UNIVERSITÀ DEGLI STUDI DI TORINO

UPMARC Workshop on Task-Based Parallel Programming
Uppsala 2012

September 28, 2012
Uppsala, Sweden

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

Marco Aldinucci - Uni. of Torino, Italy

Massimo Torquati and Marco Danelutto - Uni. Pisa, Italy
Massimiliano Meneghin - IBM Research, Ireland
Peter Kilpatrick - Queen's Uni. Belfast, U.K.
Maurizio Drocco - Uni. Torino, Italy

SEVENTH FRAMEWORK PROGRAMME

PARAPHRASE

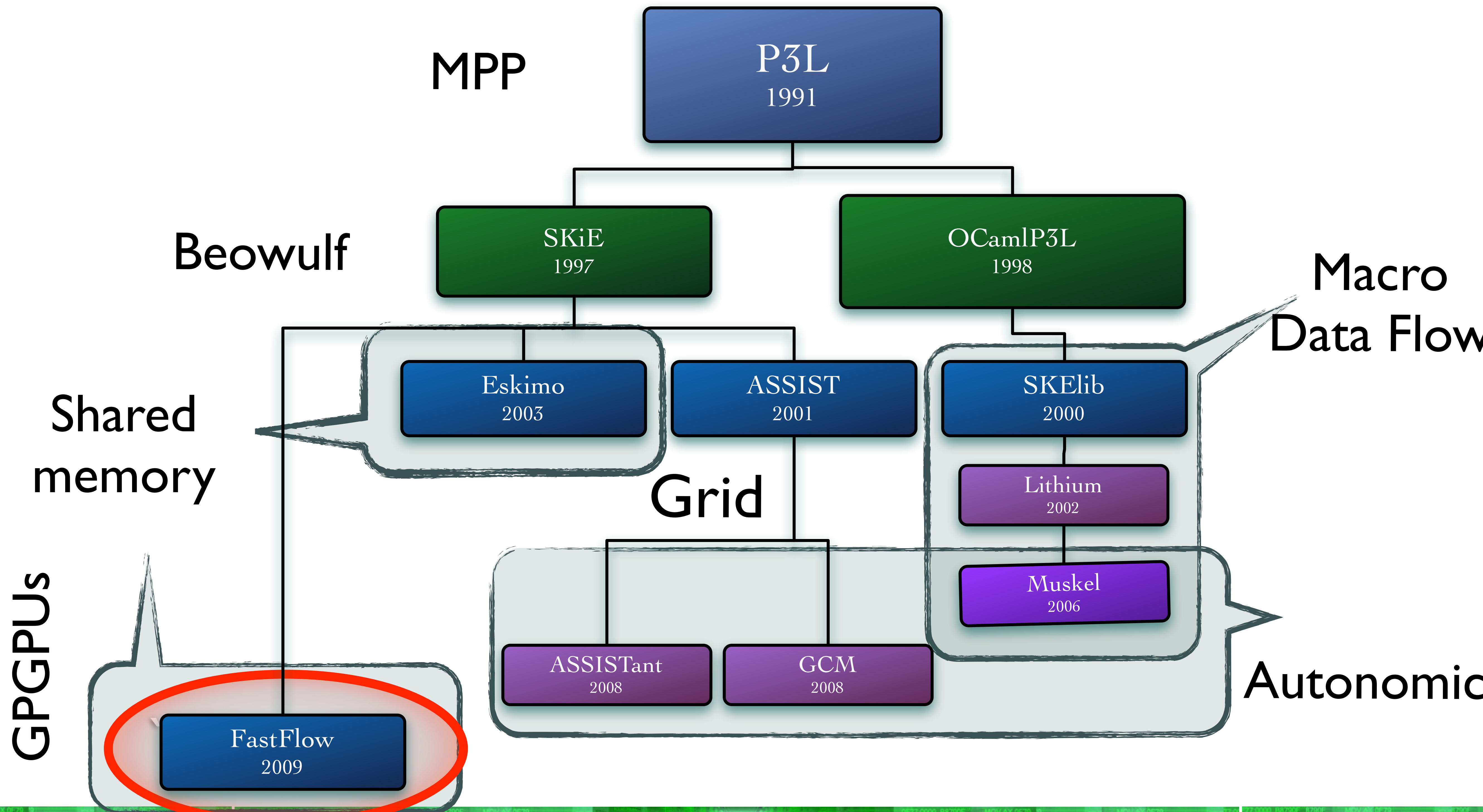
Outline

UNIVERSITÀ DEGLI STUDI DI TORINO

- 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

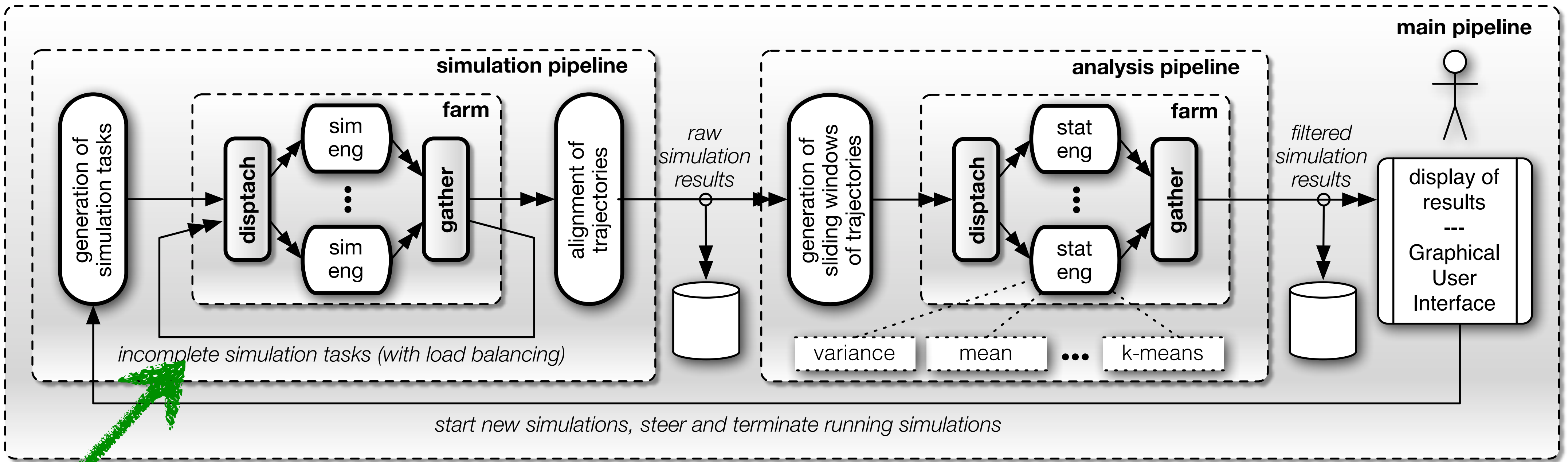
UPMARC workshop on task based programming ...

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

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



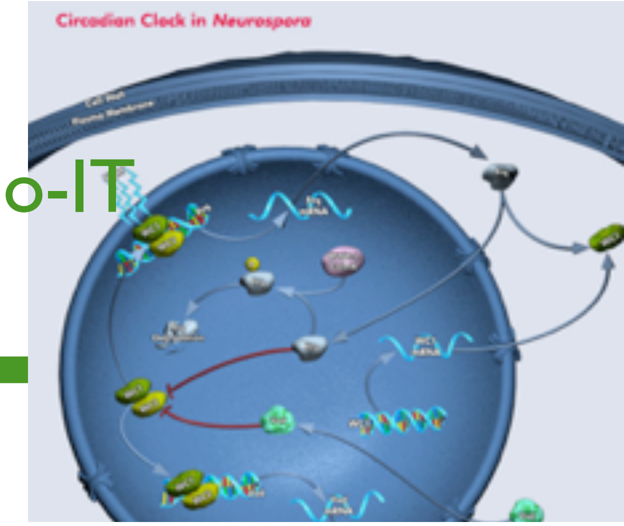
DSL task engine

task size

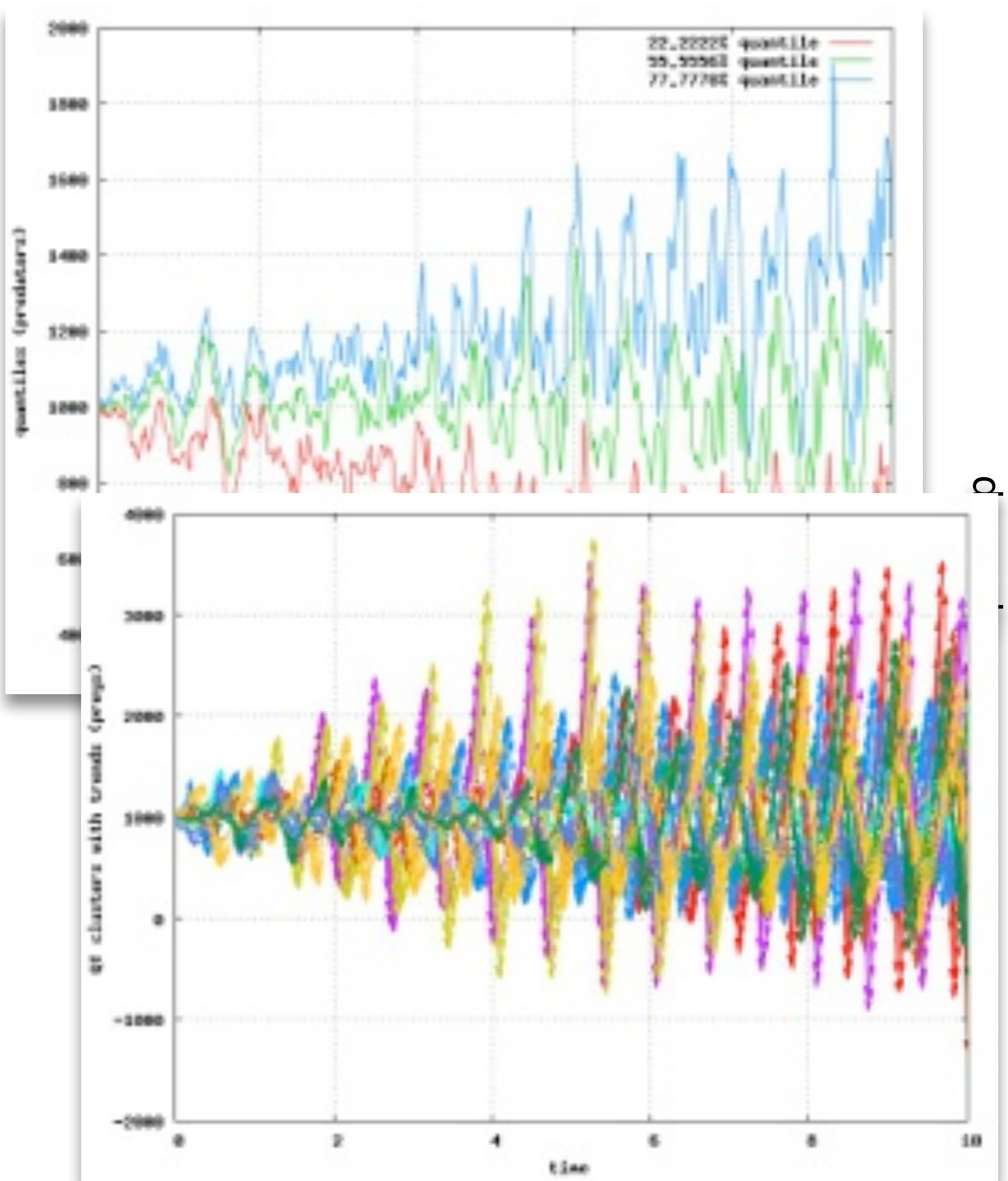
Table 2 - Performance (Intel 32 core platform)

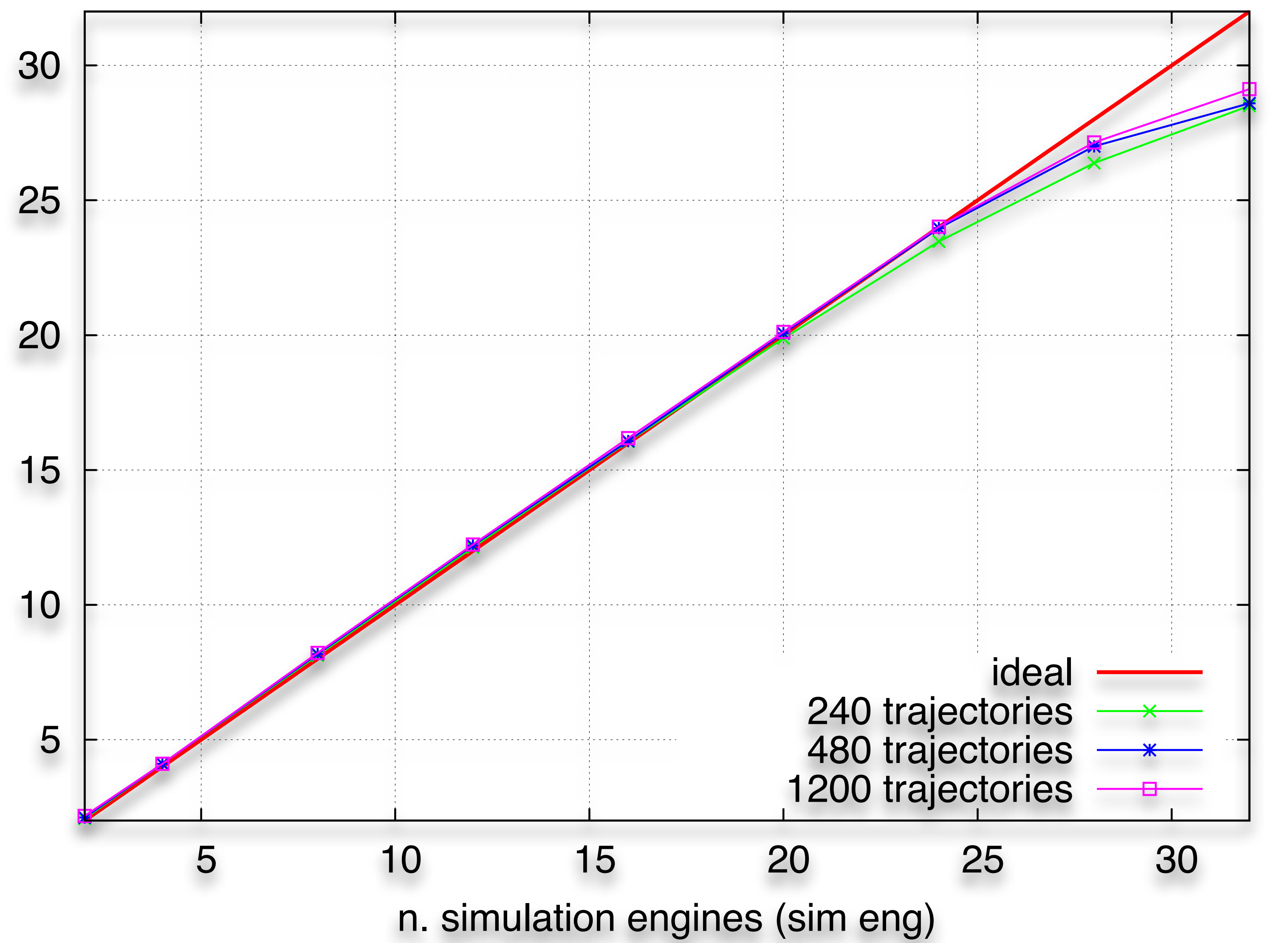
Model	Single trajectory information			Overall data (20 sim eng, 3 stat eng)		
	N. samples	Avg sim step	Sample time	Inter-arrival time	Throughput	Output size
Neurospora	10^4	7.80 μs	517.24 μs	25.86 μs	11.87 MB/s	36.62 MB
Neurospora	10^5	8.37 μs	55.51 μs	2.78 μs	11.98 MB/s	366.21 MB
Neurospora	10^6	75.63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





