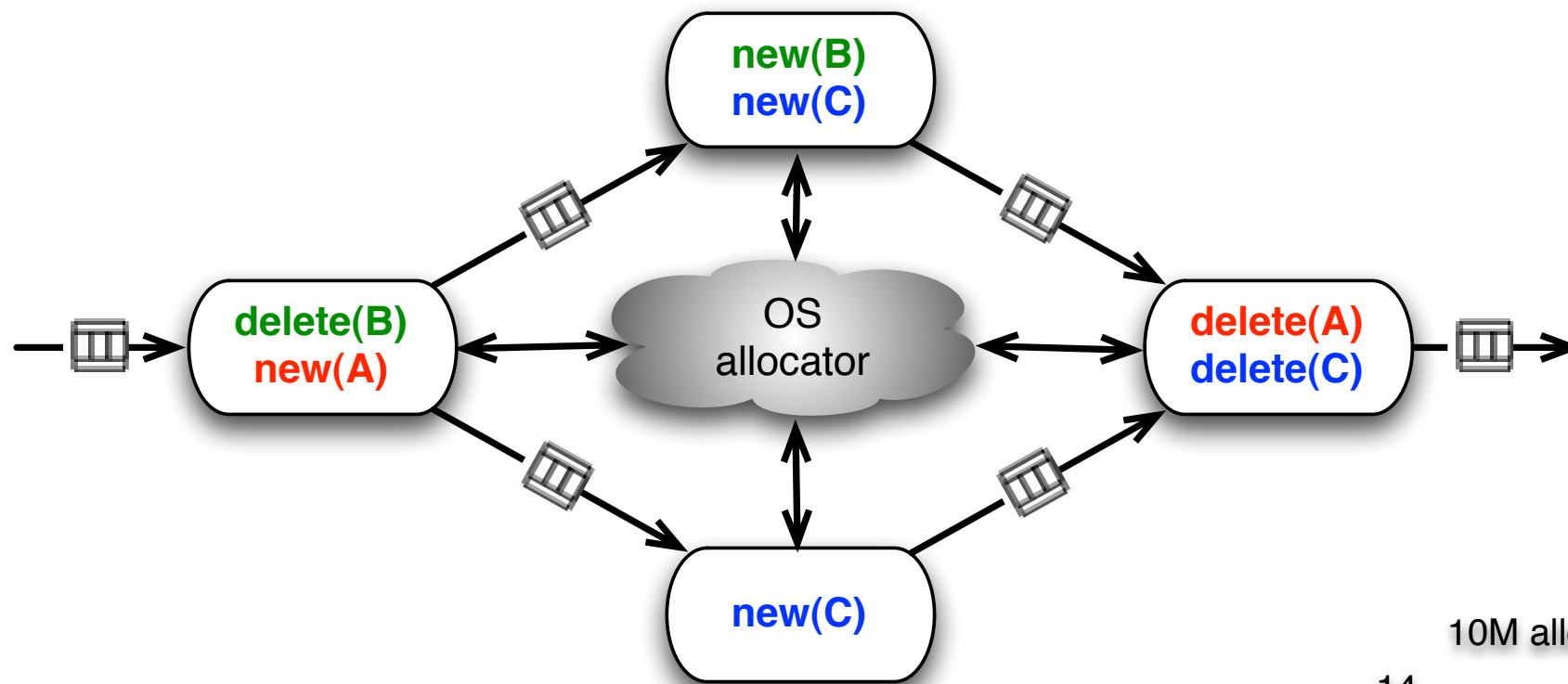
UNIVERSITÀ DEGLI STUDI
DI TORINO



Unbound queues are useful



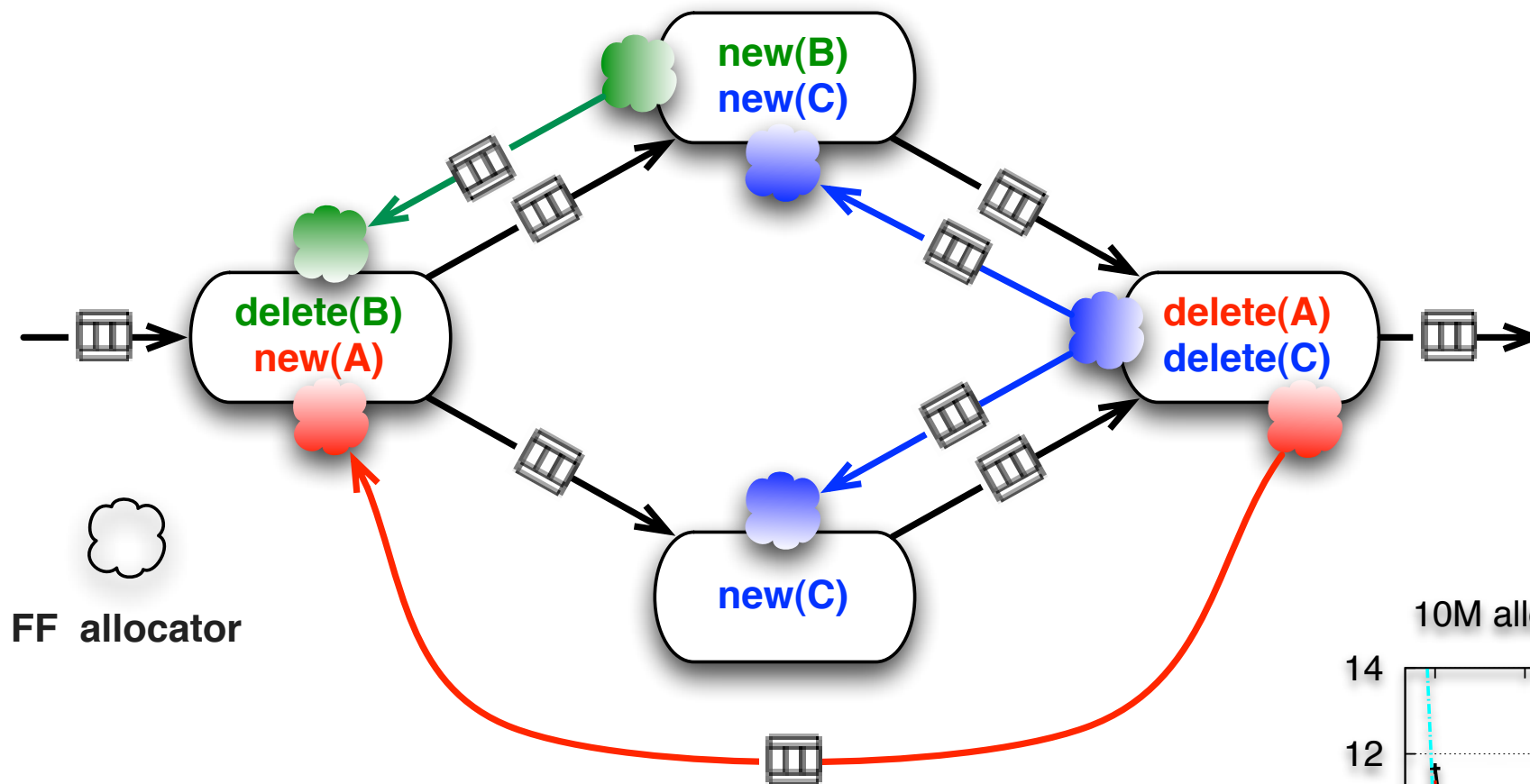
UNIVERSITÀ DEGLI STUDI
DI TORINO



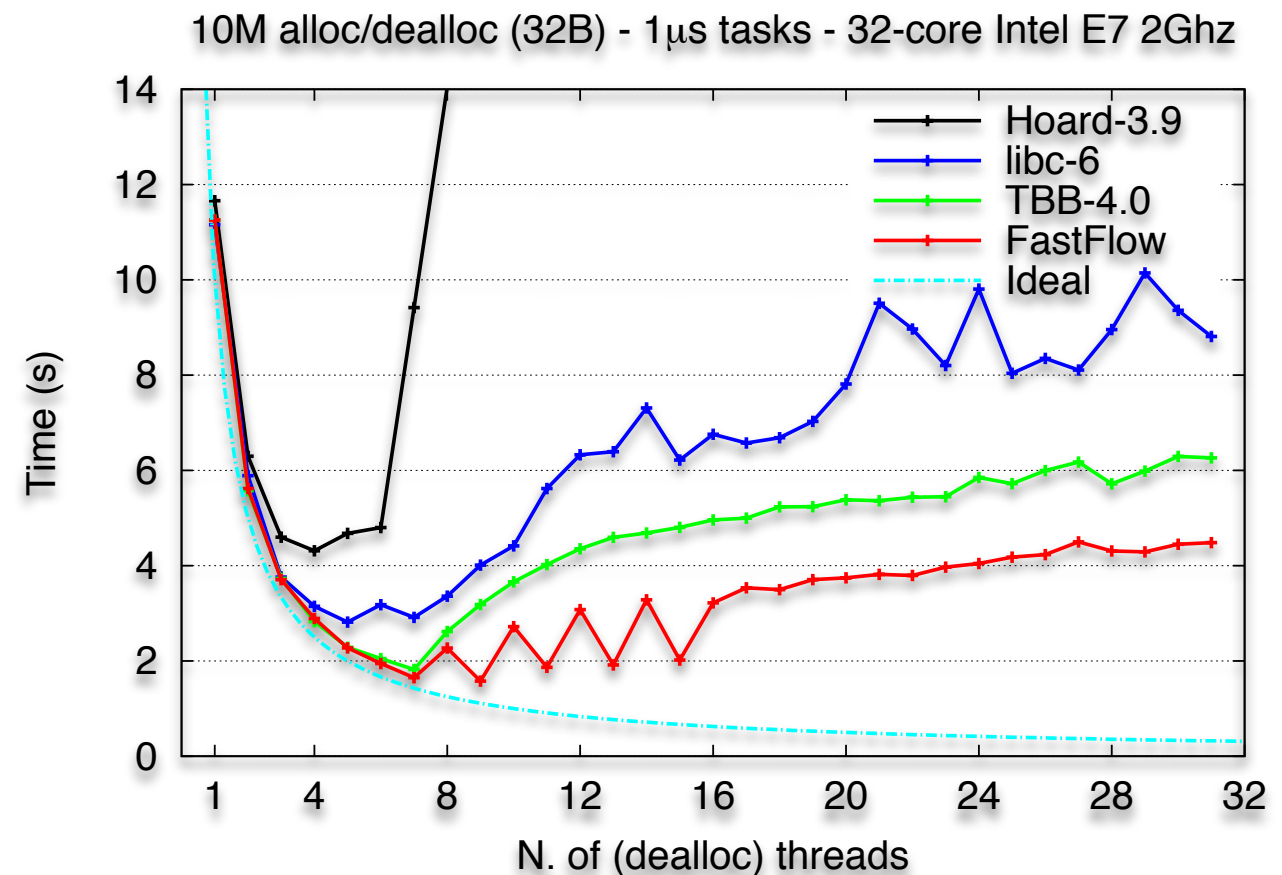
Unbound queues are useful



UNIVERSITÀ DEGLI STUDI
DI TORINO



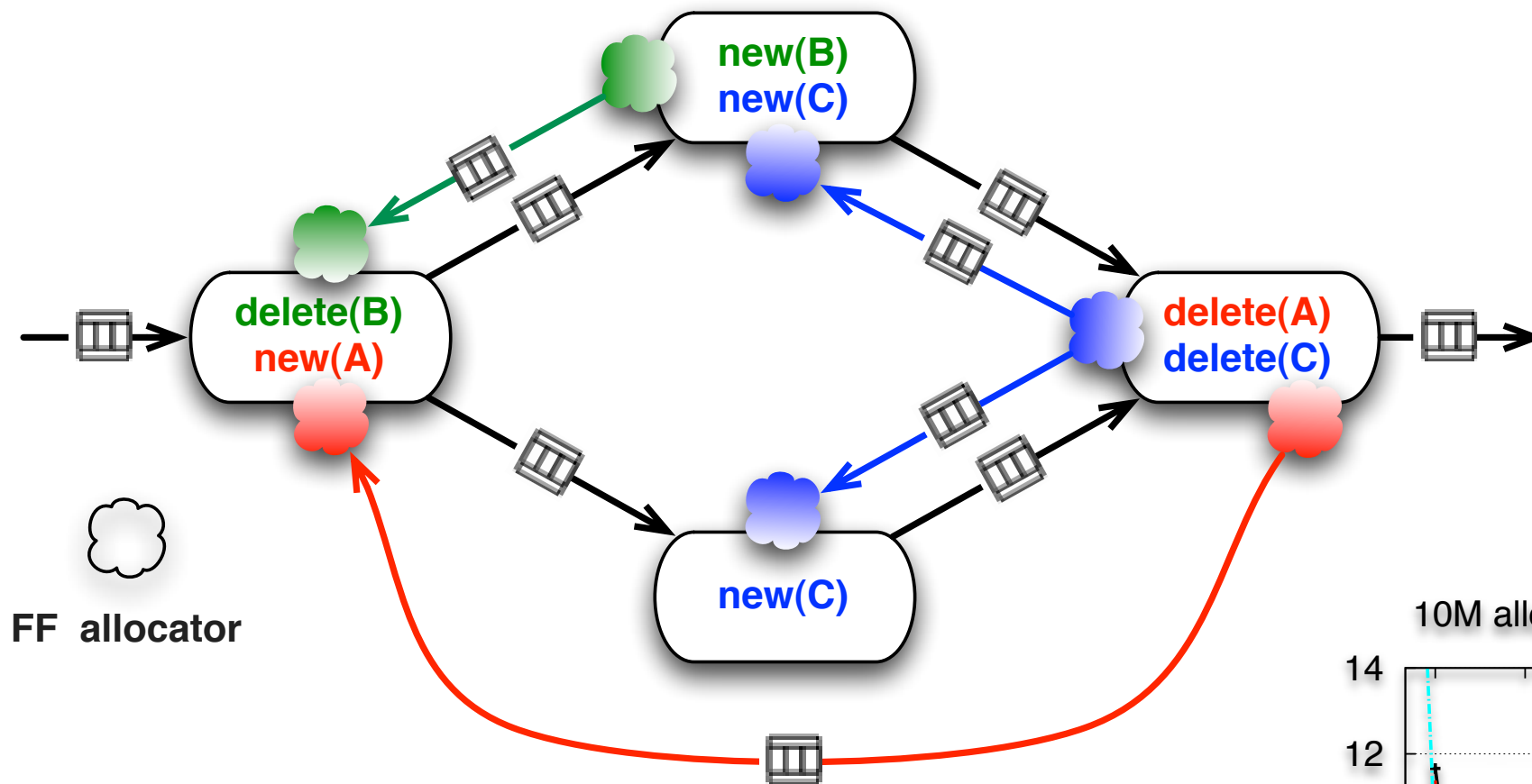
- * Faster than posix, often faster than hoard and TBB
 - ◆ unpublished, but available on sourceforge
 - ◆ needs lot of comparative testing to be published
