UNIVERSITÀ DEGLI STUDI DI TORINO

International Summer School in Parallel Patterns

June 10, 2014
Dublin, Ireland

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

Marco Aldinucci
Parallel programming models group
University of Torino
Italy

email: aldinuc@di.unito.it

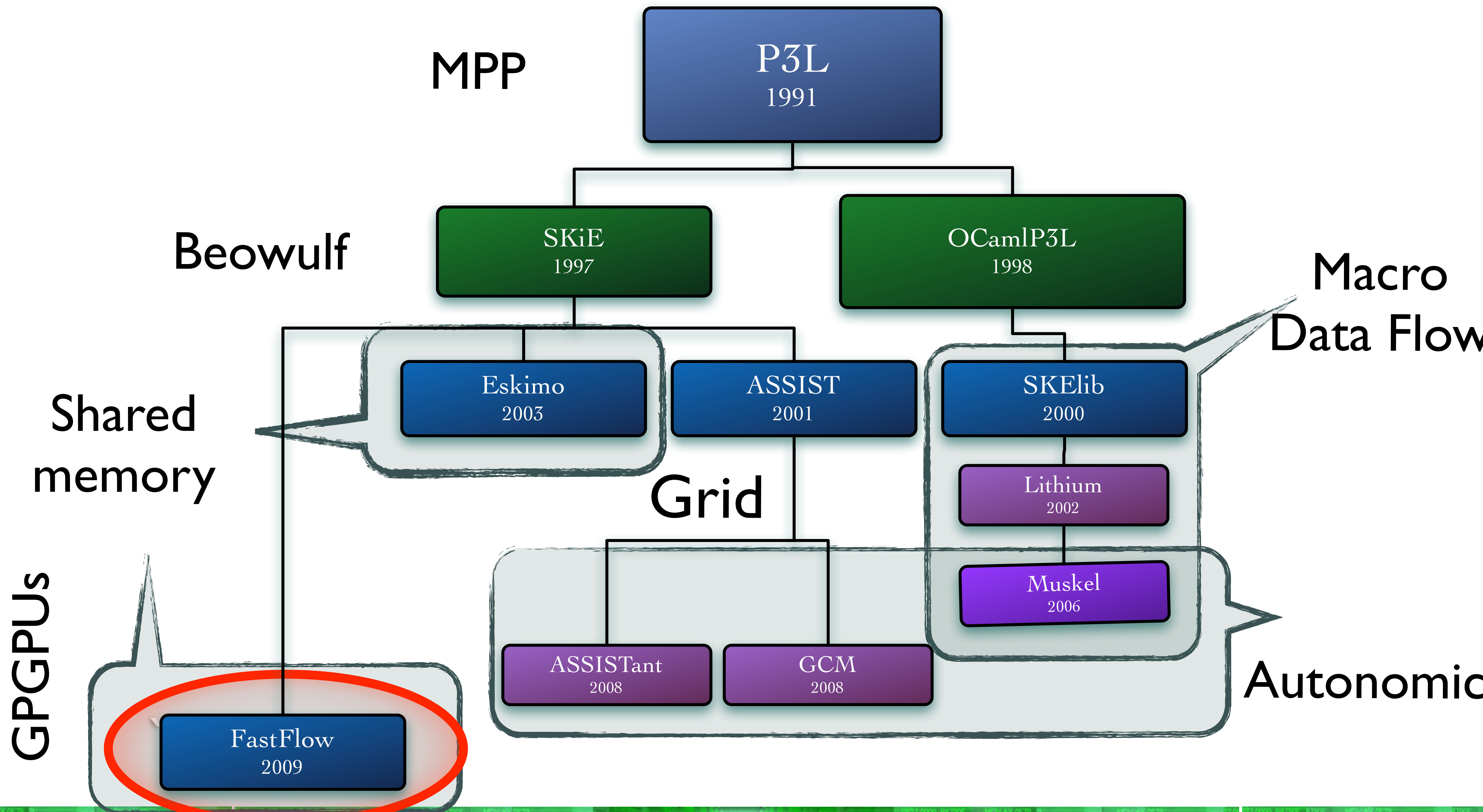
SEVENTH FRAMEWORK PROGRAMME

PARAPHRASE

Outline

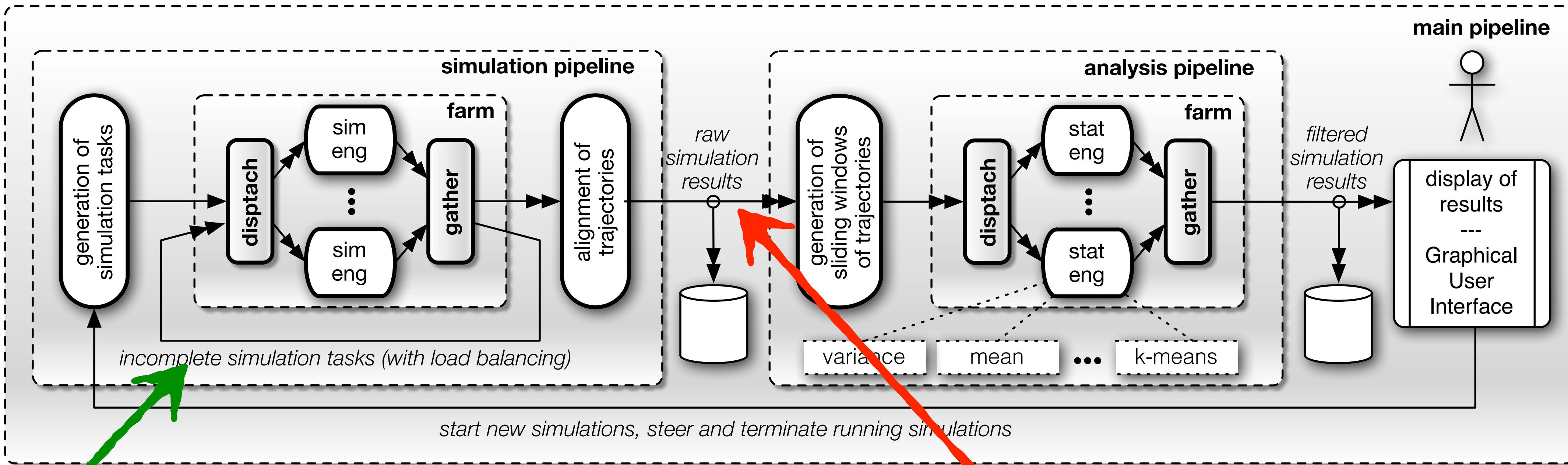
- Concurrency and multi-core, the theoretical background