Micro-benchmarks: farm of tasks

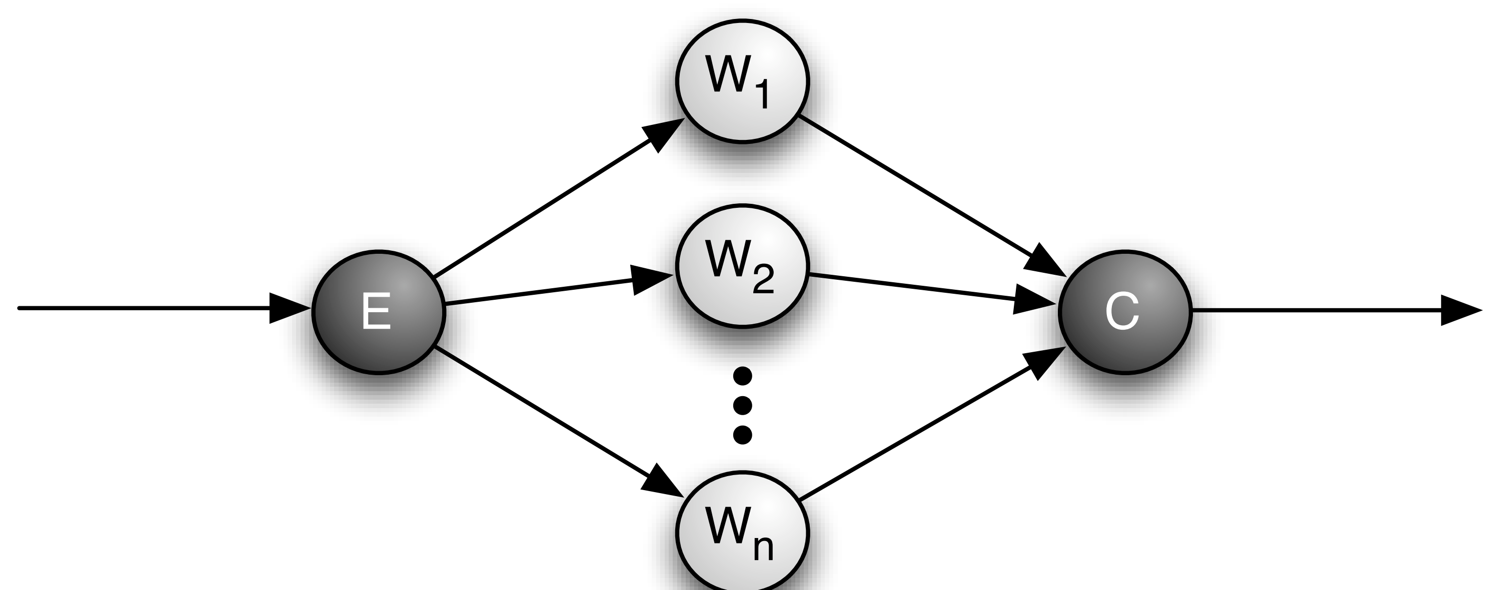
UNIVERSITÀ DEGLI STUDI DI TORINO

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);
  }
}
```

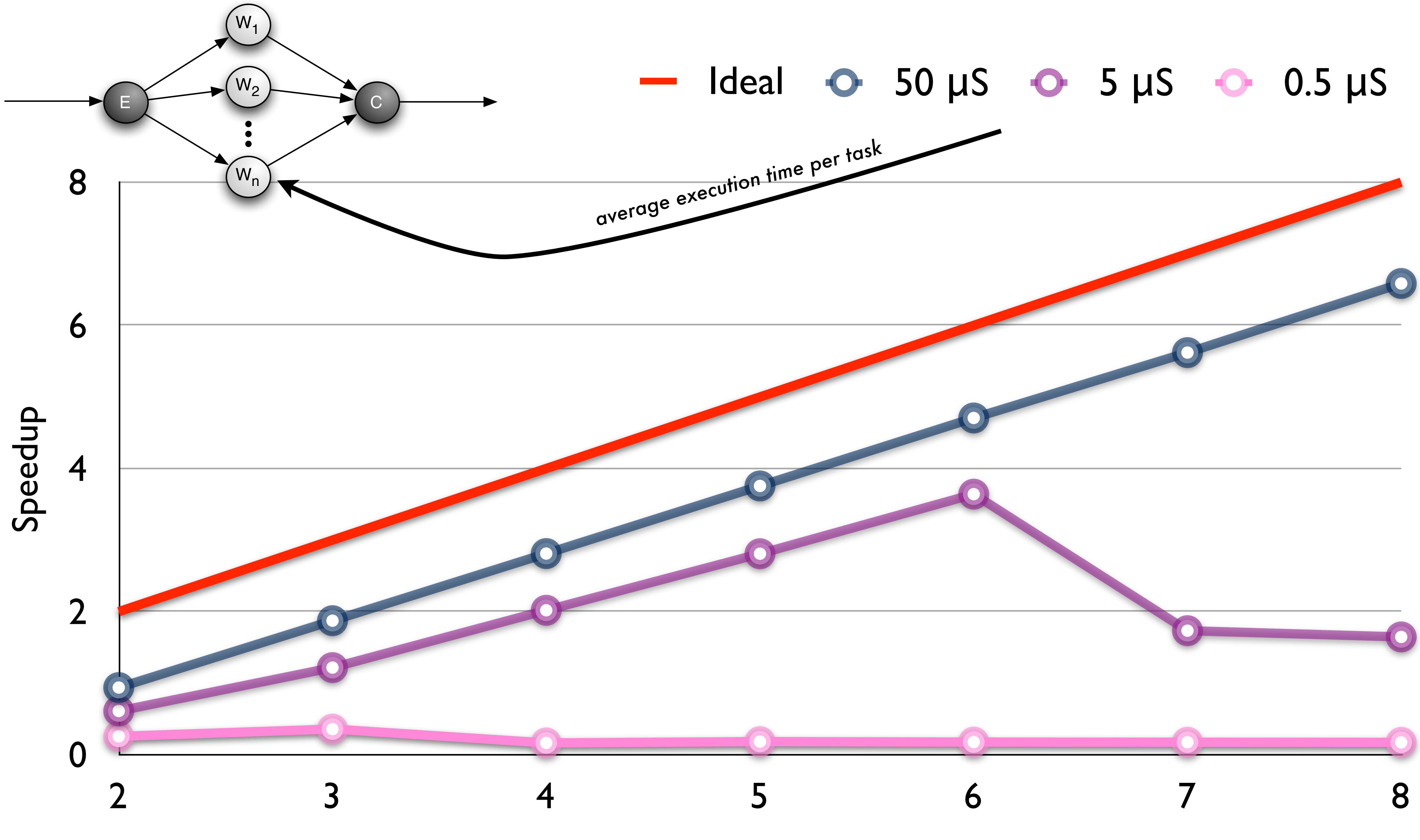
```
int main () {
  spawn_thread( Emitter ) ;
  for ( i =0; i <nworkers;++i){
    spawn_thread(Worker);
  }
  wait_end () ;
}
```

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



Task farm with POSIX lock/unlock

UNIVERSITÀ DEGLI STUDI DI TORINO

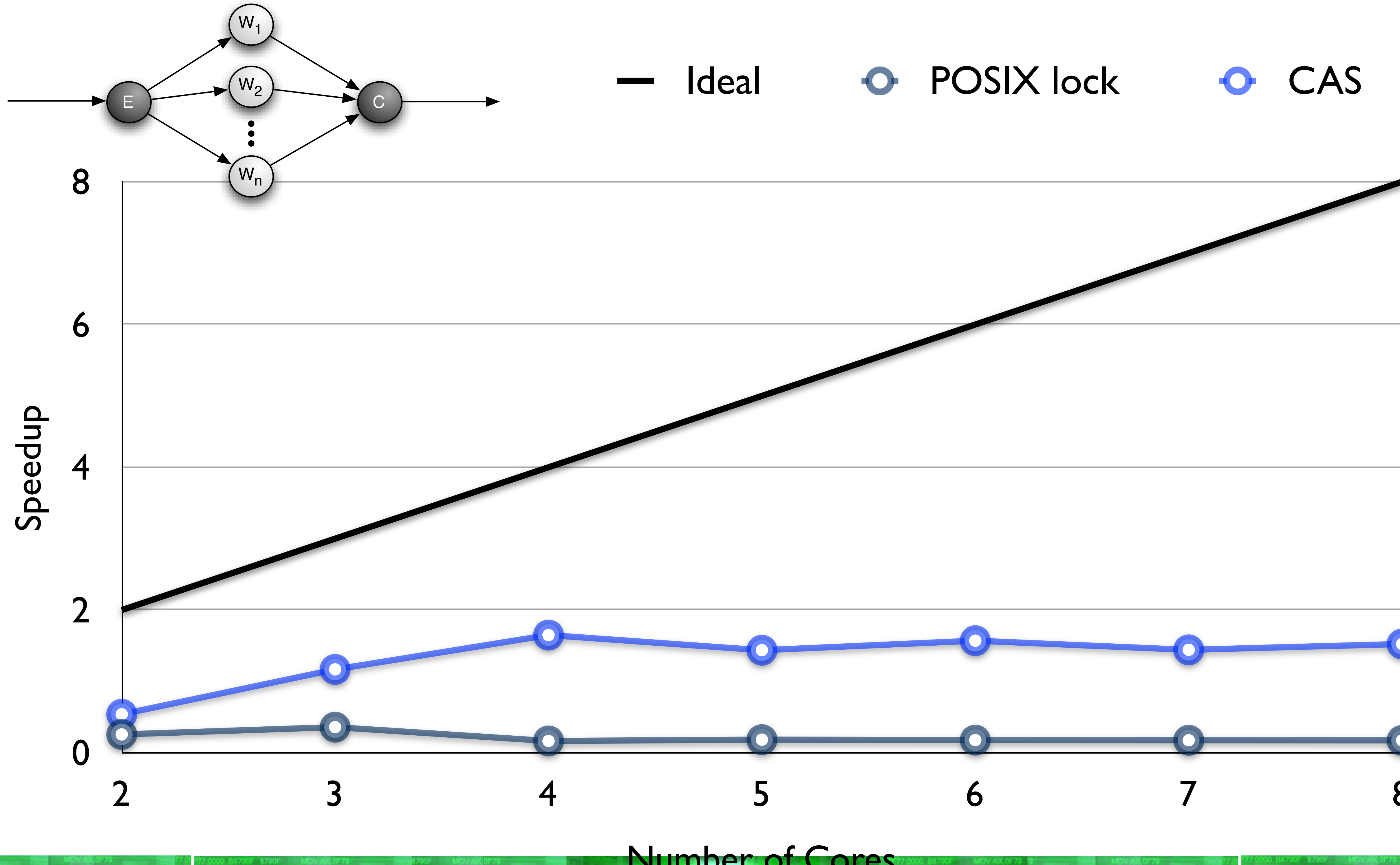


Can we avoid locks?

- 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 μS)

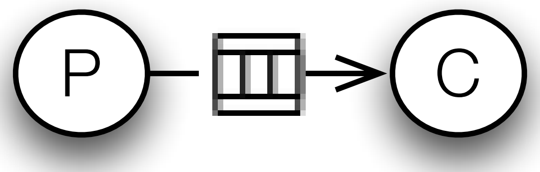
UNIVERSITÀ DEGLI STUDI
DI TORINO



Re-starting from the basics

UNIVERSITÀ DEGLI STUDI
DI TORINO

- 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

UNIVERSITÀ DEGLI STUDI DI TORINO

- 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

UNIVERSITÀ DEGLI STUDI DI TORINO

- 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

UNIVERSITÀ DEGLI STUDI
DI TORINO

- What are the performance implications of the progress properties ?
- For medium/coarse grain applications:
  - Blocking faster than Non-Blocking
- For fine grain applications:
  - Non-Blocking faster than Blocking
  - Obstruction-Free faster than Lock-Free faster than Wait-Free
- In the general case:
  - Stronger properties are harder to maintain

several task-based approaches are here

I'm focusing here

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

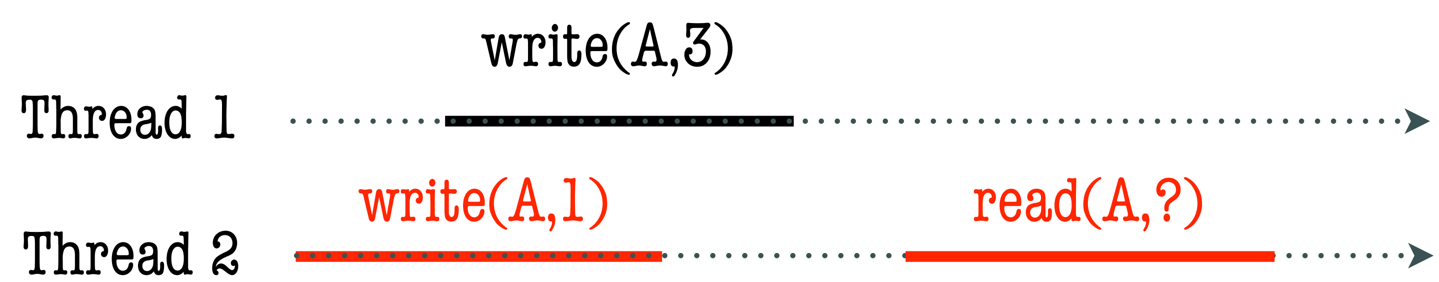
UNIVERSITÀ DEGLI STUDI DI TORINO

- Single-Producer-Single-Consumer FIFO queues
  - Lamport et al. 1983 Trans. PLS (Sequential consistency only - in memory)
  - Higham and Kavalsh. 1997 ISPAN (P1C1 - 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 ➠ Cannot be done
  - Nonblocking without CAS - with mediator thread ➠ FastFlow

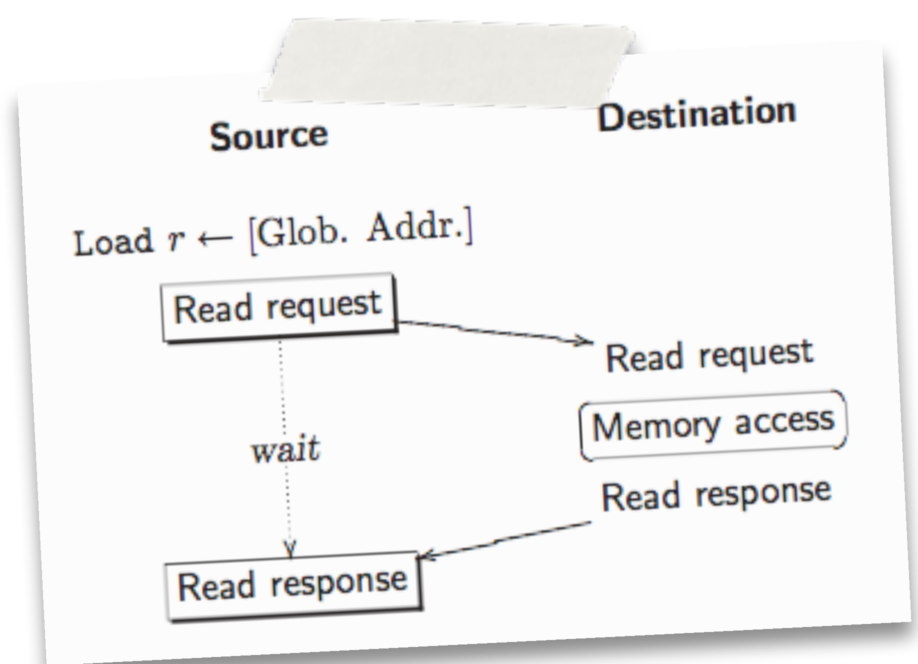
Recap: coherence and consistency

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

UNIVERSITÀ DEGLI STUDI
DI TORINO

```
push_nonbocking(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_nonbocking(data) {
  if (NULL != buffer[head]) {
    return EWOULDBLOCK;
  }                                  (WMB)
  buffer[head] = data;               For any
  head = NEXT(head);                 model
  return 0;                          weaker
}                                    than TSO

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

FastFlow FIFO

derived from P1C1 (Higham and Kavalsh, ISPAN 1997)
& FastForward (Giacomoni et al, PPoPP 2008)