- * Implements deferred deallocation to avoid ABA problem



Unbound queues are useful

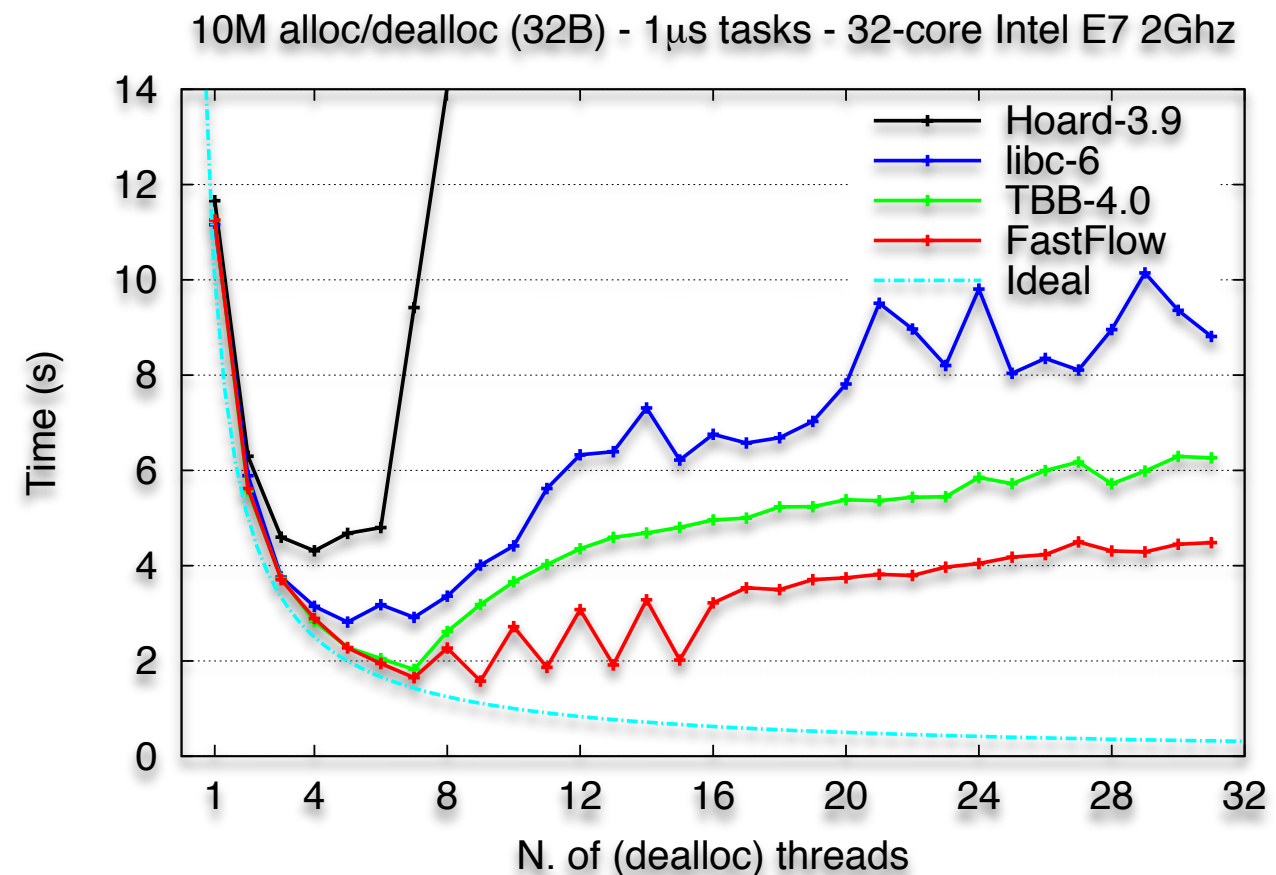


UNIVERSITÀ DEGLI STUDI
DI TORINO



The graph is now cyclic
and needs unbound
queues to avoid
deadlocks

- * Faster than posix, often faster than hoard and TBB
 - ◆ unpublished, but available on sourceforge
 - ◆ needs lot of comparative testing to be published
- * Implements deferred deallocation to avoid ABA problem





FastFlow

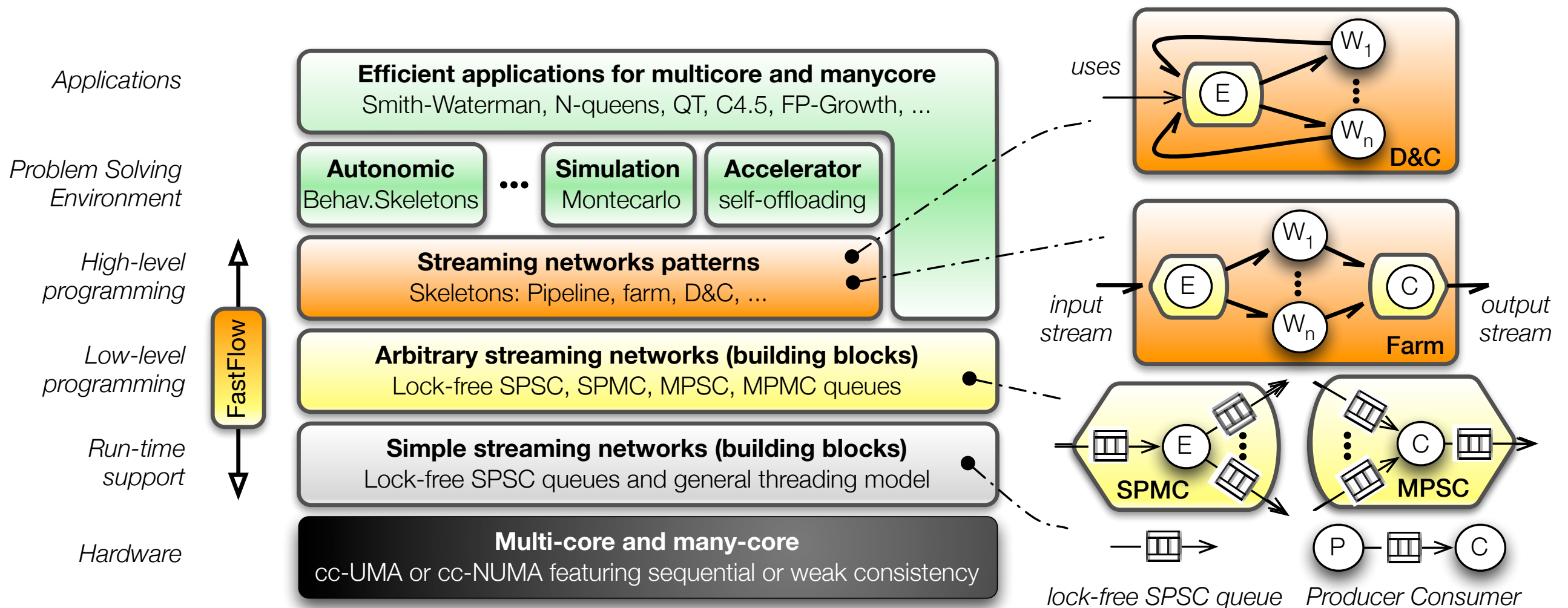
Lock-free and CAS-free?