  - a personal perspective
- FastFlow
  - A programming model (and a library) for multicore (& manycore)
- Code examples
- Discussion

Algorithmic skeletons and parallel patterns at UniPI and UniTO



Concurrency and multi-core theoretical background: a personal perspective

Use case: Parallel stochastic simulations for system biology



simulation engine

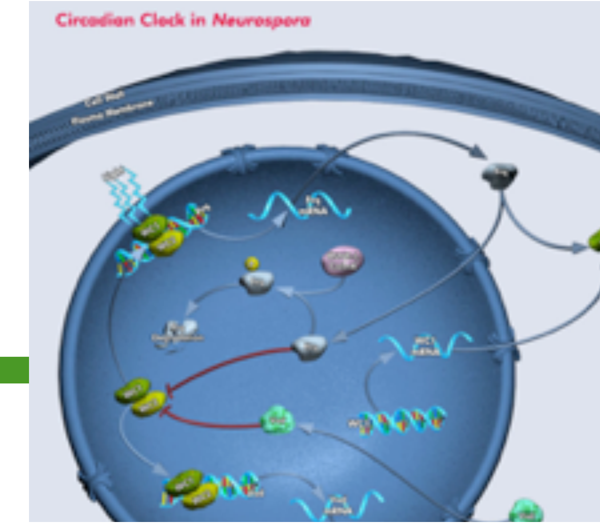
task size

Table 2 - Performance (Intel 32 core platform)