FastFlow SPSC queues

UNIVERSITÀ DEGLI STUDI
DI TORINO

```
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 1 cache miss every push and pop (at least)

Lamport FIFO - 1983

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

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

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

excluding "true" deps

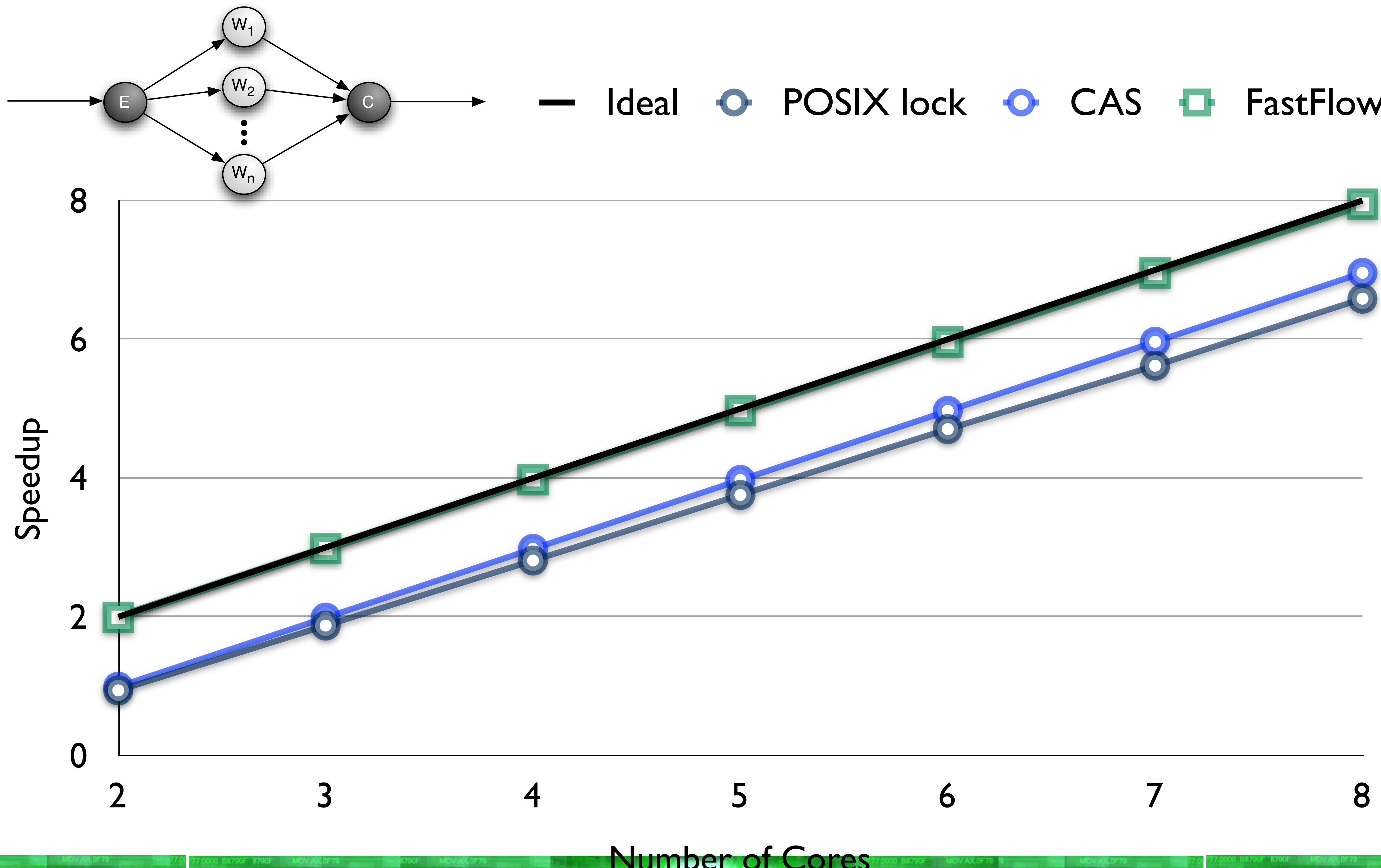
extended domain
on void *

FastFlow FIFO

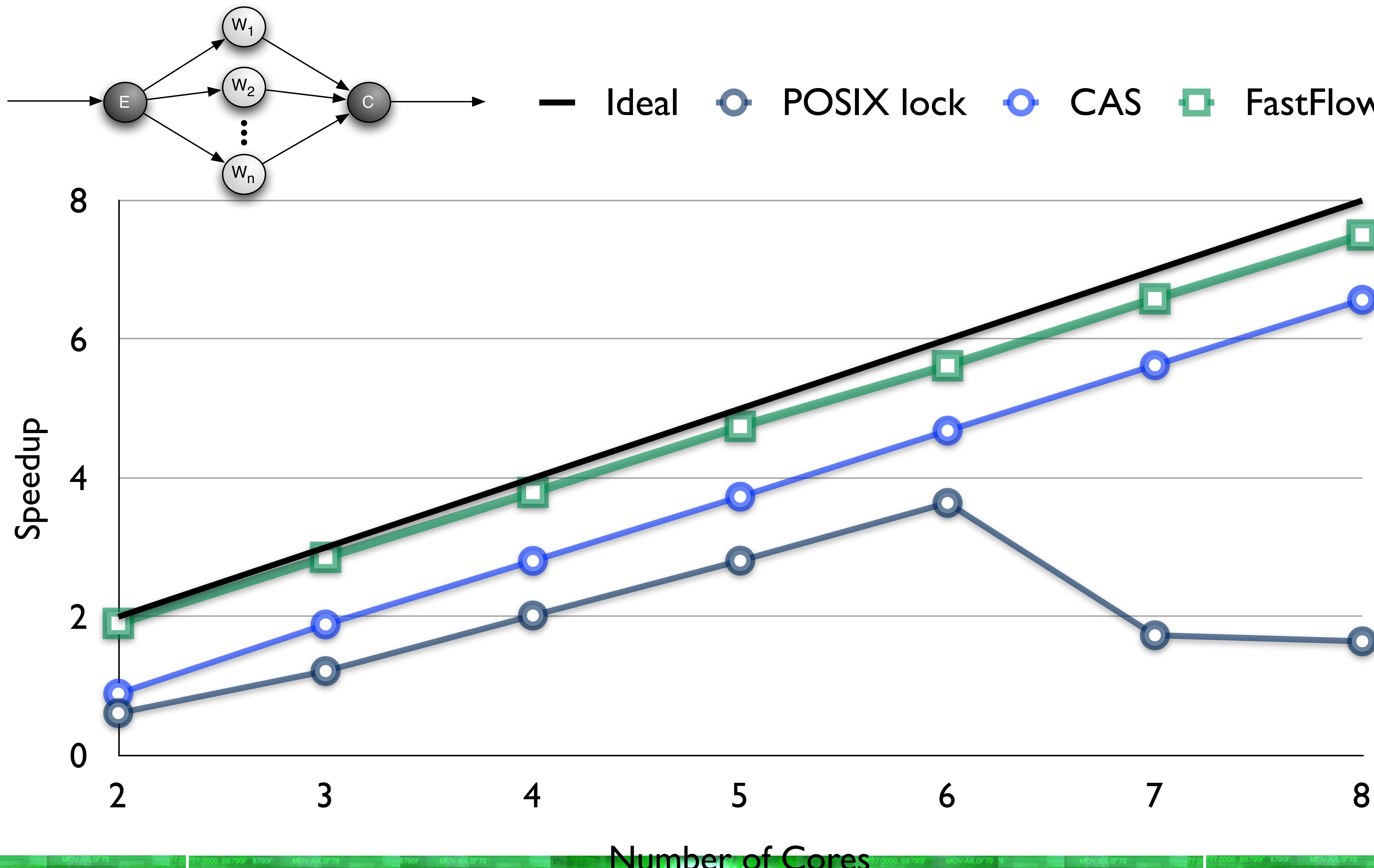
derived from P1C1 (Higham and Kavalsh, ISPAN 1997)
& FastForward (Giacomoni et al, PPoPP 2008)

Lock vs CAS vs SPSC FastFlow (50 μS)

UNIVERSITÀ DEGLI STUDI
DI TORINO

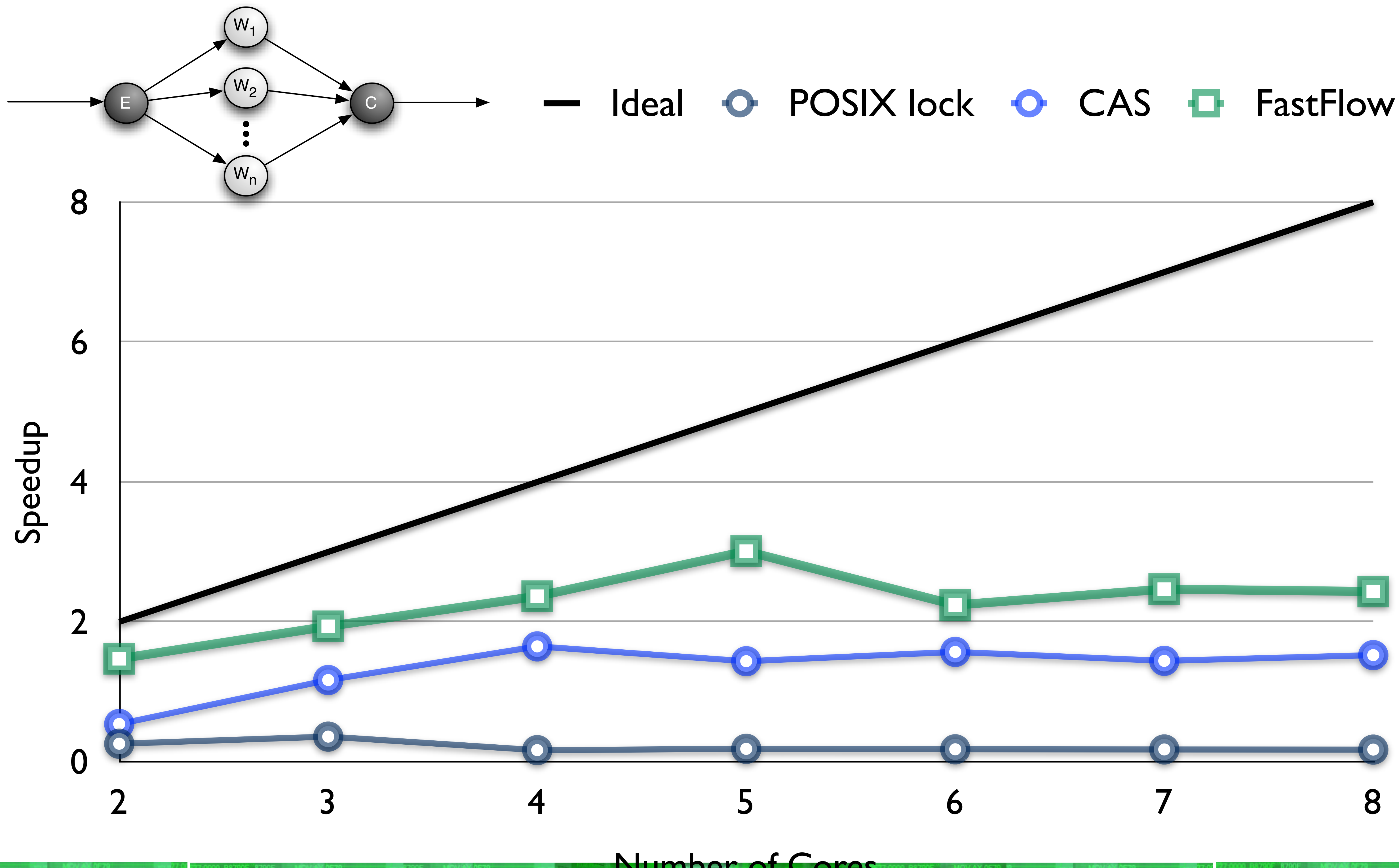


Lock vs CAS vs SPSC FastFlow (5 μS)



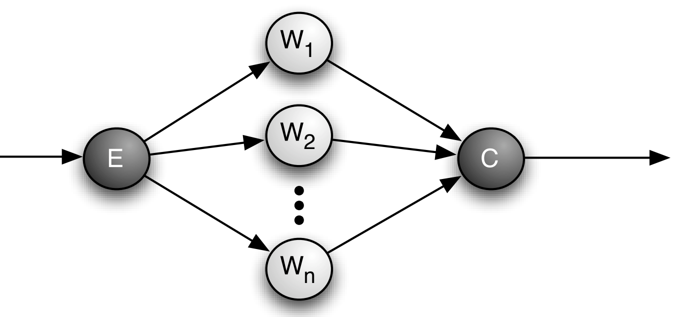
Lock vs CAS vs SPSC FastFlow (0.5 μS)

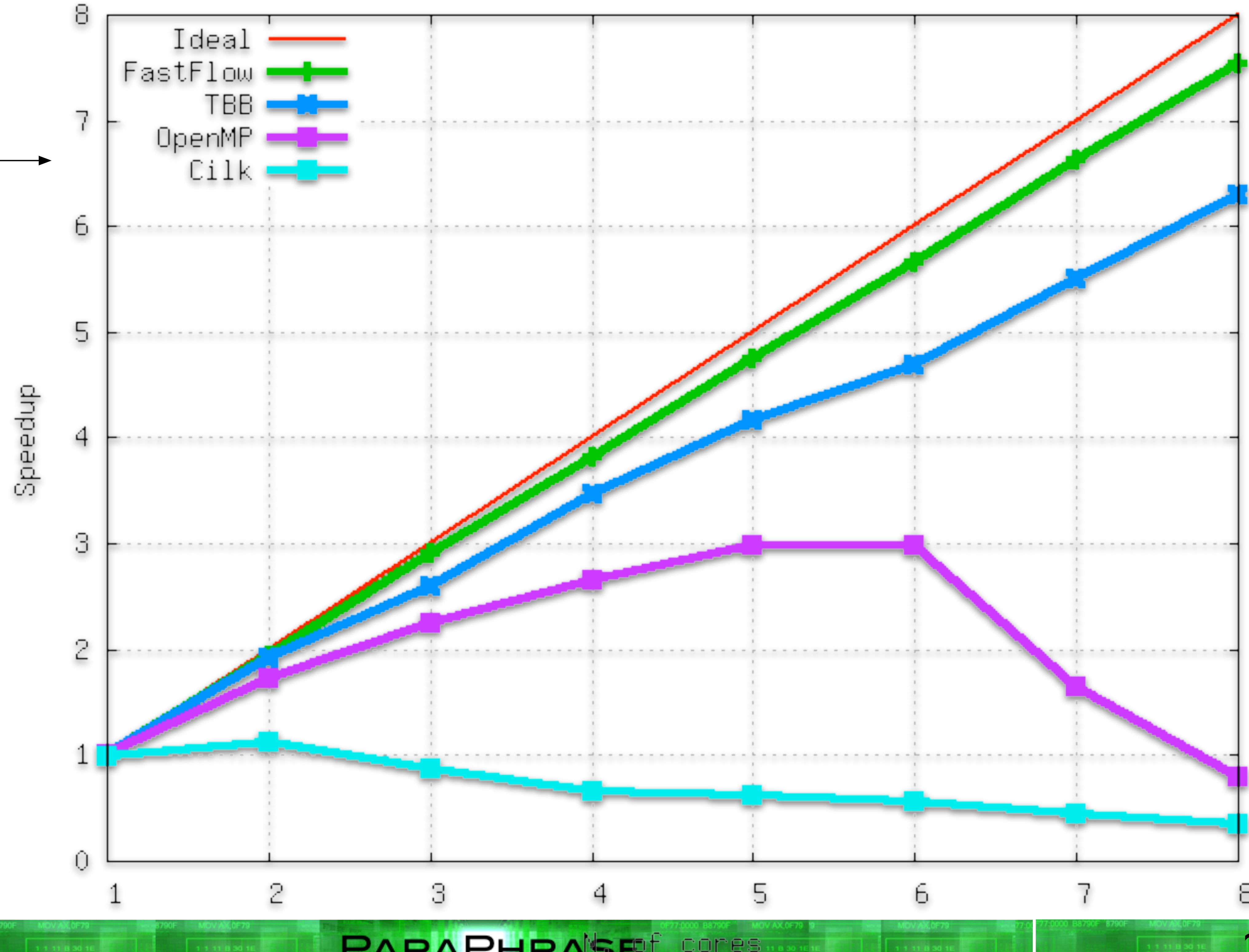
UNIVERSITÀ DEGLI STUDI
DI TORINO



Medium grain (5 μS workload)

UNIVERSITÀ DEGLI STUDI
DI TORINO





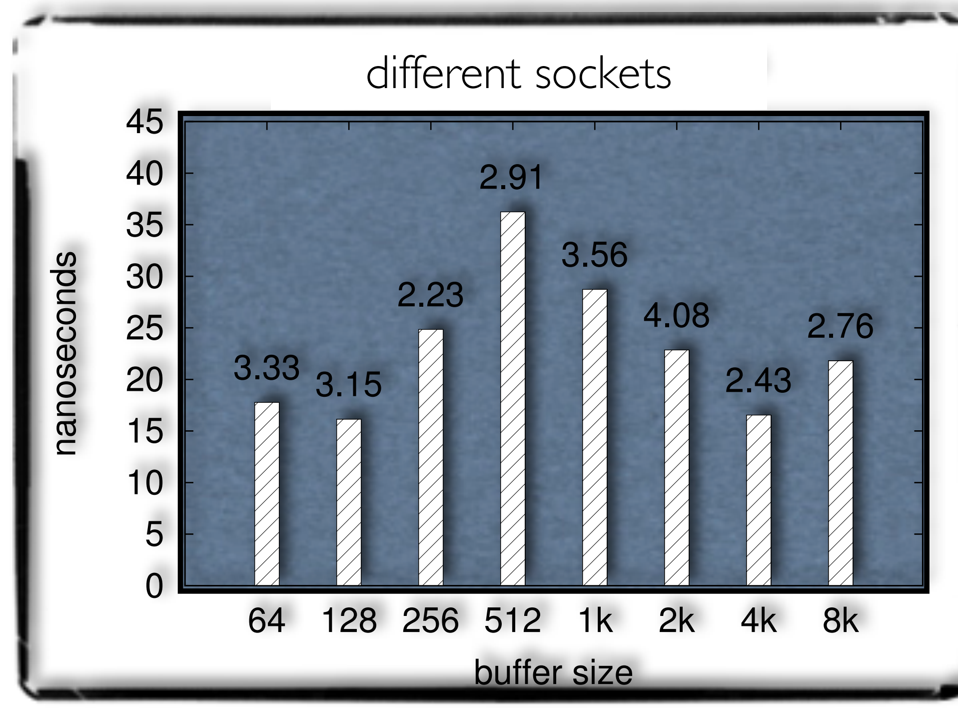
Layer 1: Simple streaming networks

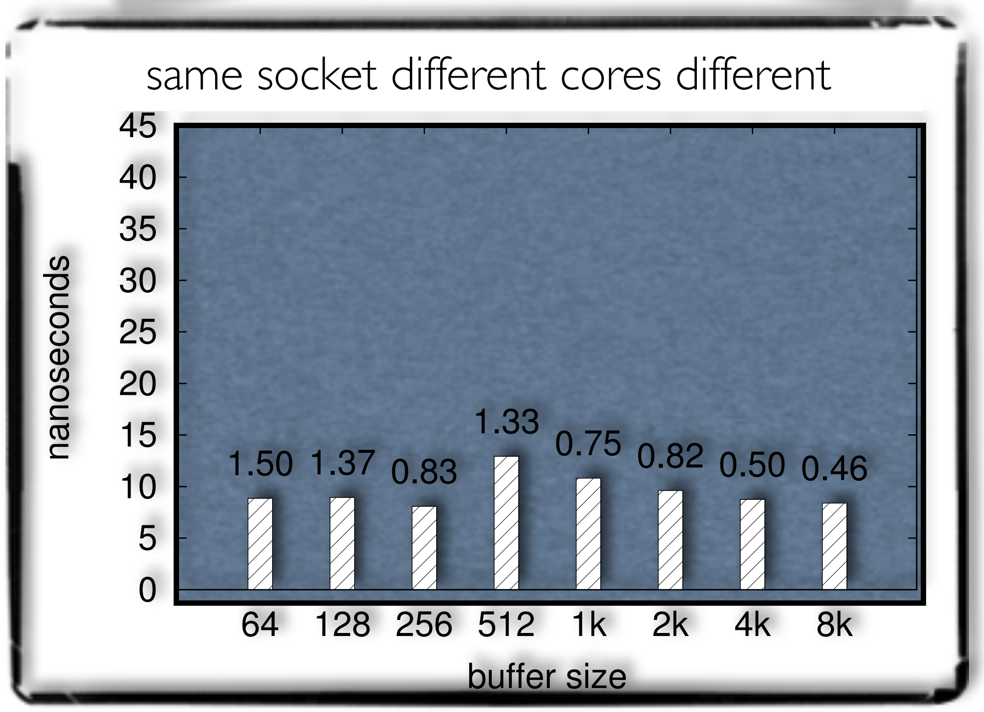
UNIVERSITÀ DEGLI STUDI
DI TORINO

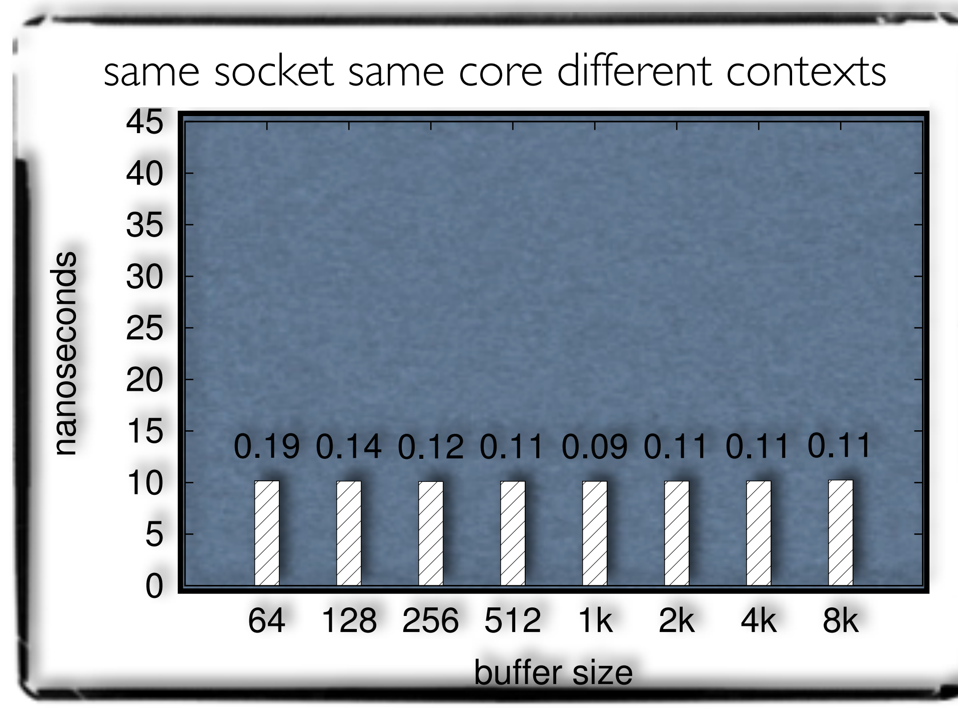
4 sockets x 8 core x 2 contexts

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

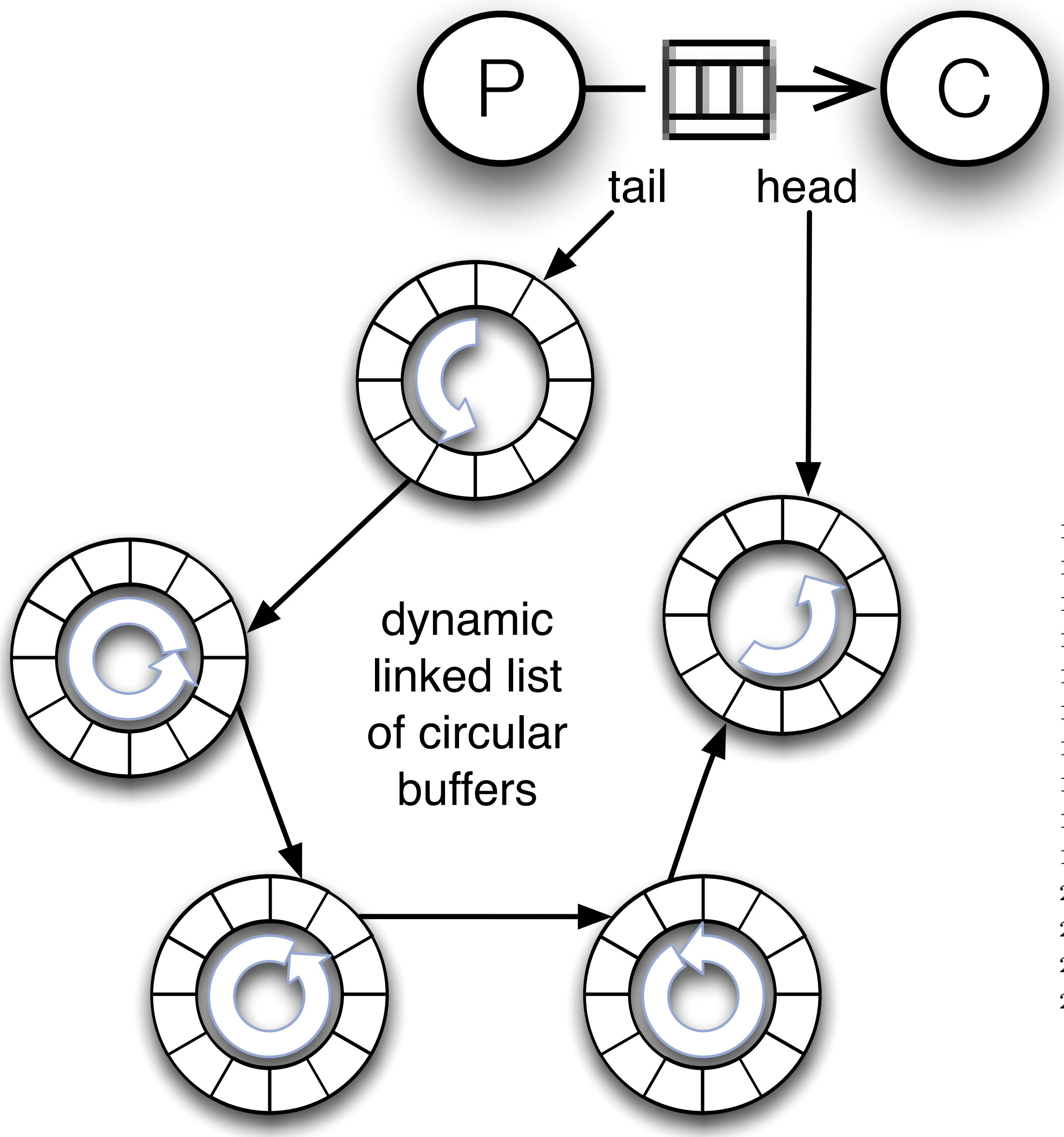
```
  MPI shmem impl
is ~190 ns at best
   (D.K. Panda)
```







Layer 1: Simple streaming networks