- * Mutex cannot be done
Single-Producer-Single-Consumer (SPSC) can be done
 - ◆ Producer-Consumer is inherently weaker with respect to Mutex
 - ◆ It does require the cooperation of partners whereas Mutex does not
- * Expressive enough to build a streaming (or dataflow) programming framework
 - ◆ MPMC = SPSC + mediator threads
- * But what about productivity at large scale?
 - ◆ Write a program is defining a graph encoding true dependencies ... not really easy

FastFlow is based on producer-consumer



UNIVERSITÀ DEGLI STUDI
DI TORINO



- ◆ Lock-free/fence-free non-blocking synchronisations
- ◆ C++ STL-like implementation
- ◆ thread-model agnostic (pthreads, QT, windows threads, ...)
- ◆ compliant with other synchronisation mechanisms in the business code (e.g. locks and semaphores)

Pattern-based approach: rationale



UNIVERSITÀ DEGLI STUDI
DI TORINO

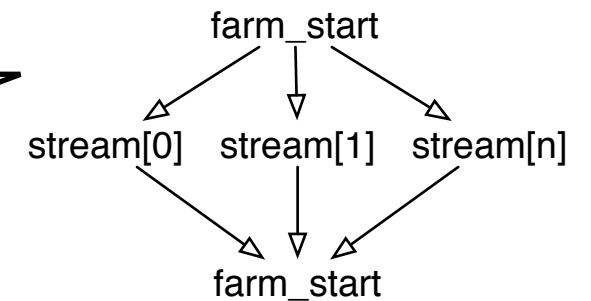
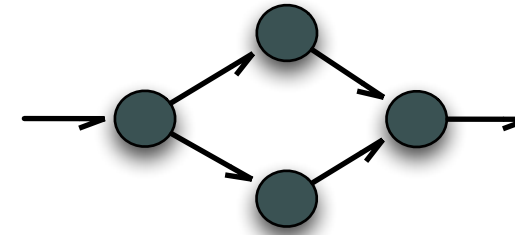
- * Abstract parallelism exploitation pattern by parametric code
 - ◆ E.g. higher order function, code factories, C++ templates, ...
 - ◆ Can be composed and nested as programming language constructs + offloading
 - ◆ Stream and Data Parallel
- * Platform independent
 - ◆ Implementations on different multi/many-cores
 - ◆ Support for hybrid architectures thanks to pattern compositionality
- * Provide state-of-the-art, parametric implementation of each parallelism exploitation pattern
 - ◆ With natural way of extending patterns, i.e. OO
 - ◆ Functional (seq code) and tunable extra-functional (QoS) parameters



Patterns, their implementation, and their purpose

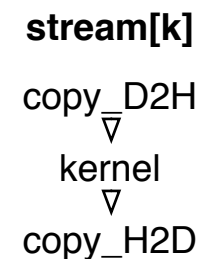
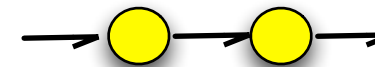
* farm

- ♦ on CPU - master-worker - parallelism exploitation
- ♦ on GPU - CUDA streams - automatic exploitation of asynch comm



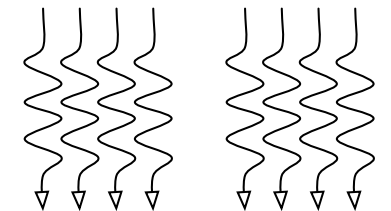
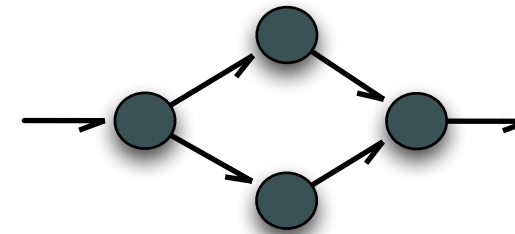
* pipeline

- ♦ on CPU - pipeline
- ♦ on GPU - sequence of kernel calls or global mem synch



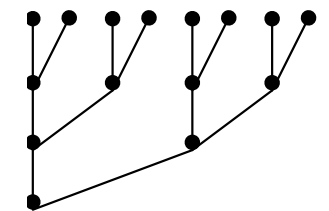
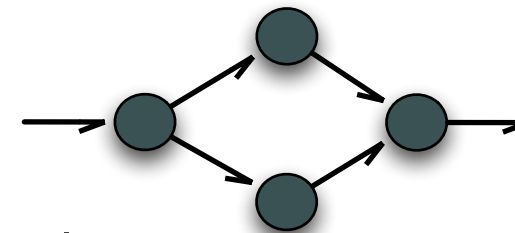
* map

- ♦ on CPU - master-worker - parallelism exploitation
- ♦ on GPU - CUDA SIMT - parallelism exploitation



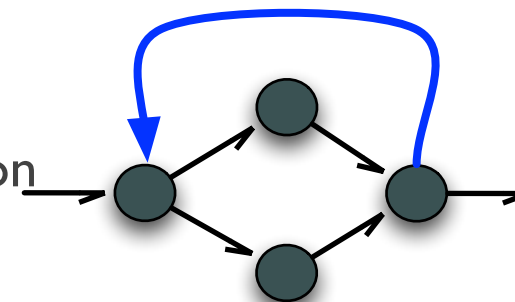
* reduce

- ♦ on CPU - master-worker - parallelism exploitation
- ♦ on GPU - CUDA SIMT (reduction tree) - parallelism exploitation



* D&C

- ♦ on CPU - master-worker with feedback - // exploitation
- ♦ on GPU - working on it, maybe loop+farm



Composition

* Composition via C++ template meta-programming

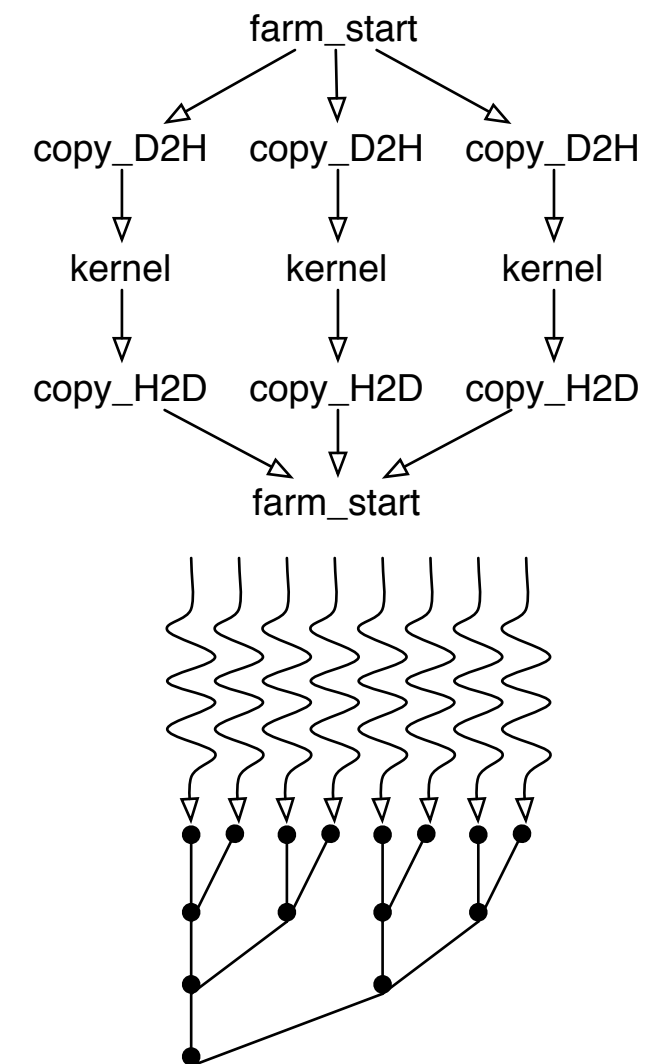
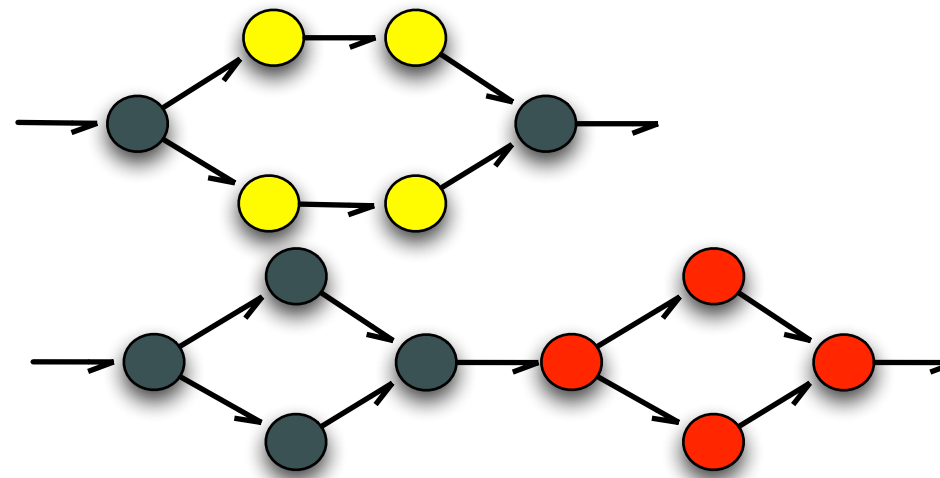
- ◆ CPU: Graph composition
- ◆ GPU: CUDA streams
- ◆ CPU+GPU: offloading