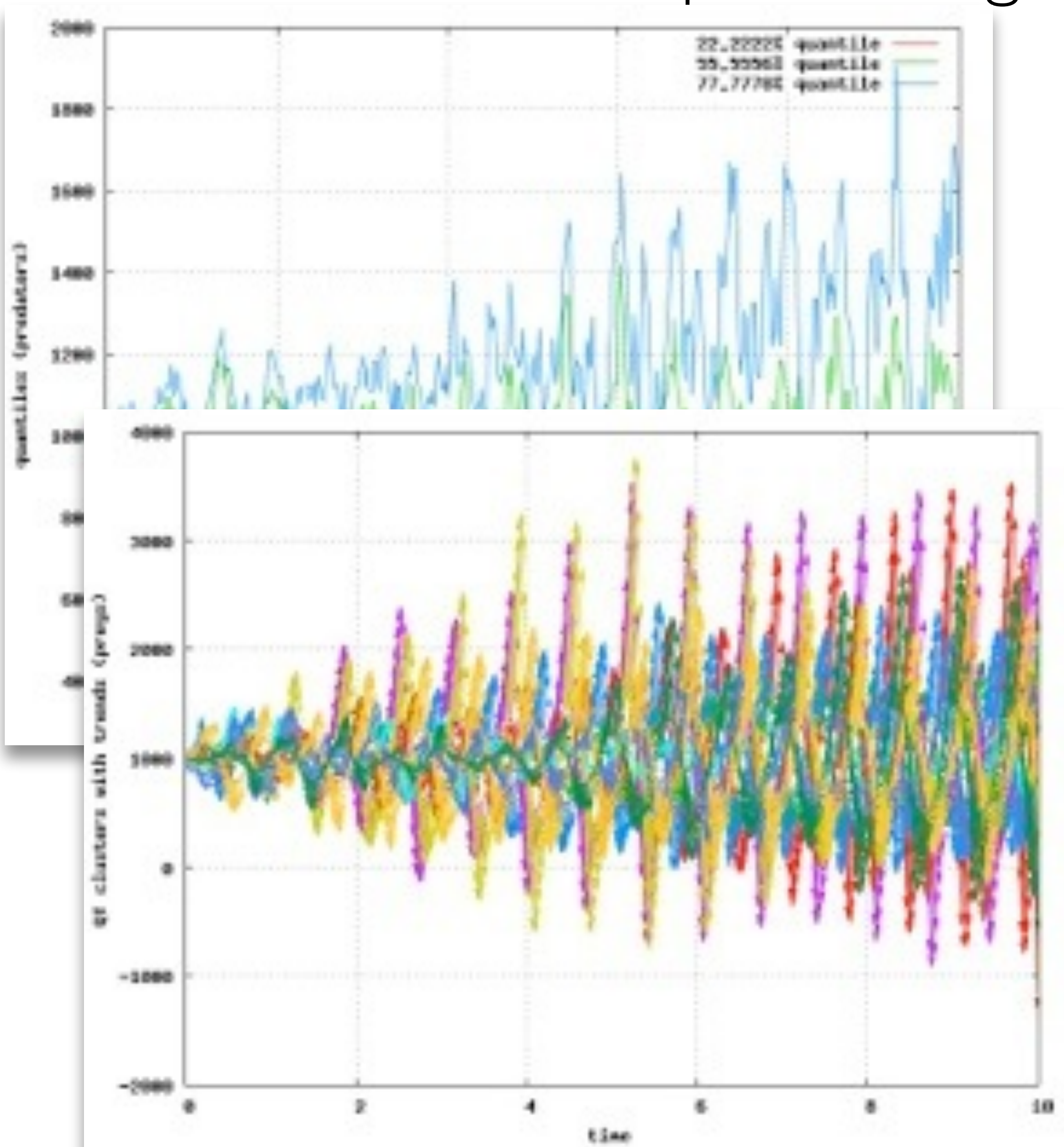
Model	Single trajectory information			Overall data (20 sim eng, 3 stat eng)		
	N. samples	Avg sim step	Sample time	Inter-arrival time	Throughput	Output size
Neurospora	10^4	7.80 μs	517.24 μs	25.86 μs	11.87 MB/s	36.62 MB
Neurospora	10^5	8.37 μs	55.51 μs	2.78 μs	11.98 MB/s	366.21 MB
Neurospora	10^6	75.63 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