```
int size = N; //SPSC size
bool push(void* data) {
  if (buf_w->full()) {
    SPSC* t = pool.next_w();
    if (!t) return false;
    buf_w = t;
  }
  buf_w->push(data);
  return true;
}
bool pop(void** data) {
  if (buf_r->empty()) {
    if (buf_r == buf_w) return false;
    if (buf_r->empty()) {
      SPSC* tmp = pool.next_r();
      if (tmp) {
        pool.release(buf_r);
        buf_r = tmp;
      }
    }
  }
  return buf_r->pop(data);
}

struct Pool {
  dSPSC inuse;
  SPSC cache;

  SPSC* next_w() {
    SPSC* buf;
    if (!cache.pop(&buf))
      buf = allocateSPSC(size);
    inuse.push(buf);
    return buf;
  }
  SPSC* next_r() {
    SPSC* buf;
    return (inuse.pop(&buf)? buf : NULL);
  }
  void release(SPSC* buf) {
    buf->reset(); // reset pread and pwrite
    if (!cache.push(buf))
      deallocateSPSC(buf);
  }
}
```

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

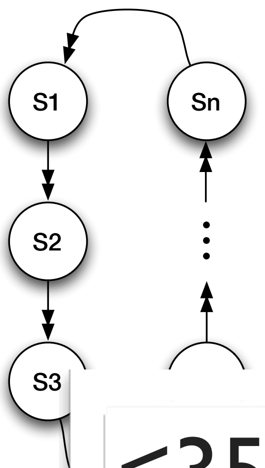
M. Aldinucci, S. Campa, M. Danelutto, M. Torquati. An Efficient Synchronisation Mechanism for Multi-Core Systems. EuroPar 2012.

Wed 29 Aug - B3 multicore 14.30-16.00

Layer 1: Simple streaming networks

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

UNIVERSITÀ DEGLI STUDI DI TORINO

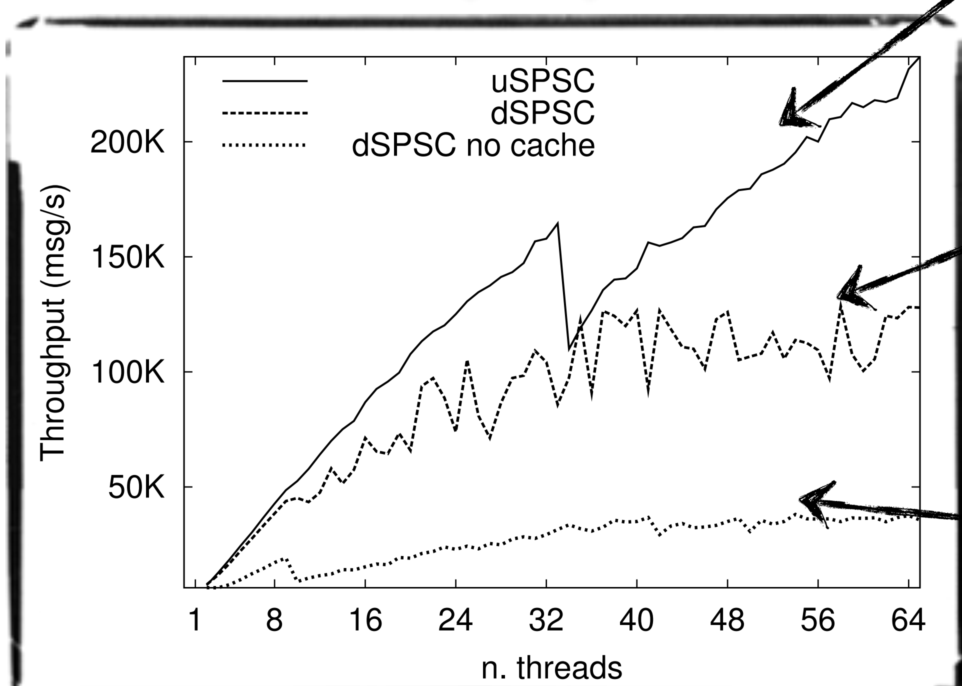


4 sockets x 8 core x 2 contexts

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

<35 ns irrespectively of the mapping

Speedup



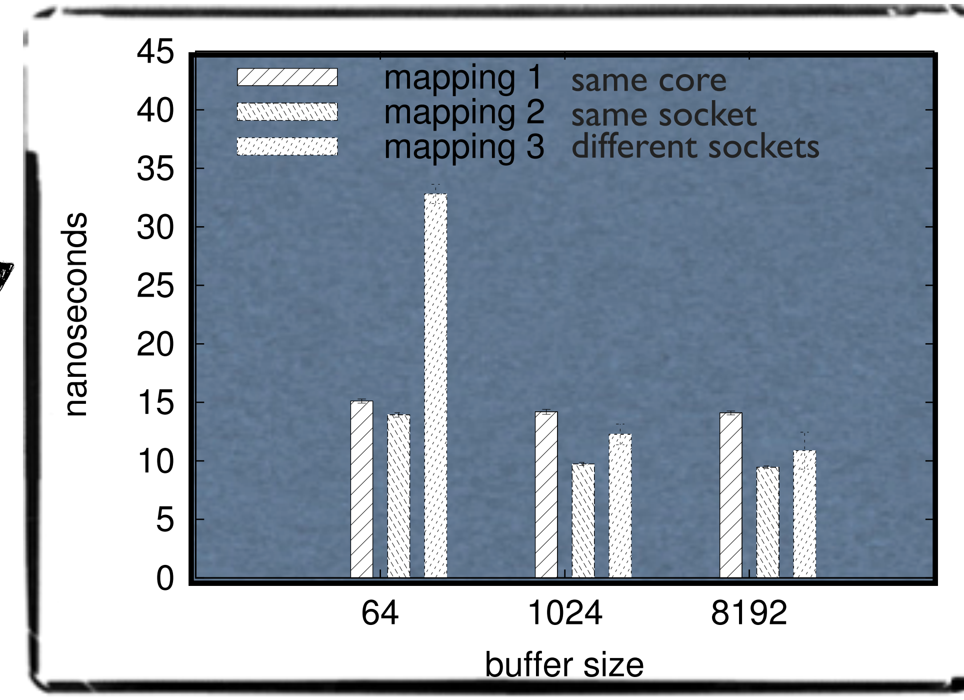
Linked list + circular buffer
FastFlow queue
(our result)

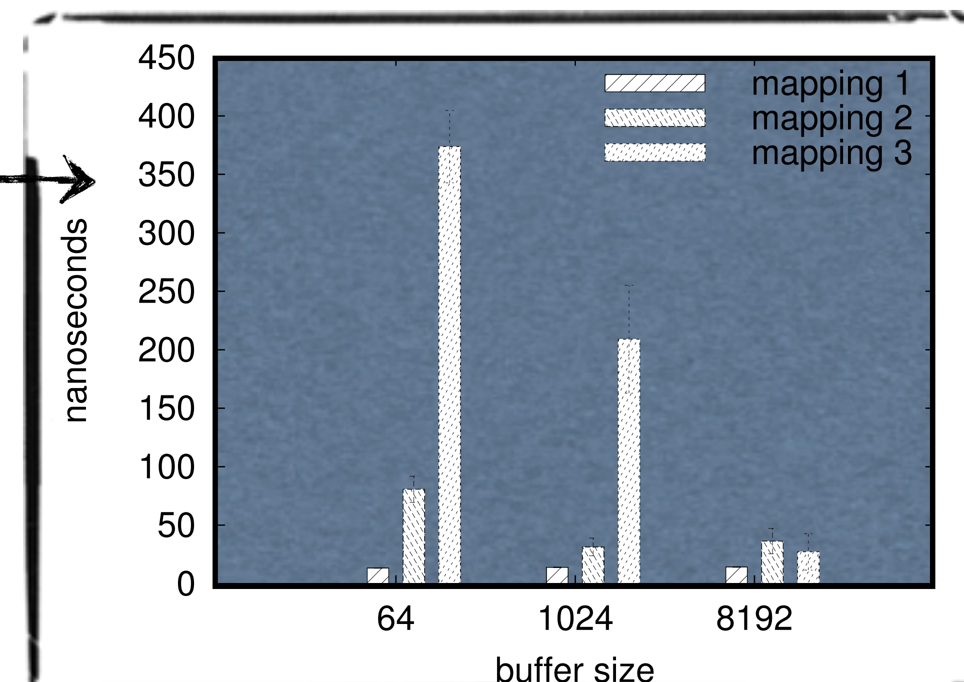
12x faster

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

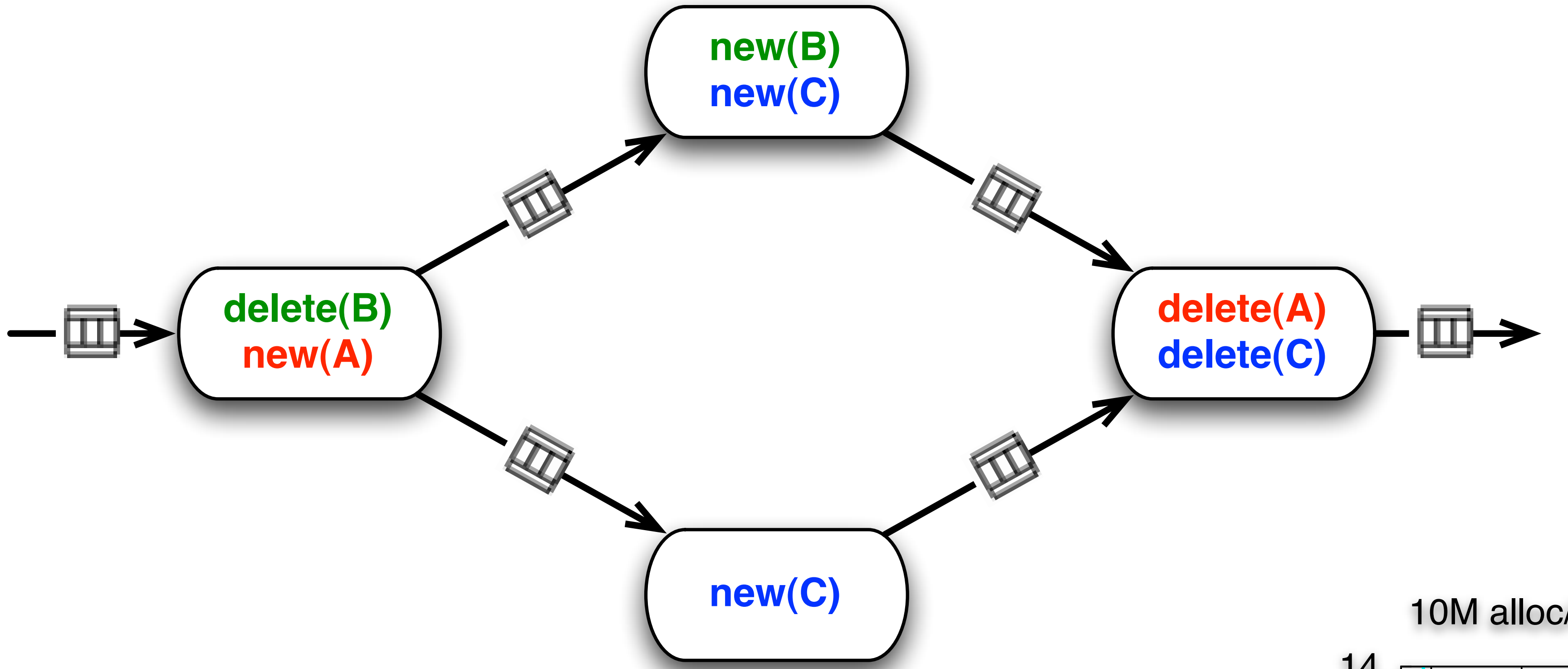
20x faster

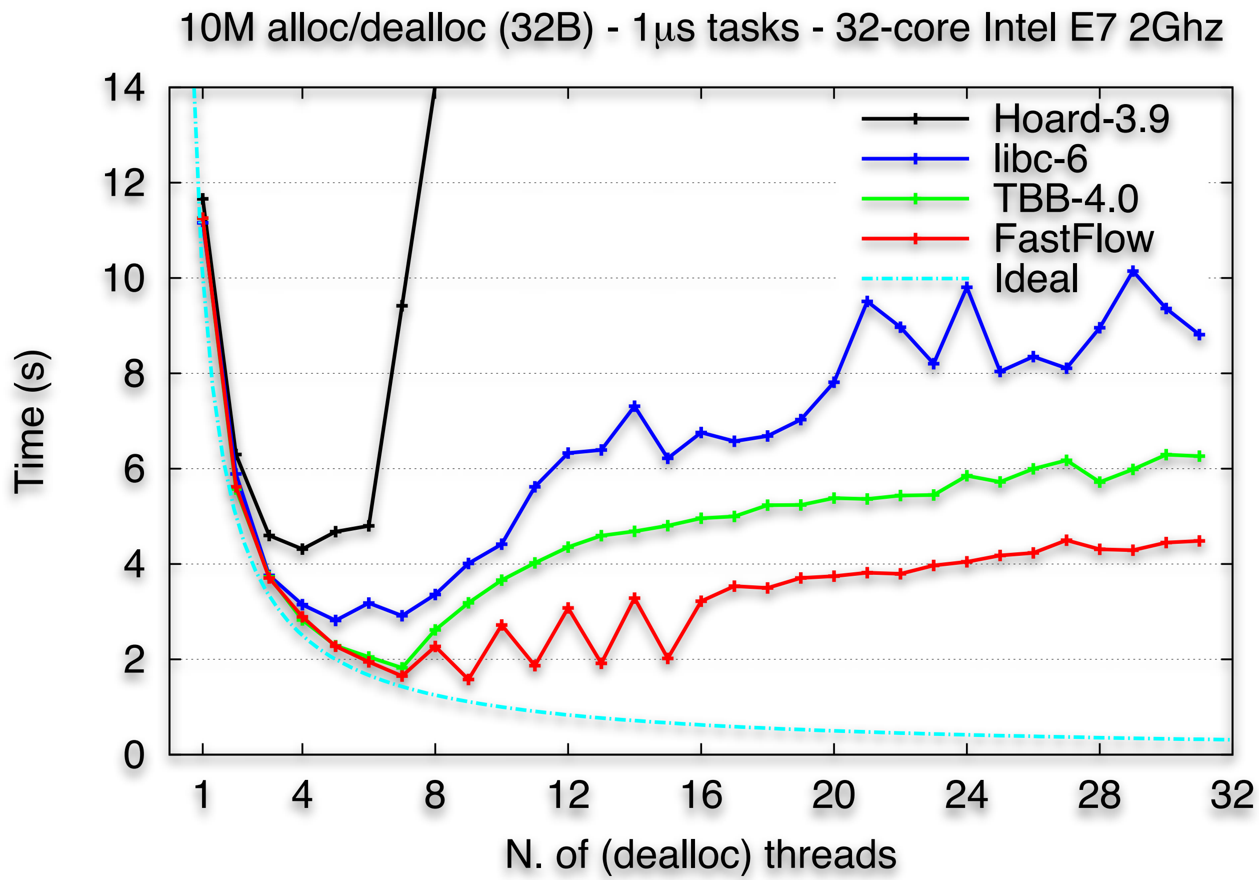
Linked list w/ dyn alloc
Michael-Scott queue
well-known ~ 400 citations





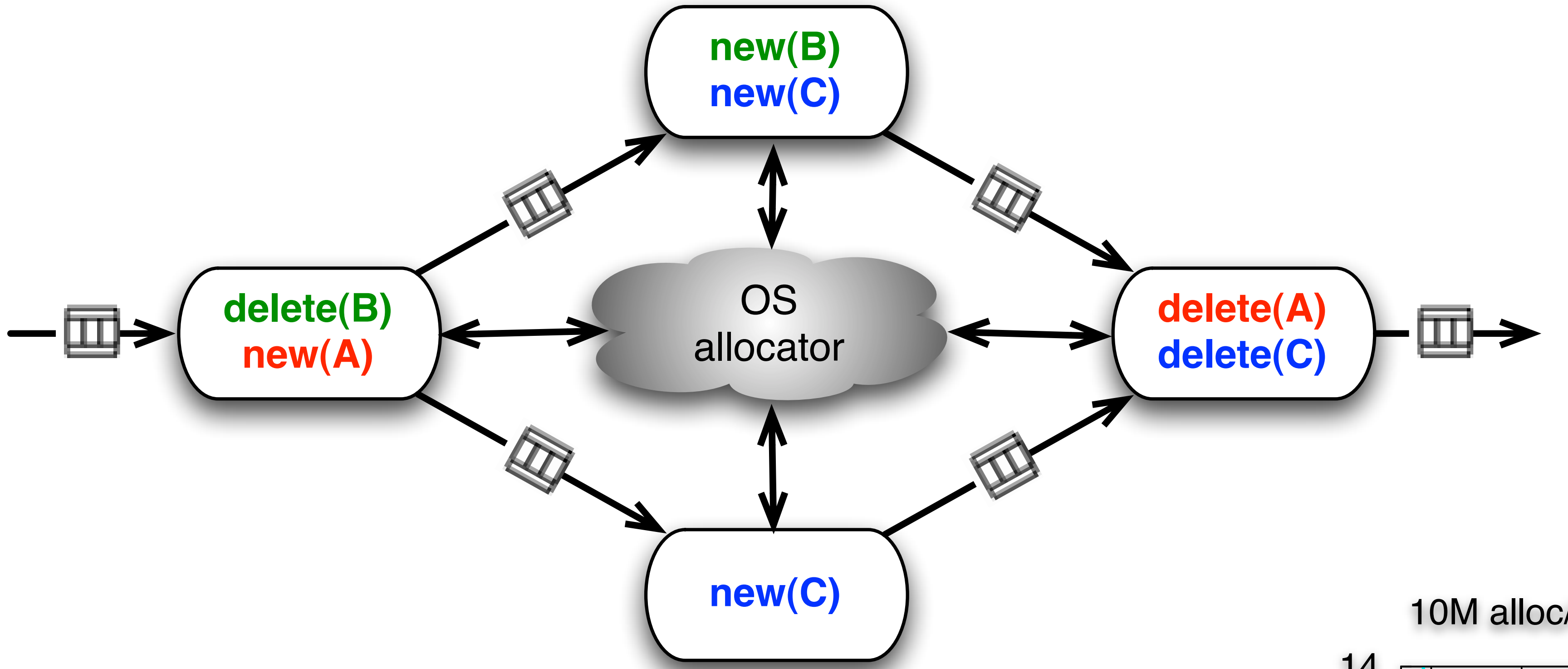
Unbound queues are useful

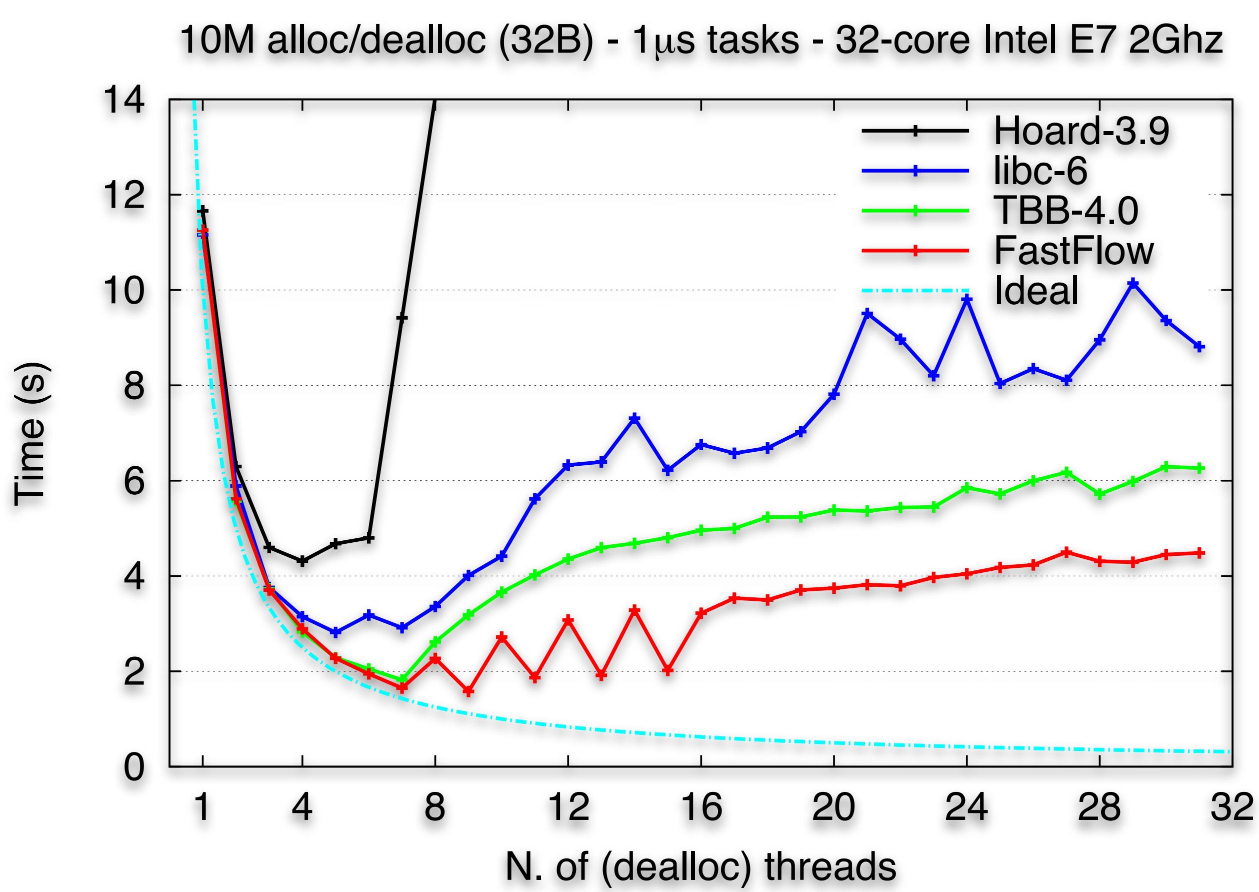




Unbound queues are useful

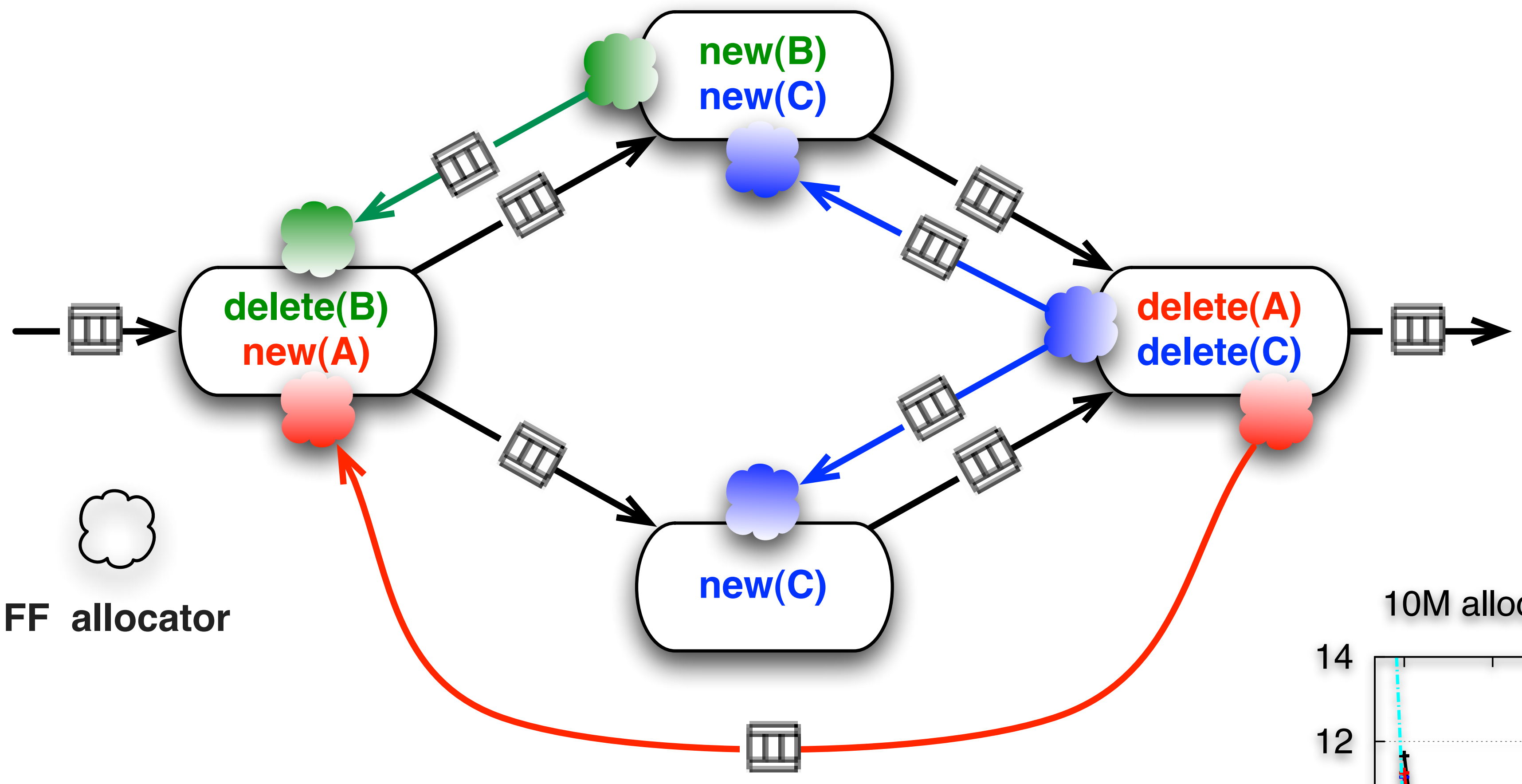
UNIVERSITÀ DEGLI STUDI
DI TORINO