* `farm{ pipe }`

* `pipe(farm, farm)`

* `pipe(map, reduce)`

*



```

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

using namespace ff;

// generic worker
class Worker: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        std::cout << "Worker " << ff_node::get_my_id()
                  << " received task " << *t << "\n";
        return task;
    }
    // I don't need the following functions for this test
    //int  svc_init() { return 0; }
    //void svc_end() {}
};

// the gatherer filter
class Collector: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        if (*t == -1) return NULL;
        return task;
    }
};

// the load-balancer filter
class Emitter: public ff_node {
public:
    Emitter(int max_task):ntask(max_task) {};

    void * svc(void *) {
        int * task = new int(ntask);
        --ntask;
        if (ntask<0) return NULL;
        return task;
    }
private:
    int ntask;
};

```

```

int main(int argc, char * argv[]) {

    if (argc<3) {
        std::cerr << "use: "
                  << argv[0]
                  << " nworkers streamlen\n";
        return -1;
    }

    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);

    if (!nworkers || !streamlen) {
        std::cerr << "Wrong parameters values\n";
        return -1;
    }

    ff_farm<> farm; // farm object

    Emitter E(streamlen);
    farm.add_emitter(&E);

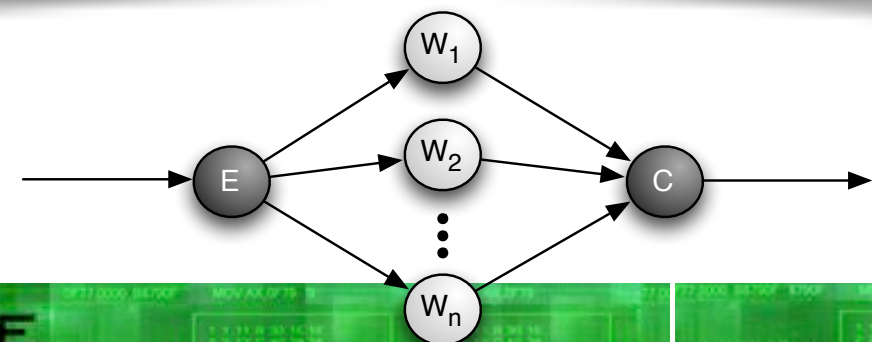
    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i) w.push_back(new Worker);
    farm.add_workers(w); // add all workers to the farm

    Collector C;
    farm.add_collector(&C);

    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
        return -1;
    }
    std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
    farm.ffStats(std::cerr);

    return 0;
}

```



+ distributed

Applications on multicore, many core & distributed platforms of multicores
Efficient and portable - designed with high-level patterns

FastFlow

Streaming network patterns

Skeletons: pipeline, map farm, reduce, D&C, ...

Arbitrary streaming networks

Lock-free SPSC/MPMC queues + FF nodes

Arbitrary streaming networks

Collective communications + FF Dnodes

Simple streaming networks

Lock-free SPSC queues + threading model

Simple streaming networks

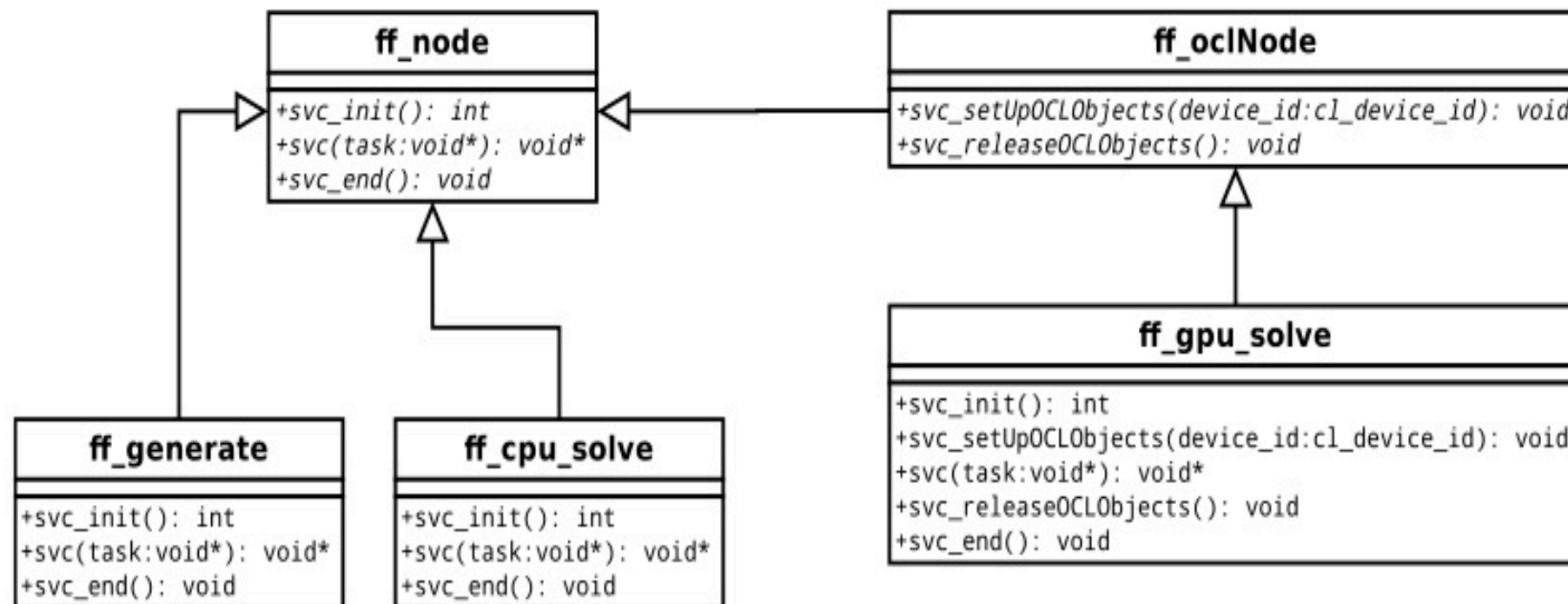
Zero copy networking + processes model

Multicore and manycore
SMP: cc-UMA & cc-NUMA

Distributed platforms
Clouds, clusters of SMPs

- * Generic `ff_node` is subclassed to `ff_dnode`
- * `ff_dnode` can support network channels
 - ◆ P2P or collective
 - ◆ used as frontier node of streaming graph
 - ◆ can be used to merge graphs across distributed platforms
- * No changes to programming model
 - ◆ at least require to “add” stub `ff_dnode`
 - ◆ when passing pointers data is serialised
 - serialisation hand-managed (zero-copy, think to Java!)

+ OpenCL (working on)



FastFlow: data-flow, not task-based



UNIVERSITÀ DEGLI STUDI
DI TORINO

- ◆ FastFlow is NOT a task based framework, focus specifically on data movements and synchronizations (shmem/distr/GPU)
- ◆ it does not expose the task concept, it rather abstracts:
 - networks of nodes (threads/processes) that can synchronize efficiently (via message passing) and move data (via shared memory or message passing)
 - predefined, OO extendable, composable patterns (i.e. networks of nodes)
- ◆ orthogonal way of thinking w.r.t. tasks
 - nodes are pinned to core, no over-provisioning, ...
- ◆ it can middleware to build your own task based framework
 - inherit lock-free synchronization mechanisms (that aren't friendly guys)
 - just create an object, and pass the pointer
 - predefined facilities to manage load-balancing, data-placement, OO-extendable

* Patterns at the high-level

- ◆ Currently as C++ templates
- ◆ Set of patterns can be extended, semantics of patterns can be changed, complexity gracefully increase with semantic distance

* Used to generate cyclic streaming networks (of threads, ...)

- ◆ Graphs, describing true data dependencies. Can be composed and transformed as graphs
- ◆ Cyclic graphs need unbound queue
- ◆ Heterogeneous cores, thread affinity, memory affinity, NUMA, can be managed while mapping graph onto the metal

Thoughts: on programming models ...

- * “Coarse grain concurrency is nearly exhausted”
- * “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”
- * “Novel computing models are needed”
 - ◆ “A computer language is not a computing model. A library is not a computing model.”
 - ◆ “System programmers should use the techniques they advocate”

Thoughts: on programming models ...

* Hardware Transactional Memory

- ◆ Intel Haswell Transactional Synchronization Extensions (TSX)
- ◆ Pragmatically orthogonal and complementary to data-flow
 - Data-flow covers (true) data-dependencies, i.e. data passing
 - Transactional memories covers concurrency in resource access

* Intel IvyBridge and PPC BlueGene (and Intel ring0)

- ◆ Wait-on-reservation
 - Blocks context on Instruction Decode
 - Can be used to overcome energy problem for active waiting on lock-free



Applications

2012: Cholesky fact vs PLASMA libraries

IEEE PDP 2012



UNIVERSITÀ DEGLI STUDI
DI TORINO

Targeting multi cores by structured programming and data flow

M. Aldinucci¹, L. Anselmi², M. Danelutto³, F. Kilpatrick¹, M. Torquati¹

¹Dept. Computer Science,
²Dept. Computer Science, U
³Dept. Computer Science, Queen

Abstract

Data flow techniques have been around for
years and in computers for sequential languages
despite they were also considered as a possible
way, although the impact here was limited. It
has been identified as a candidate for efficient
programming models on multi-core architectures
the burden of determining data flow 'natural'
progression, while the compiler/runtime is
responsible of their execution. We discuss
programming approach supporting automatic

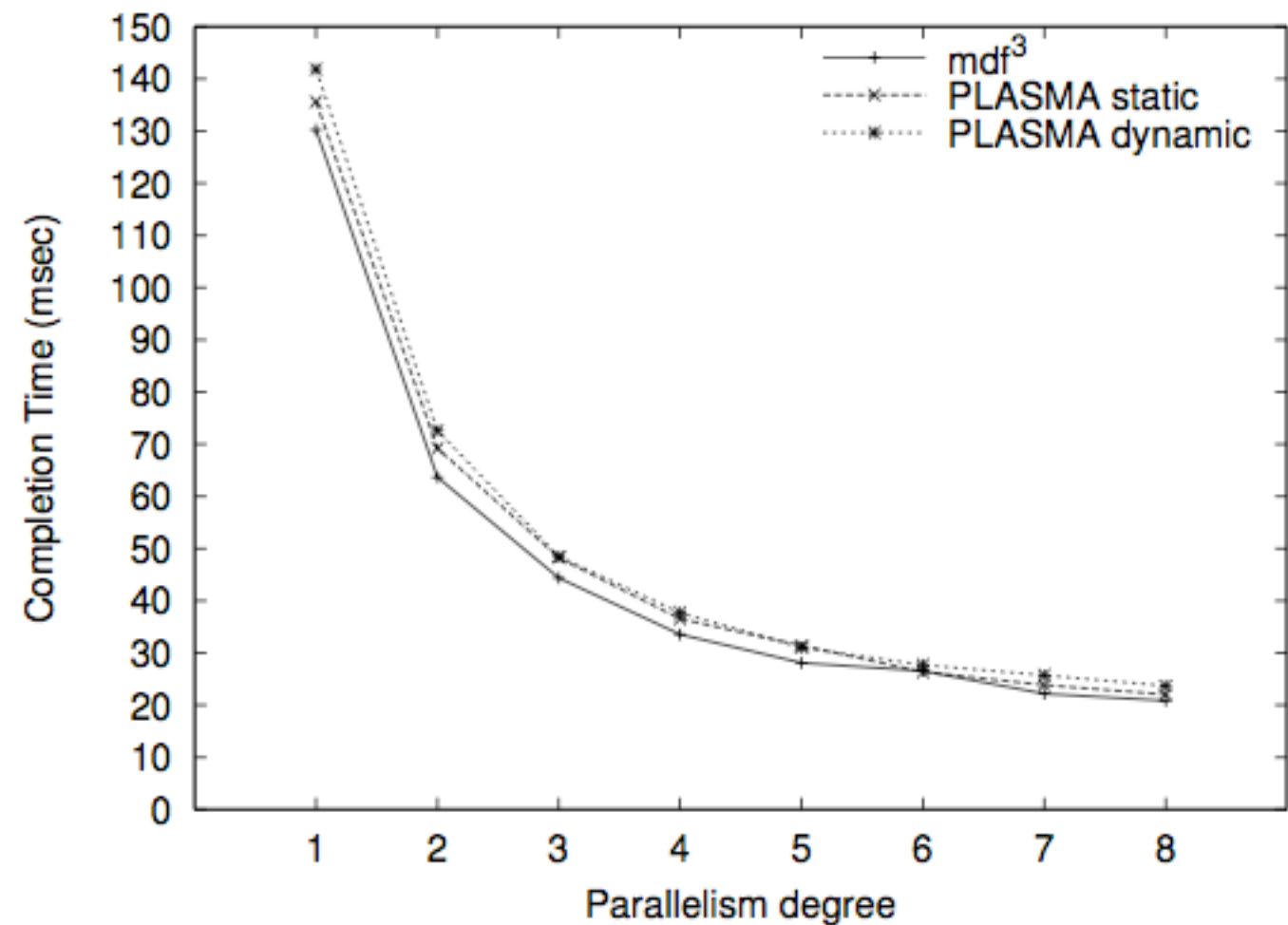


Figure 10: *mdf³* vs PLASMA library. Cholesky factorization for a single 1024x1024 complex matrix (Intel Nehalem).

C4.5 (Fine grain - irregular D&C) PKDD 2011



UNIVERSITÀ DEGLI STUDI
DI TORINO

Decision Tree Growing and Pruning on General-Purpose Multicore

Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati

Abstract—The whole computer hardware industry embraced multicores. The extreme optimisation of sequential algorithms is then no longer sufficient to squeeze the real machine power, which can be only exploited via thread-level parallelism. Decision tree algorithms exhibit natural concurrency that makes them suitable to be parallelised. This paper presents an in-depth study of the parallelisation of an implementation of the C4.5 algorithm for multicore architectures. In addition to the tree growing phase, we cover also the so far unaddressed problem of parallelising the error-based pruning-with-grafting phase. We characterize elapsed time lower bounds for the forms of parallelisations adopted, and achieve close to optimal performances. Our implementation is based on the FastFlow parallel framework.

Index Terms—Parallel decision tree

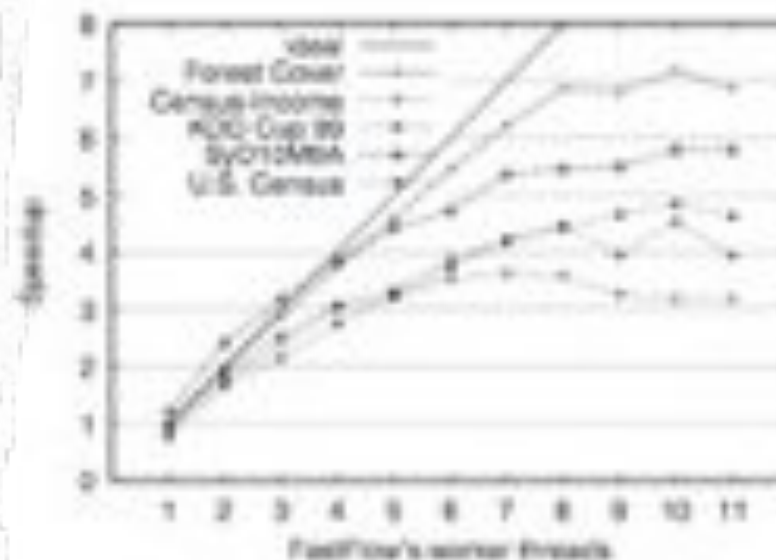


Fig. 11: NAP strategy speedup.

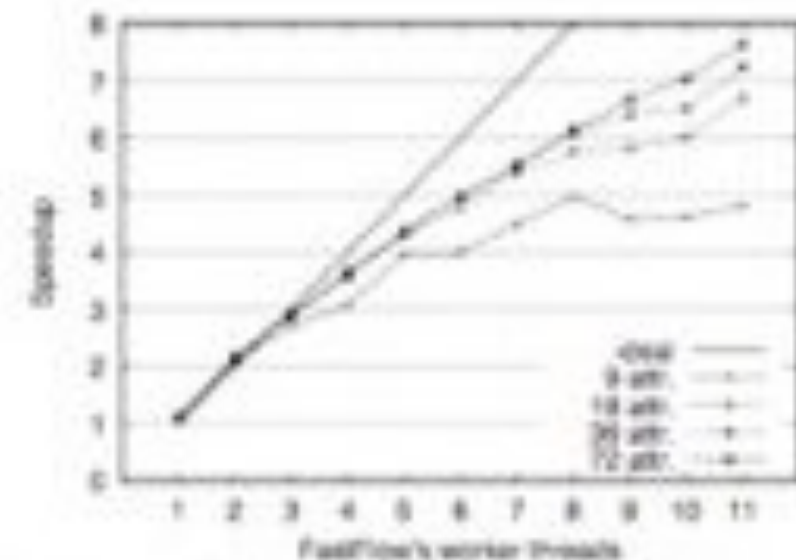
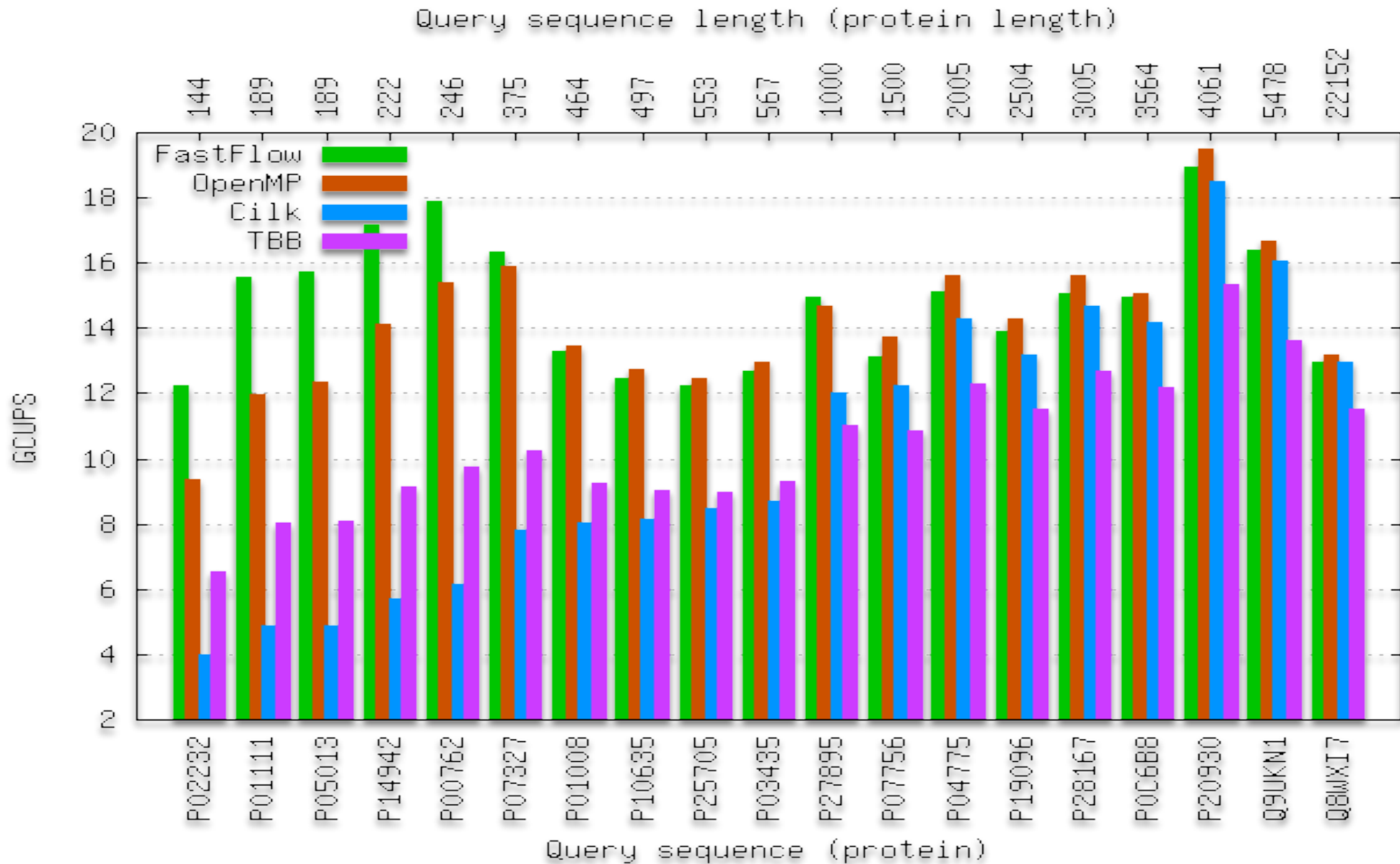


Fig. 12: Speedup of the NAP strategy vs no. of attributes (for 1M sample subset of SCD10M9A1).