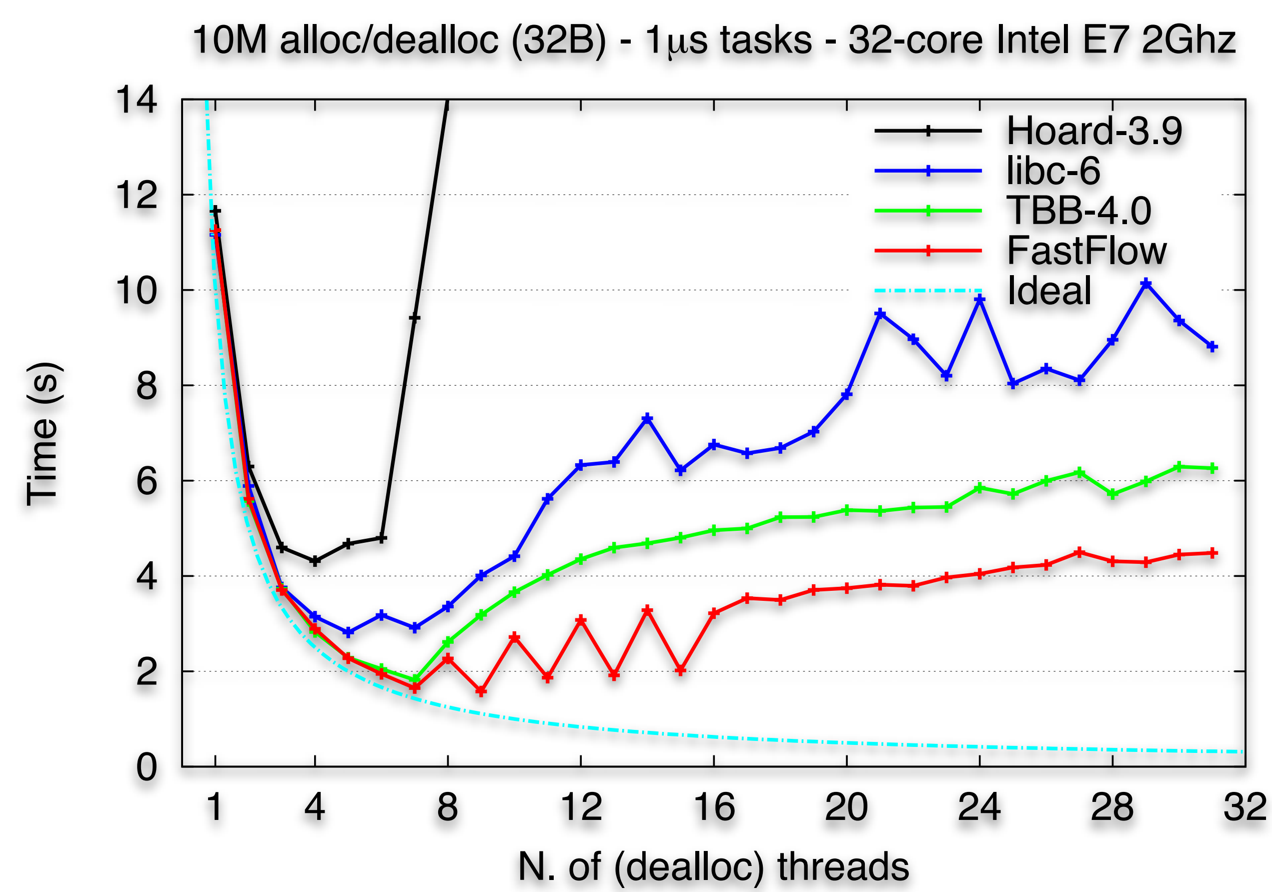


Unbound queues are useful

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

UNIVERSITÀ DEGLI STUDI
DI TORINO

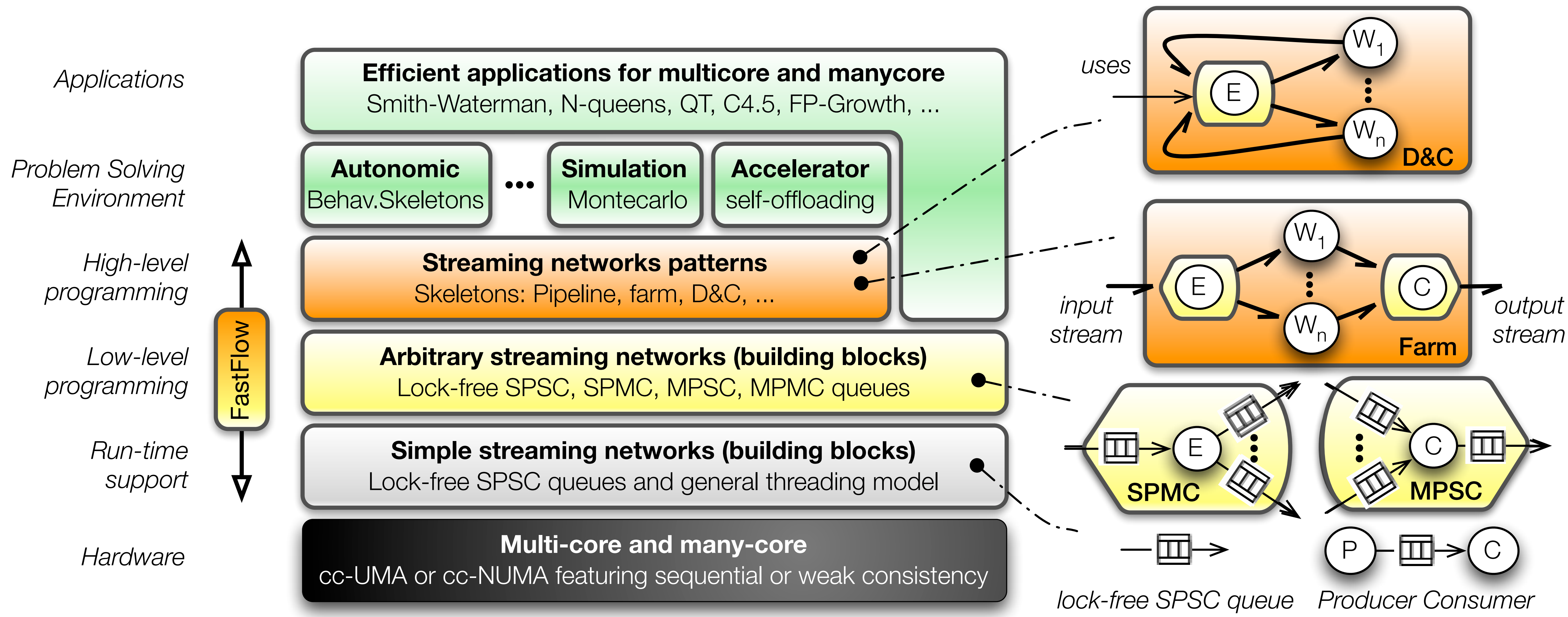
FastFlow

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

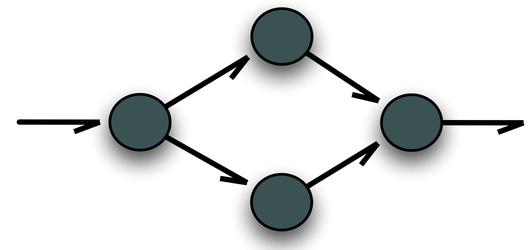
UNIVERSITÀ DEGLI STUDI DI TORINO

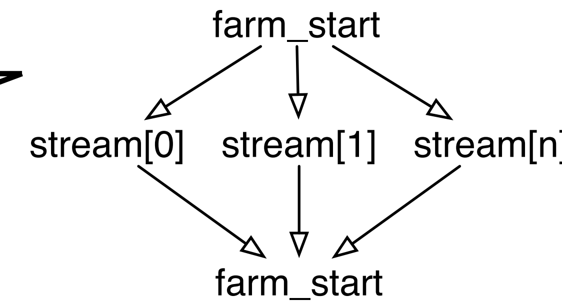
- Abstract parallelism exploitation pattern by parametric code
  - E.g. higher order function, code factories, C++ templates, ...
  - Can composed and nested as programming language constructs + offloading
  - Stream and Data Parallel
- Platform independent
  - Implementations on different multi/many-cores
  - 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

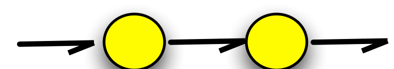
UNIVERSITÀ DEGLI STUDI DI TORINO

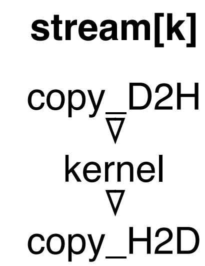
- farm
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA streams - automatic exploitation of asynch comm



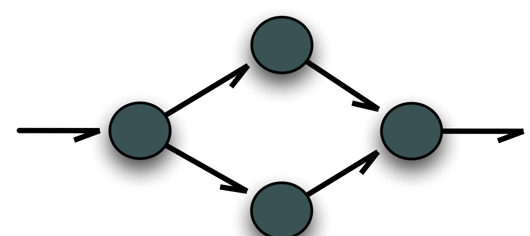


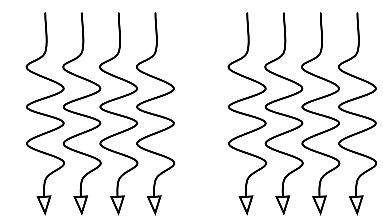
- pipeline
  - on CPU - pipeline
  - on GPU - sequence of kernel calls or global mem synch



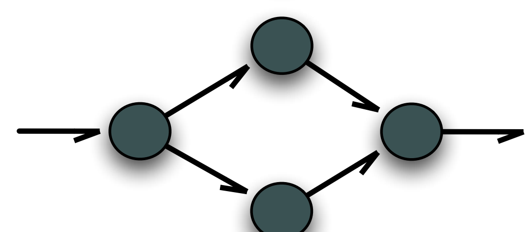


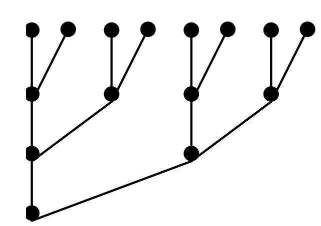
- map
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT - parallelism exploitation



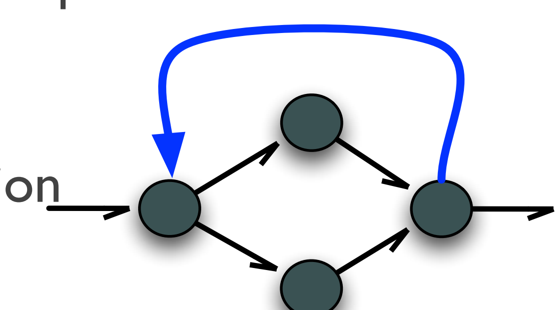


- reduce
  - on CPU - master-worker - parallelism exploitation
  - on GPU - CUDA SIMT (reduction tree) - parallelism exploitation





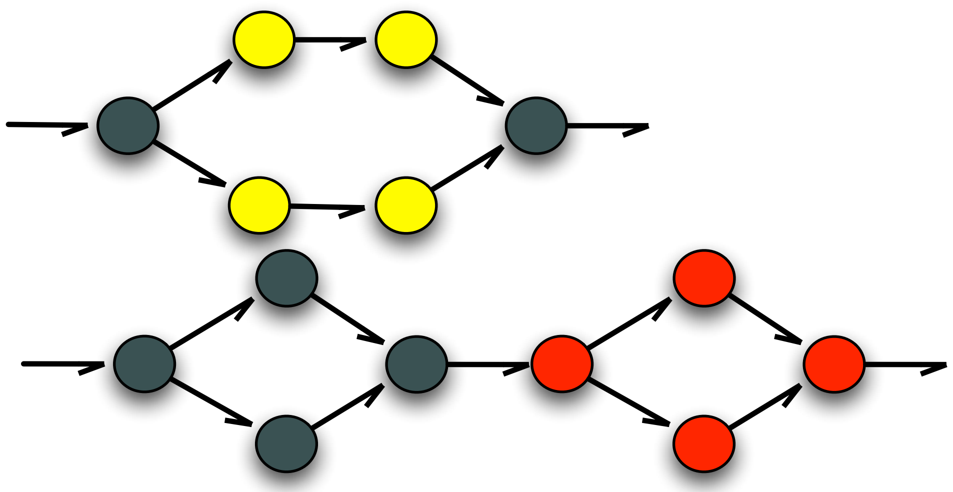
- D&C
  - on CPU - master-worker with feedback - // exploitation
  - on GPU - working on it, maybe loop+farm

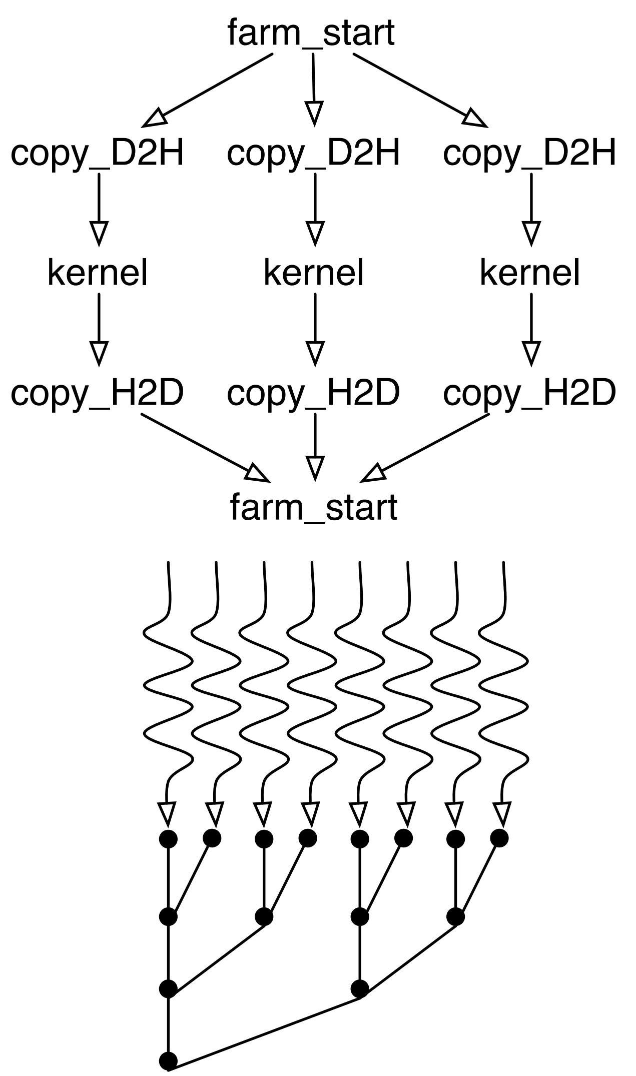


Composition

UNIVERSITÀ DEGLI STUDI DI TORINO

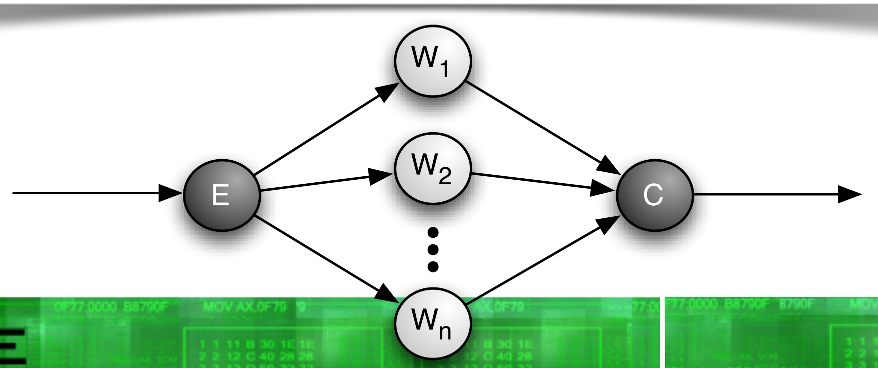
- Composition via C++ template meta-programming
  - CPU: Graph composition
  - GPU: CUDA streams
  - CPU+GPU: offloading
- farm{ pipe }
- pipe(farm, farm)
- pipe(map, reduce)
- ....



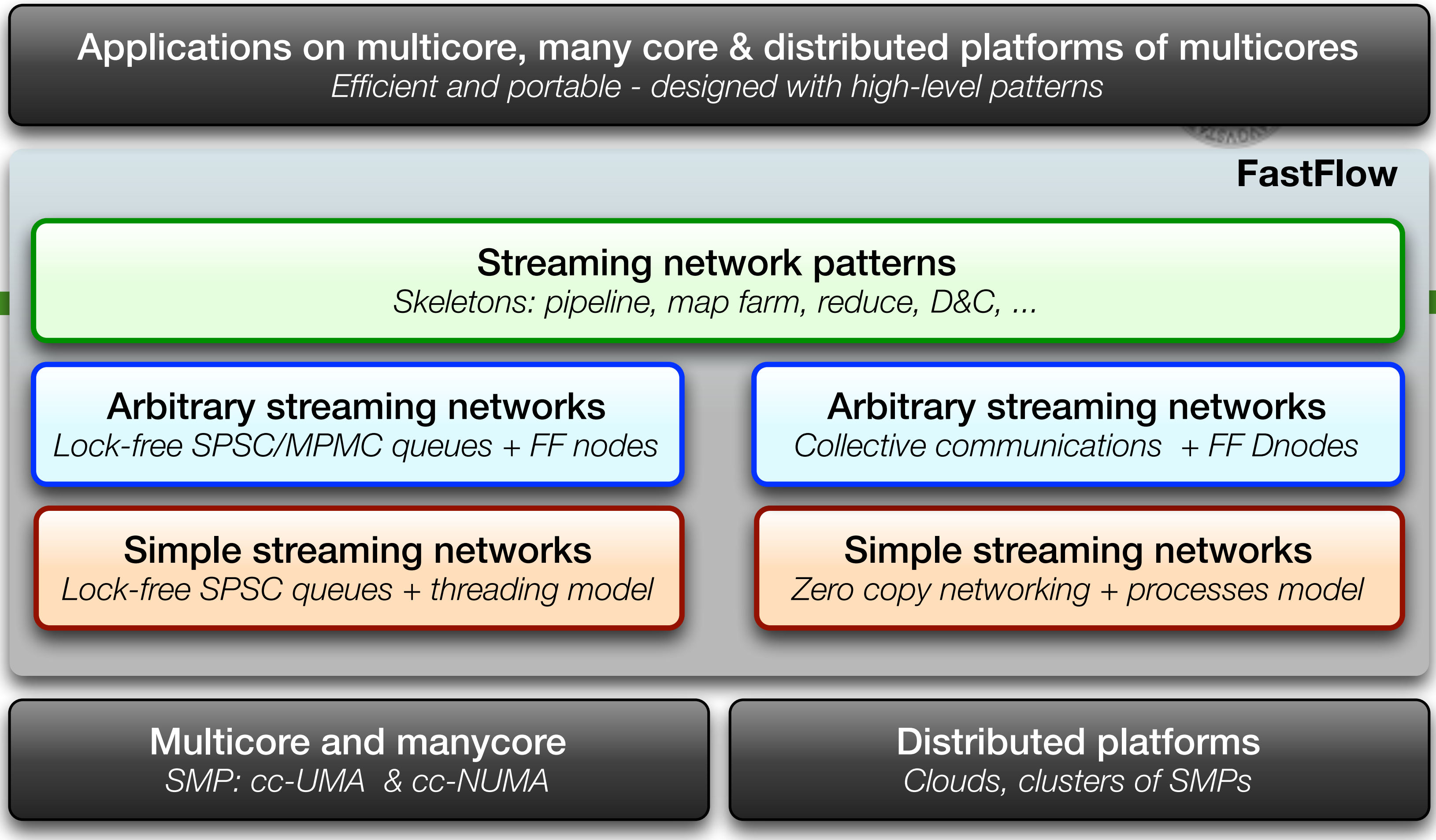