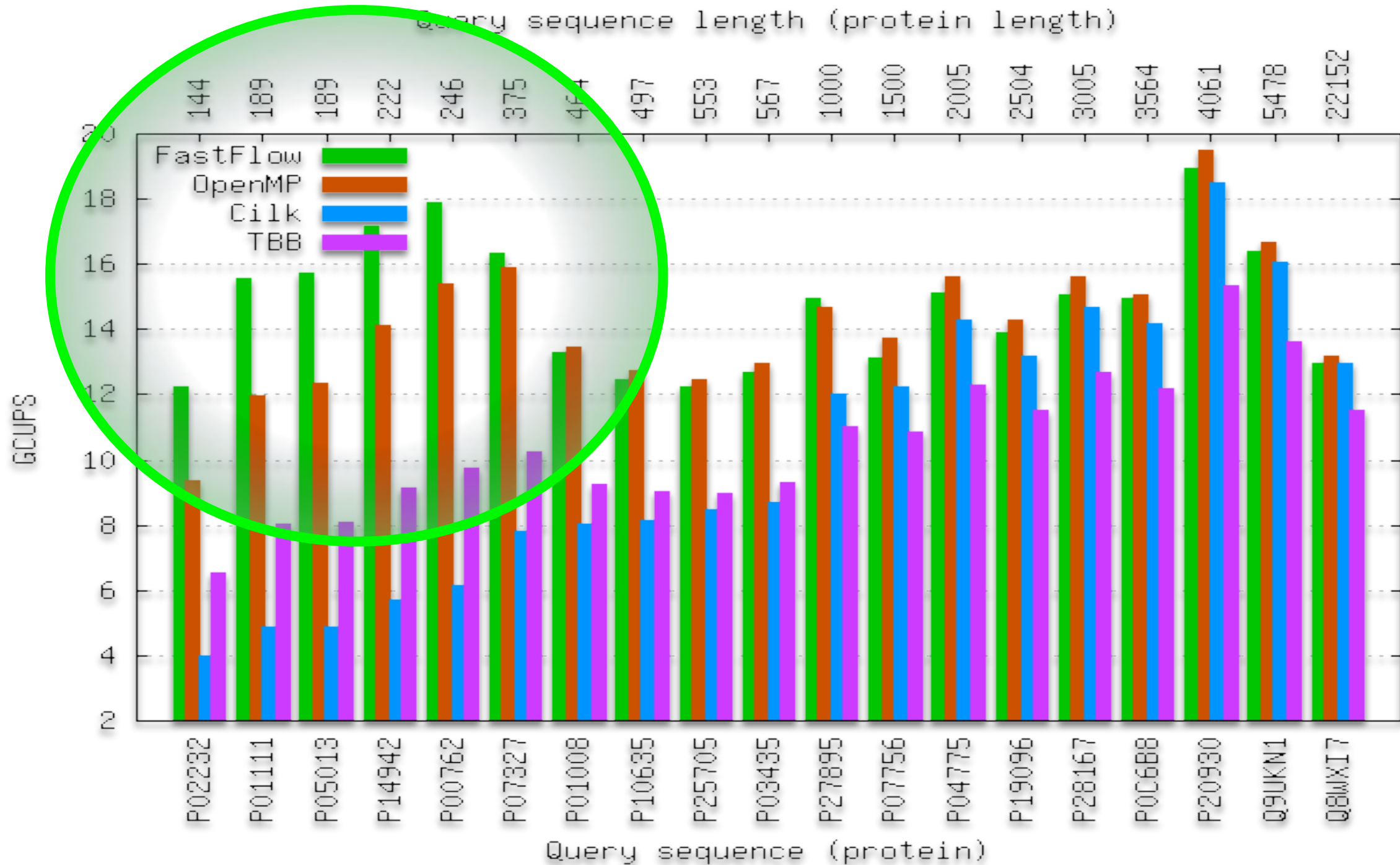
Smith Waterman vs TBB vs OpenMP vs Cilk



UNIVERSITÀ DEGLI STUDI
DI TORINO



Smith Waterman vs TBB vs OpenMP vs Cilk



Two-phase denoising

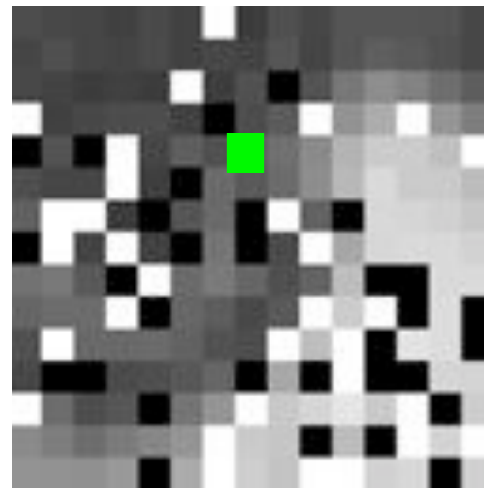
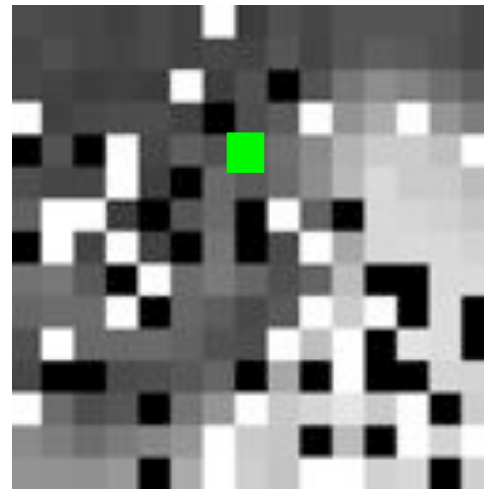
IEEE IPTA 2012



```
map p in pixels detect
  while (winsize < MAX)
    if (homogenous(p, winsize))
      winsize++;
    else if isImpluse(p) return NOISY;
  return NOT_NOISY;
```



```
while !fixpoint denoise
  map u in N (noisy pixels)
    new_u = value_that_minimize F(u);
  reduce (u in N, new_u in NEW_N, diff);
```



Adaptive median filter

different pixels are
independent and can
be easily processed
in parallel
pixels are read-only

Iterative variational method

answer to the question:
which is the greyscale level
that better “integrate” in
the surrounding
(i.e. keeps edges)
at each iteration an
approximation of restored
pixels is available

$$F_d|_N(u) = \sum_{(i,j) \in N} [|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)]$$

$$\square(t) = |t|^\alpha \text{ with } 1 < \alpha \leq 2$$

$$S_1 = \sum_{(m,n) \in V_{i,j} \cap N} 2 \cdot \square(u_{i,j} - d_{m,n})$$

$$S_2 = \sum_{(m,n) \in V_{i,j} \cap N^c} \square(u_{i,j} - u_{m,n})$$

Two-phase denoising

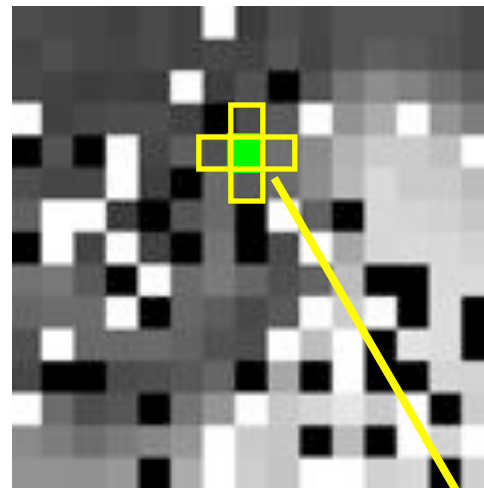
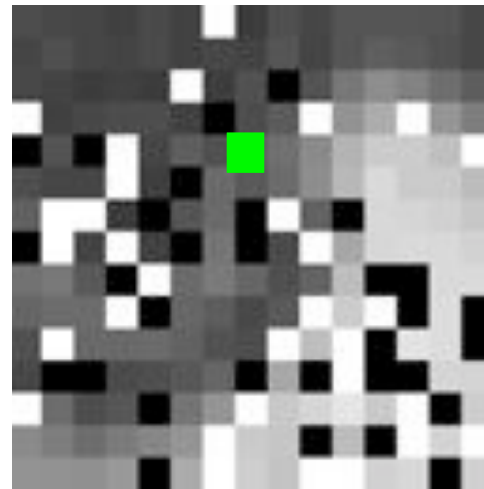
IEEE IPTA 2012



```
map p in pixels detect
  while (winsize < MAX)
    if (homogenous(p, winsize))
      winsize++;
    else if isImpluse(p) return NOISY;
  return NOT_NOISY;
```



```
while !fixpoint denoise
  map u in N (noisy pixels)
    new_u = value_that_minimize F(u);
  reduce (u in N, new_u in NEW_N, diff);
```



Adaptive median filter

different pixels are independent and can be easily processed in parallel
pixels are read-only

Iterative variational method

answer to the question: which is the greyscale level that better “integrate” in the surrounding (i.e. keeps edges)
at each iteration an approximation of restored pixels is available

$$F_d|_N(u) = \sum_{(i,j) \in N} [|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2)]$$

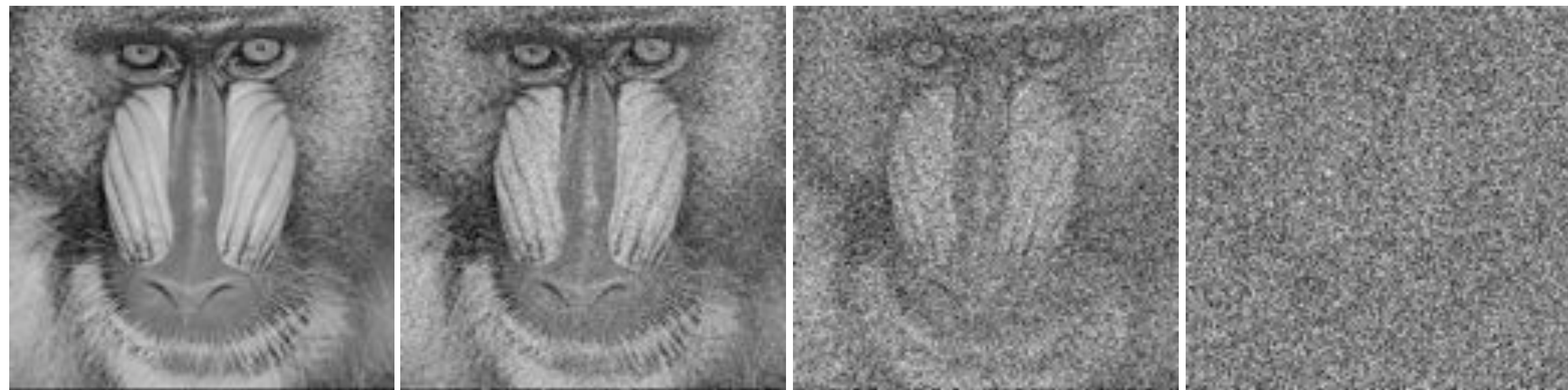
$$\square(t) = |t|^\alpha \text{ with } 1 < \alpha \leq 2$$

$$S_1 = \sum_{(m,n) \in V_{i,j} \cap N} 2 \cdot \square(u_{i,j} - d_{m,n})$$

$$S_2 = \sum_{(m,n) \in V_{i,j} \cap N^c} \square(u_{i,j} - u_{m,n})$$

Two-phase denoising

IEEE IPTA 2012 (Istanbul, 15-18 Oct)