```
#include <vector>
#include <iostream>
#include <ff/farm.hpp>


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

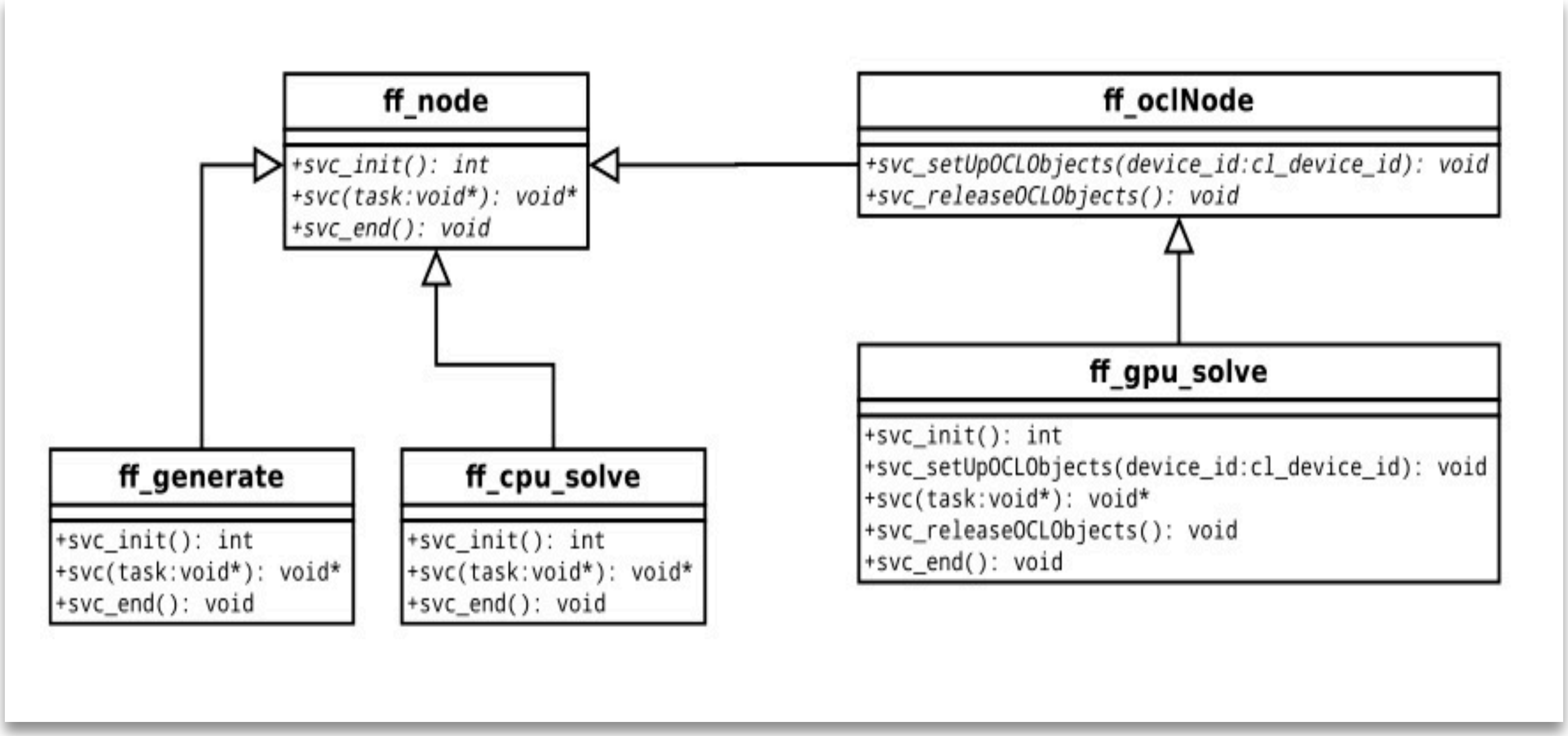


- Generic `ff_node` is subclassed to `ff_dnode`
- `ff_dnode` can support network channels
  - P2P or collective
  - used as frontier node of streaming graph
  - can be used to merge graphs across distributed platforms
- No changes to programming model
  - at least require to "add" stub `ff_dnode`
  -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

M. Aldinucci, S. Campa, M. Danelutto, M. Torquati, P. Kilpatrick. Targeting distributed systems in FastFlow. CGW-Europar 2012

+ OpenCL (working on)

UNIVERSITÀ DEGLI STUDI DI TORINO



Summary

- 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

2012: Cholesky fact vs PLASMA libraries
IEEE PDP 2012

UNIVERSITÀ DEGLI STUDI
DI TORINO

Targeting multi cores by structured programming and data flow

M. Aldinucci°, L. Anardu*, M. Danelutto*, P. Kilpatrick†, M. Torquati*

Dept. Computer Science,
Dept. Computer Science, U
Dept. Computer Science, Queen

Abstract

Data flow techniques have been around sin
were used in compilers for sequential language
duction they were also considered as a possible
ing, although the impact here was limited. R
has been identified as a candidate for efficient
programming models on multi-core architectur
the burden of determining data flow "macro"
programmer, while the compiler/run time sy
ficient scheduling of these instructions. We d
programming approach supporting automatic

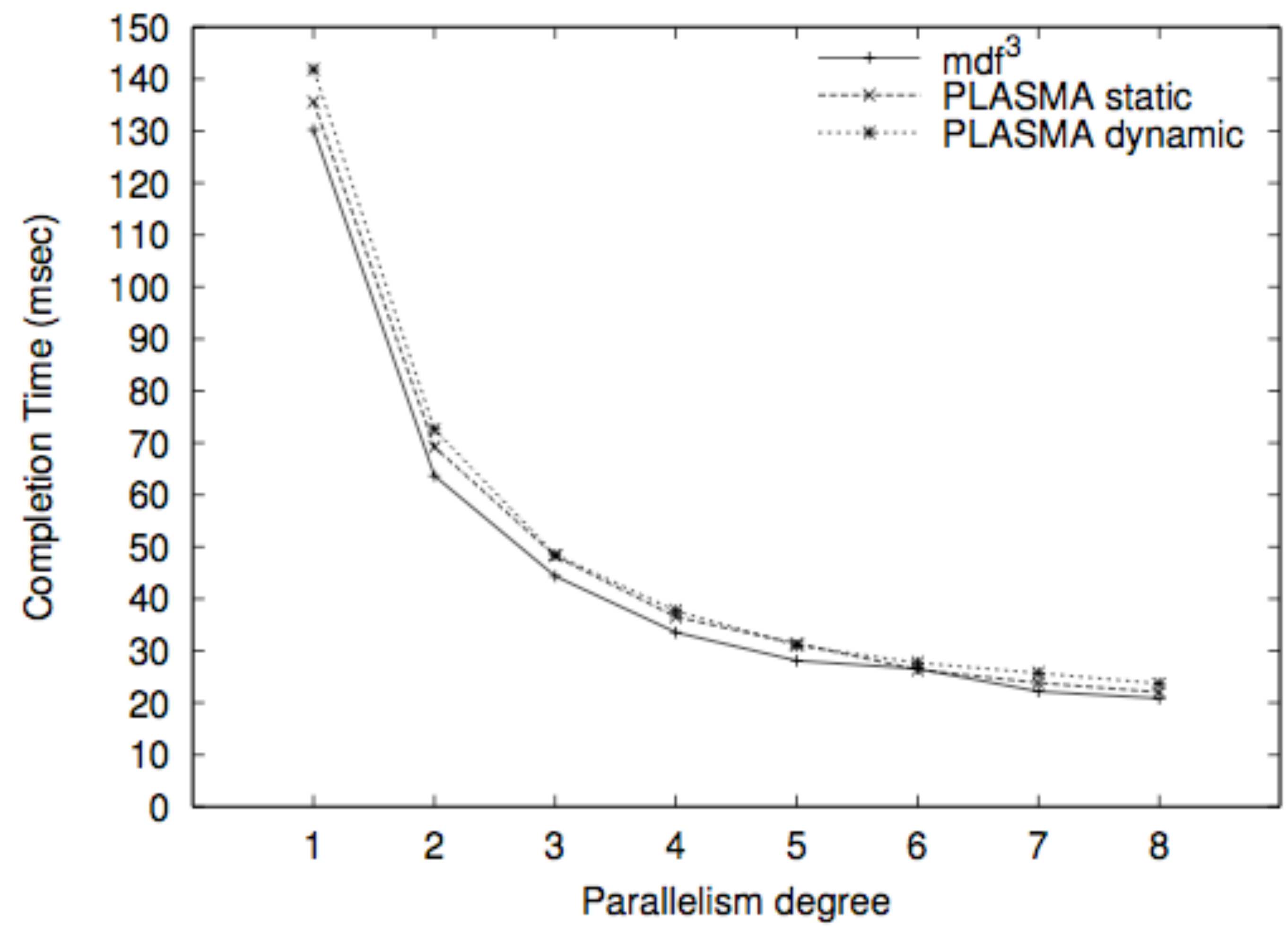


Figure 10: *mdf*^{3} vs PLASMA library. Cholesky factorization for a single 1024x1024 complex matrix (Intel Nehalem).

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 characterise elapsed time lower bounds for the forms of parallelisations adopted, and achieve close to optimal performances. Our implementation is based on the FastFlow para...

Index Terms—Parallel clas...

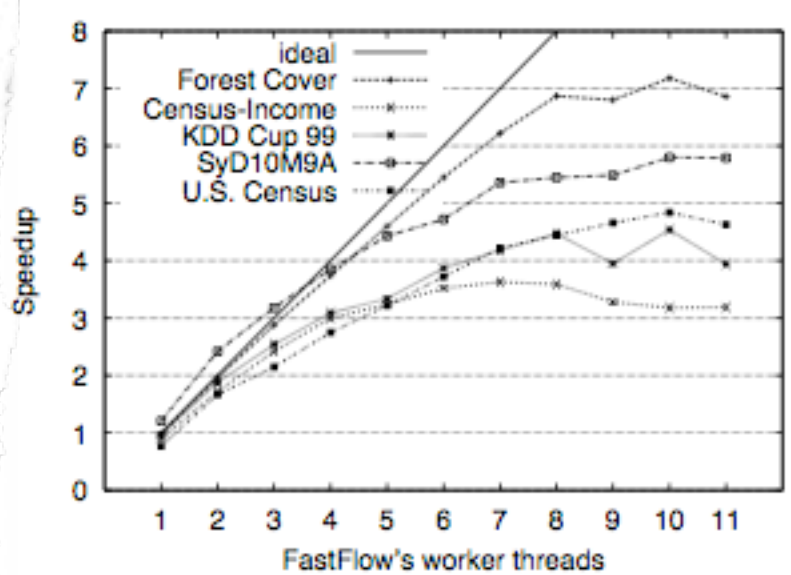


Fig. 11: NAP strategy speedup.

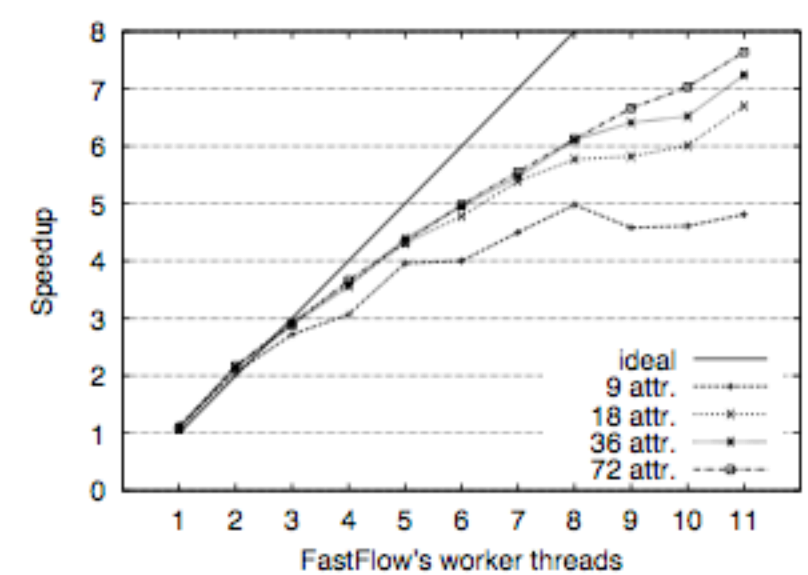
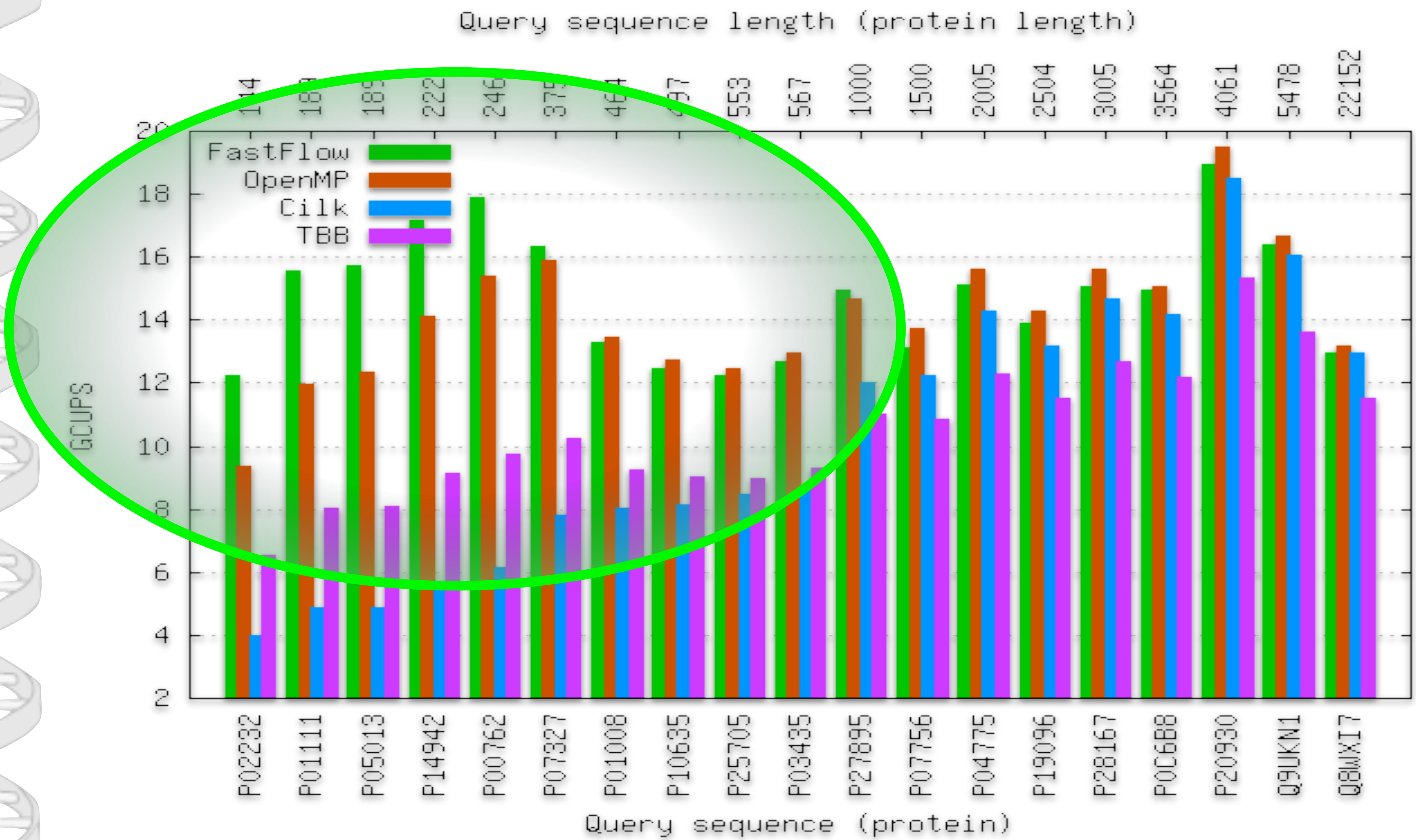


Fig. 12: Speedup of the NAP strategy vs no. of attributes (for 1M sample subset of SyD10M9A).

Smith Waterman vs TBB vs OpenMP vs Cilk
Parallel Computing 2010

UNIVERSITÀ DEGLI STUDI
DI TORINO



Two-phase denoising
IEEE IPTA 2012

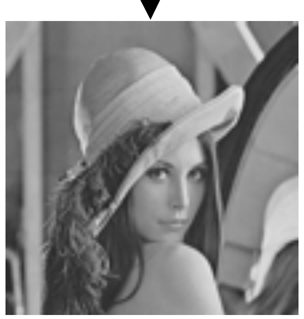
UNIVERSITÀ DEGLI STUDI
DI TORINO

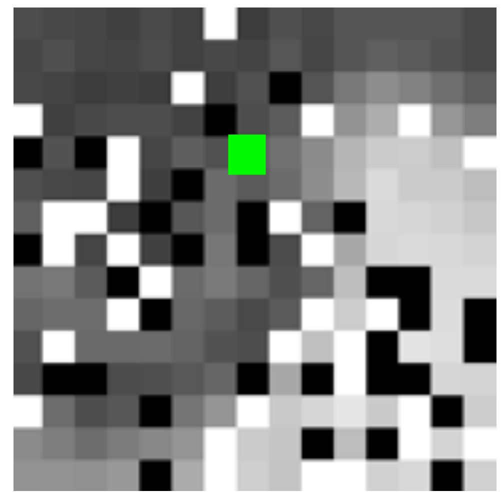
detect

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

denoise

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

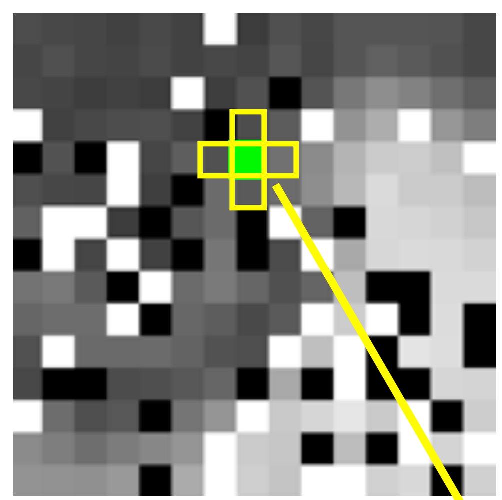




Adaptive median filter

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

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

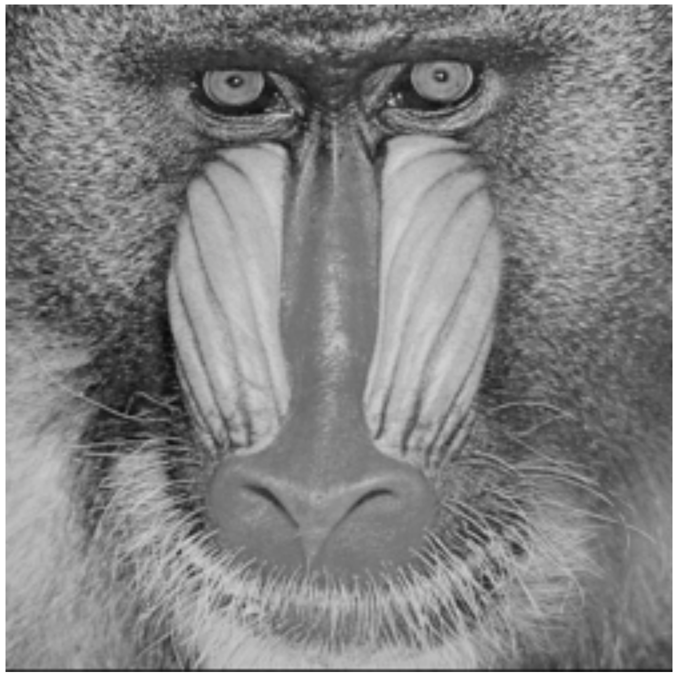
$$\varphi(t) = |t|^\alpha \text{ with } 1 < \alpha \le 2$$

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

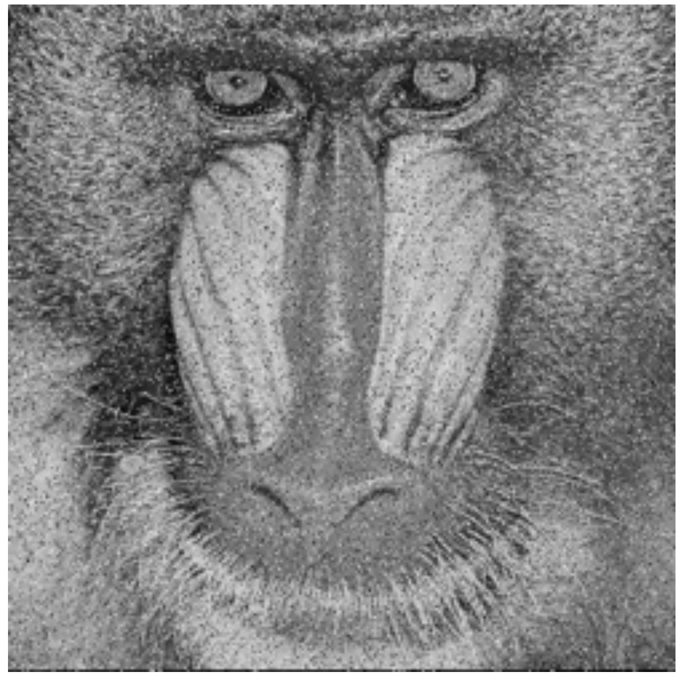
$$S_2 = \sum_{(m,n)\in V_{i,j}\cap N^c} \varphi(u_{i,j} - u_{m,n})$$

Two-phase denoising
IEEE IPTA 2012 (Istanbul, 15-18 Oct)