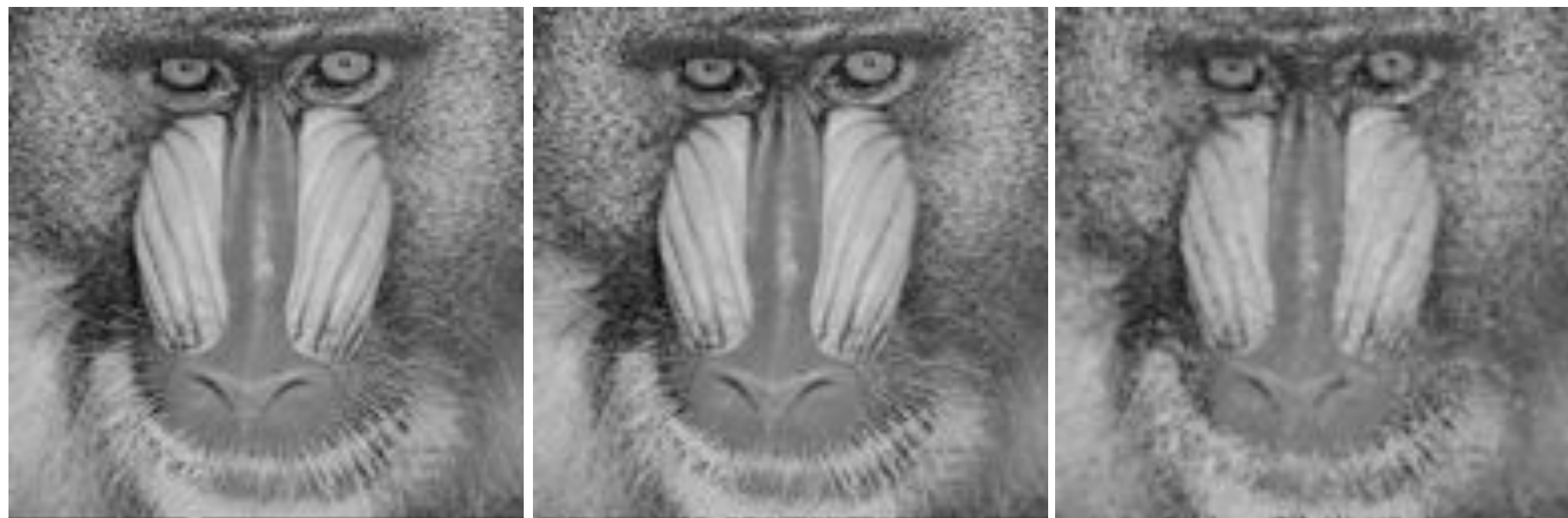
Original
Baboon standard
test image
1024x1024

10% impulsive noise

50% impulsive noise

90% impulsive noise

Restored



PNSR 43.29dB MAE 0.35

PNSR 32.75dB MAE 2.67

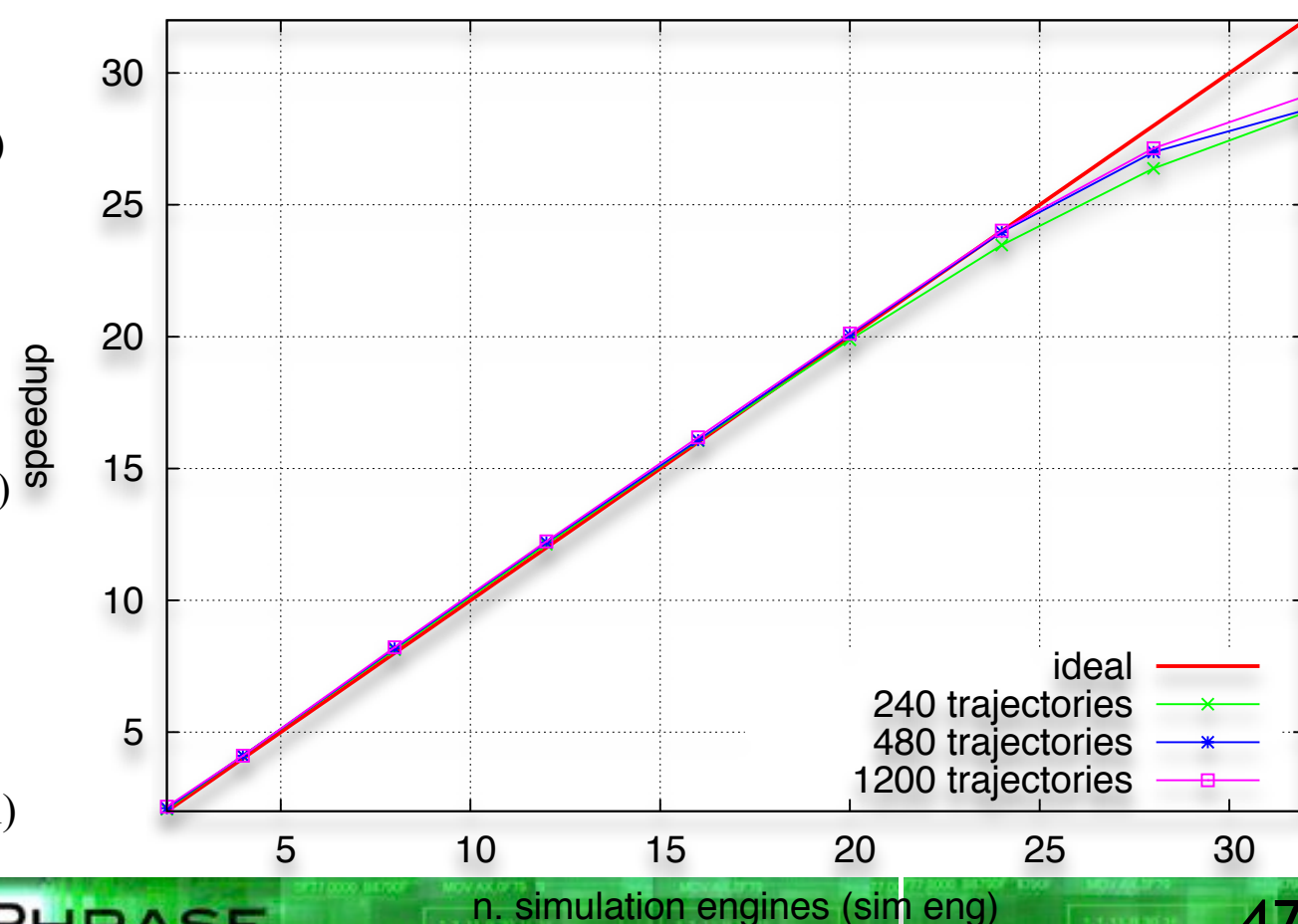
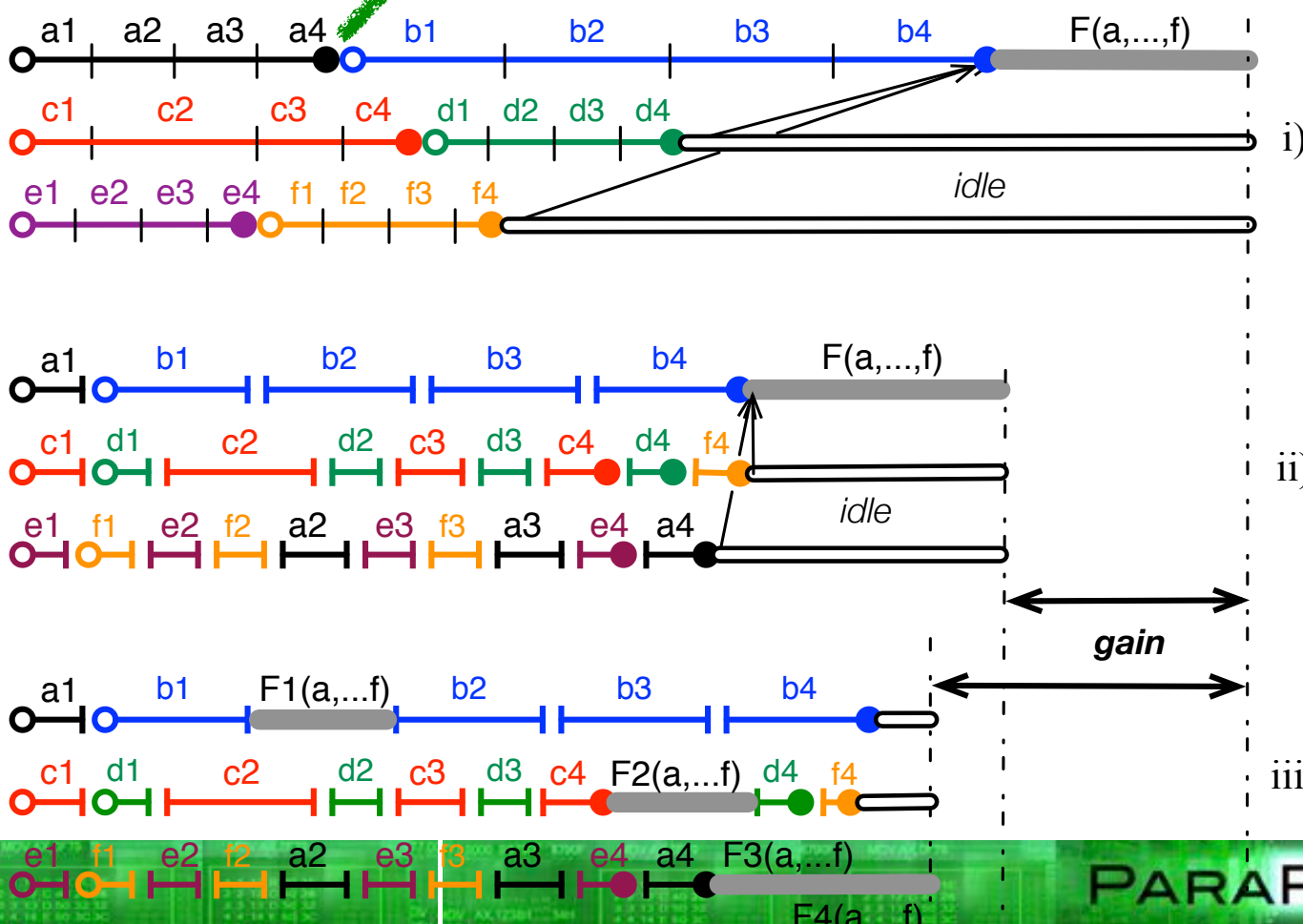
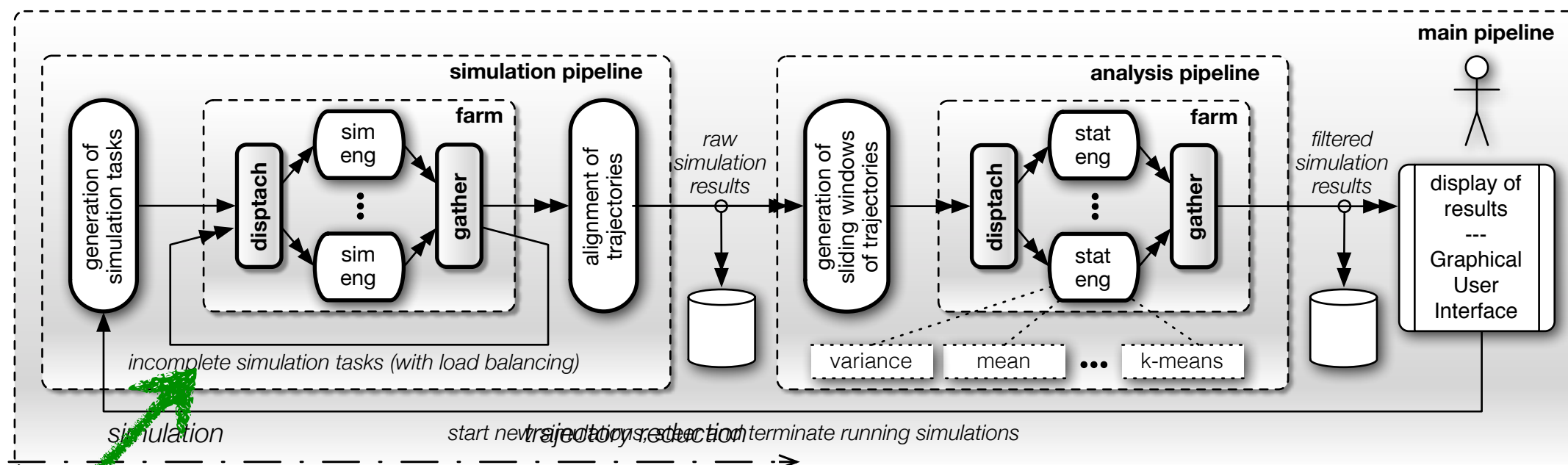
PNSR 23.4 MAE 11.21