UNIVERSITÀ DEGLI STUDI
DI TORINO

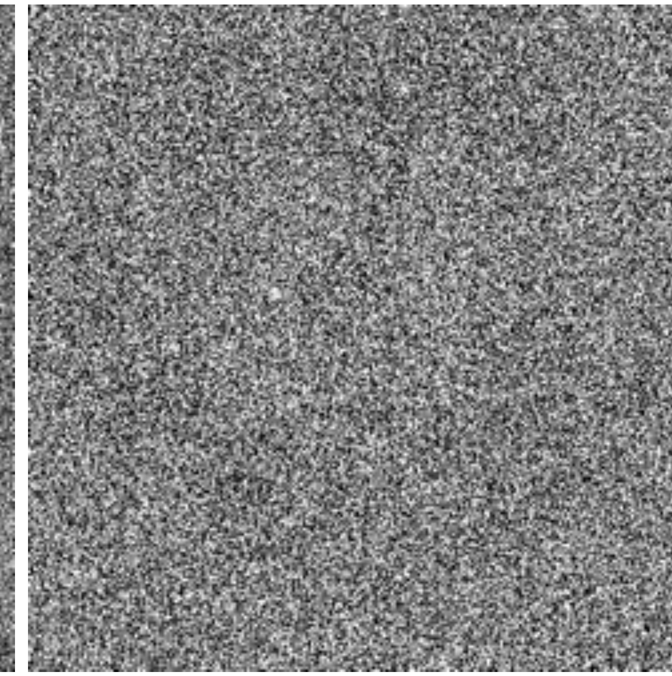


Original
Baboon standard
test image
1024x1024



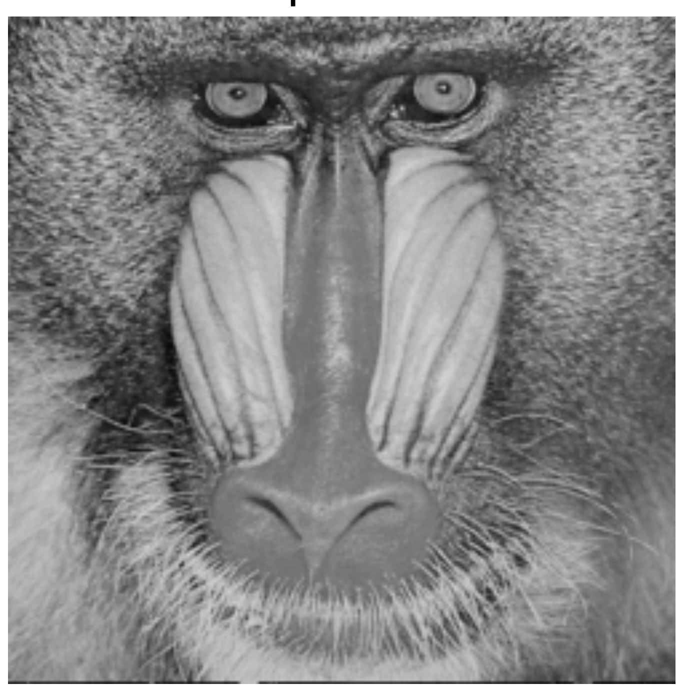
10% impulsive noise

50% impulsive noise

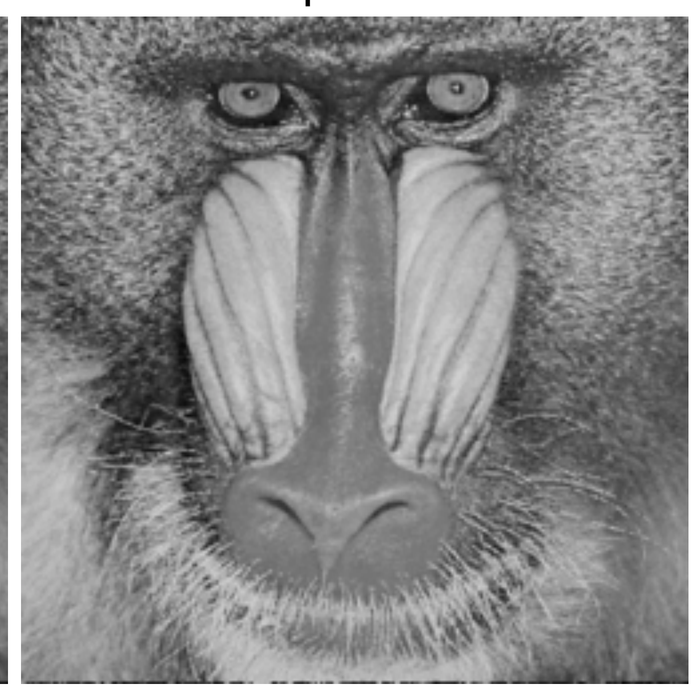


90% impulsive noise

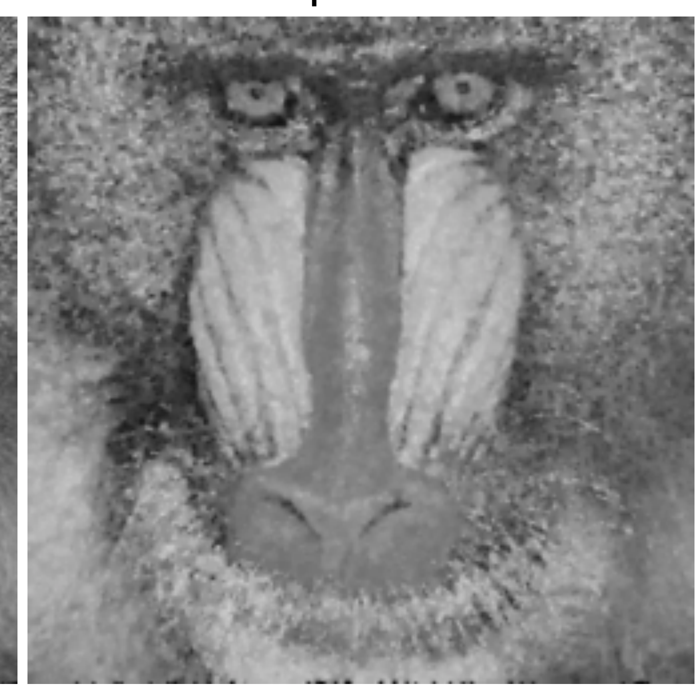
Restored



PNSR 43.29dB MAE 0.35



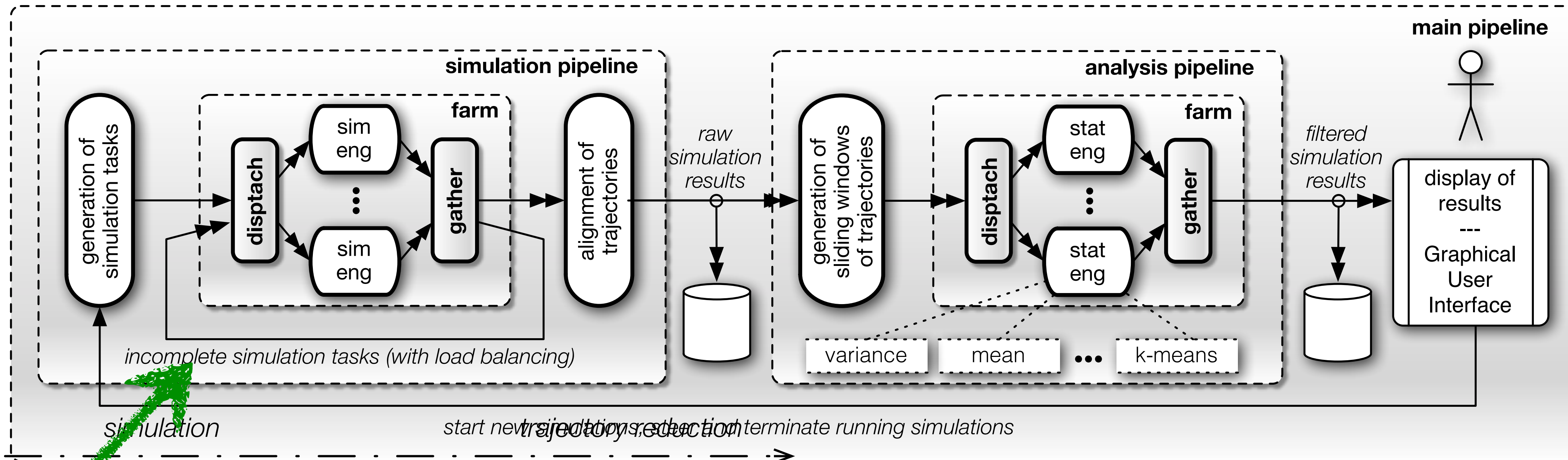
PNSR 32.75dB MAE 2.67

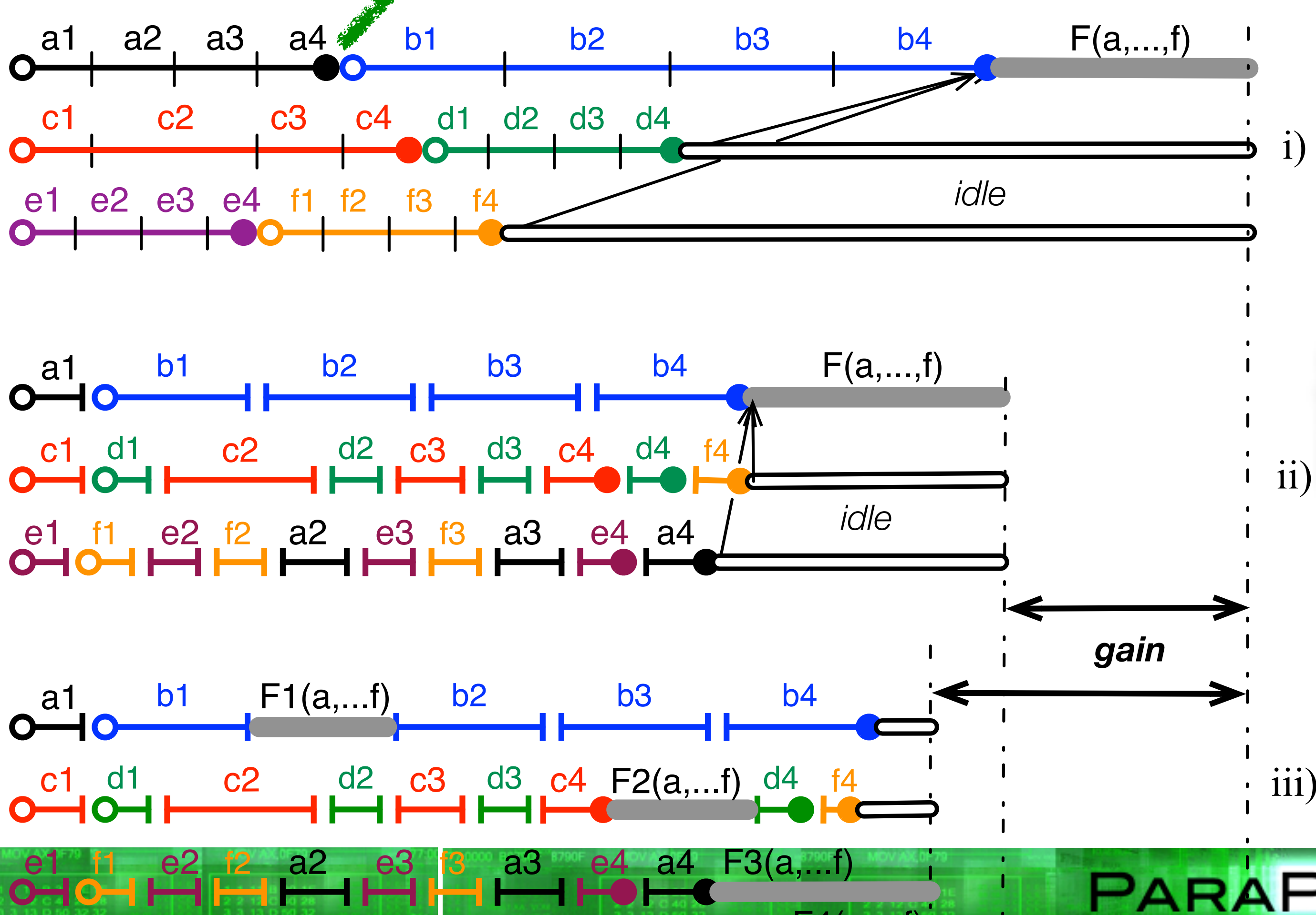


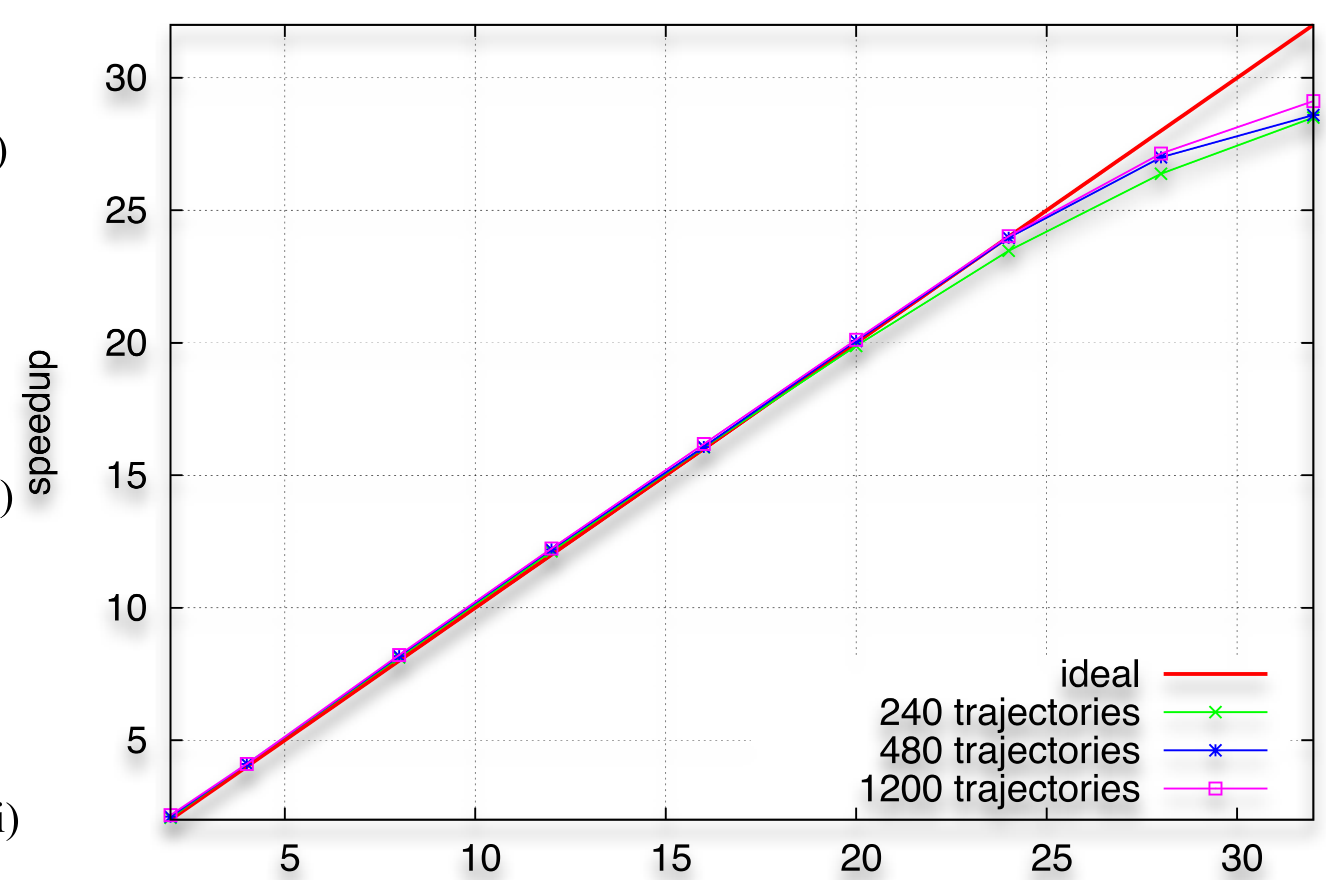
PNSR 23.4 MAE 11.21

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

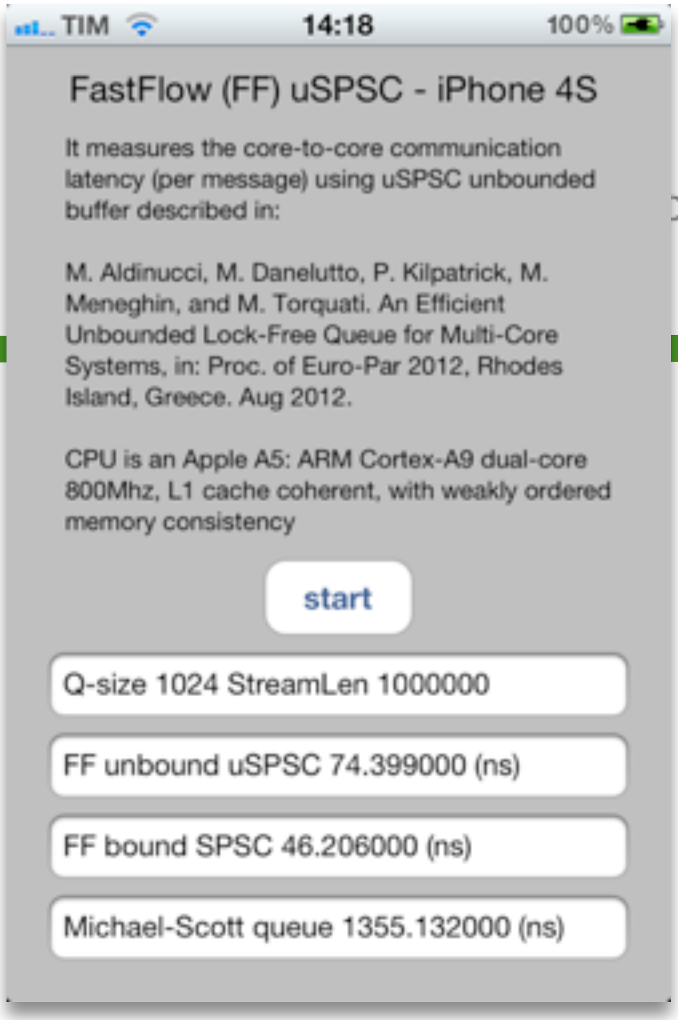
UNIVERSITÀ DEGLI STUDI DI TORINO







Conclusions



- FastFlow C++ pattern-based framework
  - A tiny, lightweight & open research framework
  - 3 years old - over 8K downloads - 40K 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
  - **1 postdoc position open: 2 years, about 1800 Euro/mount after taxes**
- 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

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

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