Parallel stochastic sim for system biology

IEEE PDP 2011, HiBB 2011, Briefings in Bioinformatics (invited), Bio-IT world (invited), IEEE PDP 2013 (submitted), BMC Bioinformatics



UNIVERSITÀ DEGLI STUDI
DI TORINO



PARAPHRASE

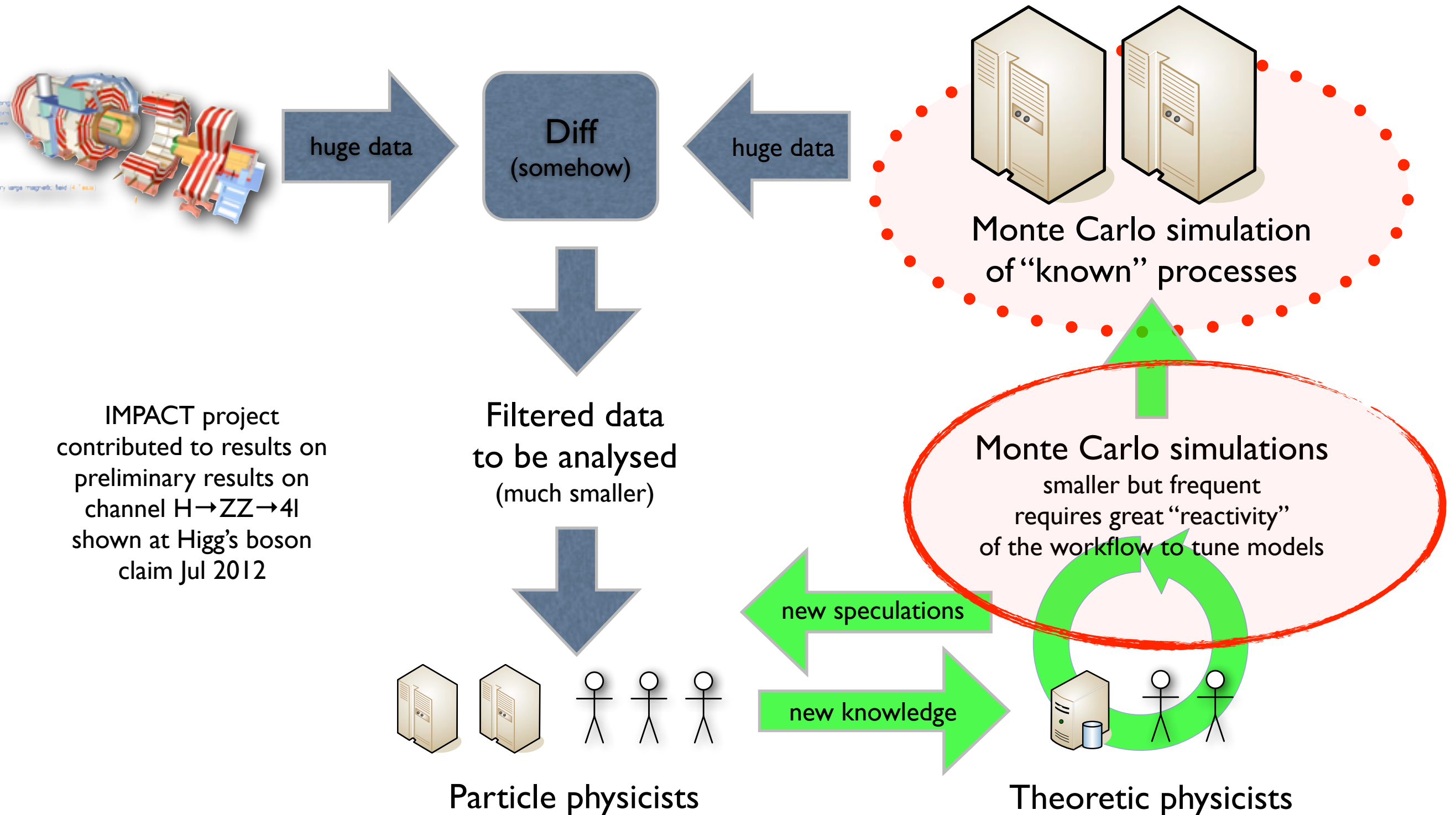
n. simulation engines (sim eng)



Innovative Methods for Particle Colliders at the Terascale (2012-2015, oversimplified vision)



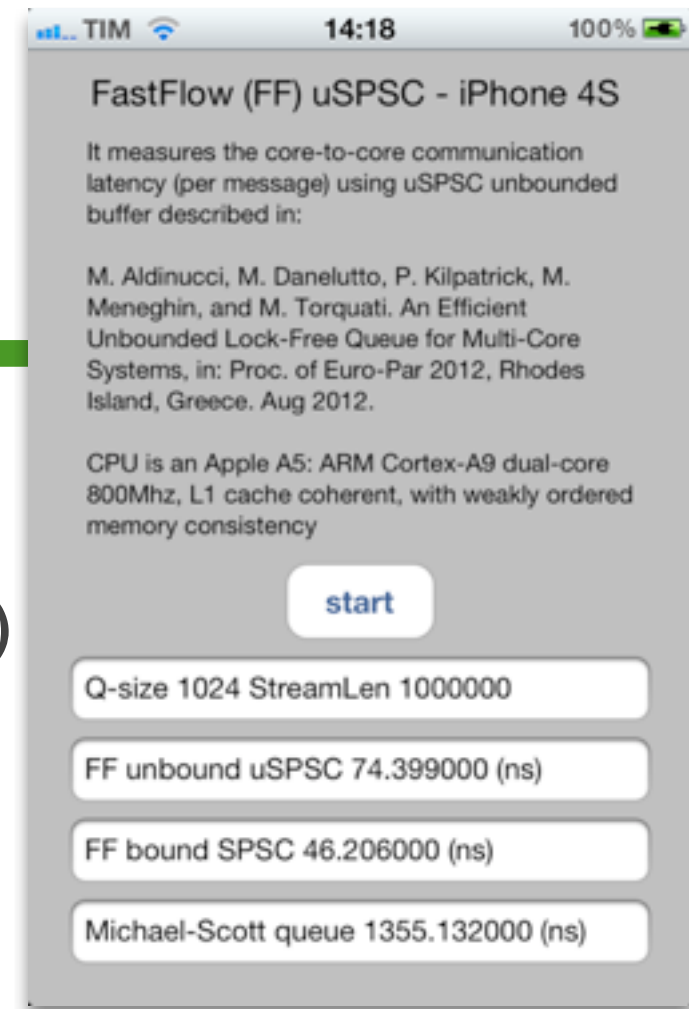
1 postdoc position open
2 years, about 1800 Euro/month after taxes



Conclusions

* FastFlow C++ pattern-based framework

- ◆ A tiny & open research framework (5K lines codebase)
- ◆ 3 years old - over 8K downloads - 40K web contacts
- ◆ x86/PPC/ARM + Linux/Mac/Win/iOS
 - Adopted as one run-time technology in ParaPhrase EU-FP7 STREP
 - a laboratory to experiment new run-time solutions
 - GPGPU integration (working on), Infiniband RDMA integration (working on), HW blocking reads (thinking on), HW transactional mem (thinking on) ...
 - Stream-specific parallel memory allocator: fast (but still under test)
- ◆ Data-centric, focus on messaging and synchronization, thread model agnostic
- ◆ High-level = performance & portability
 - Speedup starting from ~20 clock cycles workload on standard x86_64 (TBB >20K)
 - Tested on dozen of apps, comparable or faster than TBB/OpenMP
 - <http://di.unito.it/fastflow>



Thank you

* Paraphrase

- ◆ Parallel Patterns for Adaptive Heterogeneous Multicore Systems
- ◆ EU-FP7 STREP, 2011-2014, FastFlow is the background technology

* IMPACT

- ◆ Innovative Methods for Particle Colliders at the Terascale
- ◆ National, 2012-2015, FastFlow is the background technology

* HiPEAC

- ◆ High Performance and Embedded Architecture and Compilation
- ◆ EU NOE, 2012-2016

* BETTY

- ◆ Behavioral Types for Reliable Large-Scale Software Systems
- ◆ EU Cost Action, 2012-2016

* CINA

- ◆ CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society
- ◆ MIUR PRIN, 2012-2014