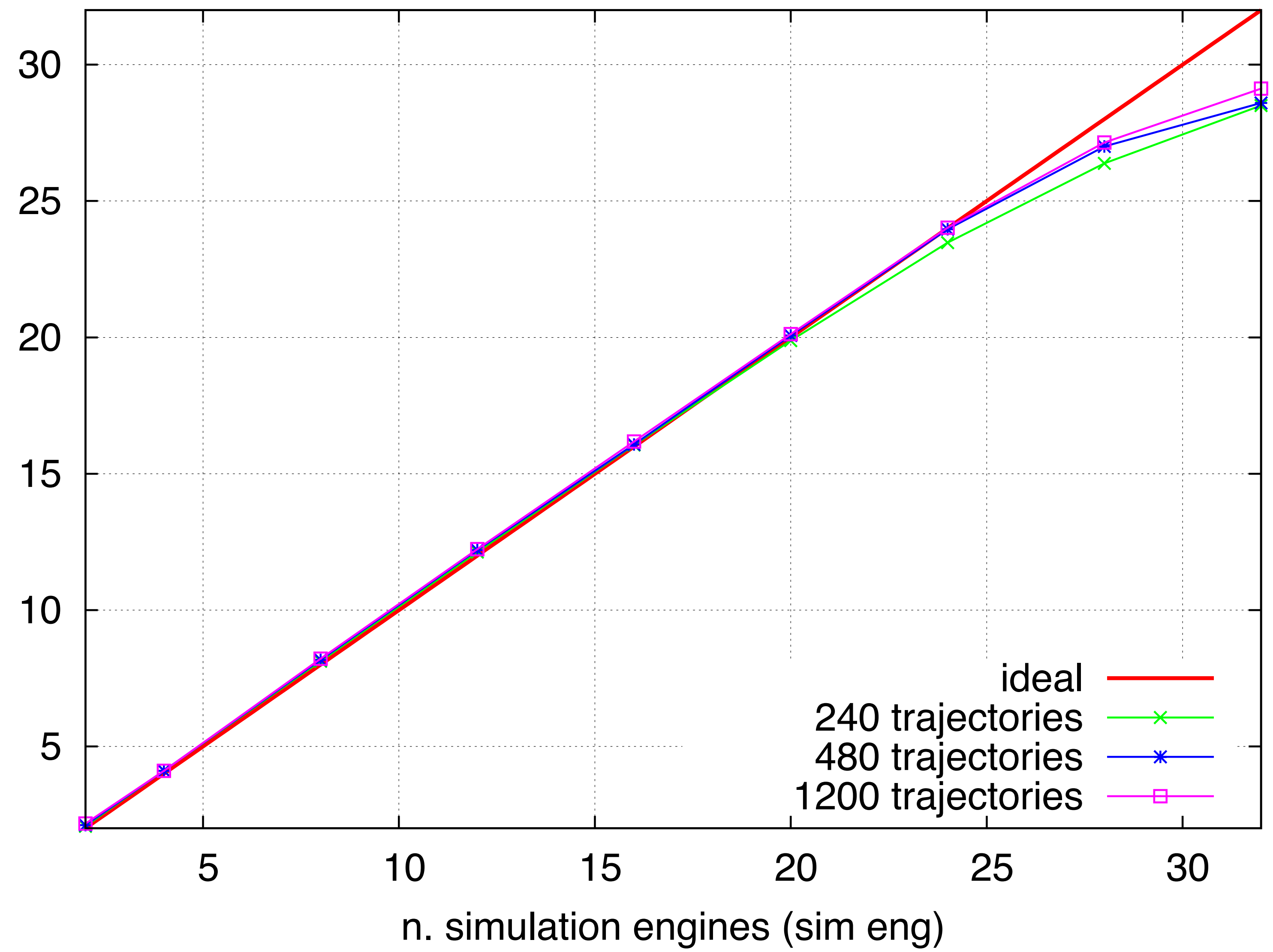
- M. Aldinucci et al. Parallel stochastic systems biology in the cloud. Briefings in Bioinformatics, 2013
- M. Aldinucci et al. On designing multicore-aware simulators for systems biology endowed with on-line statistics. BioMed Research International, 2014

Parallel stochastic simulations for system biology



Simulation of transcriptional regulation in Neurospora

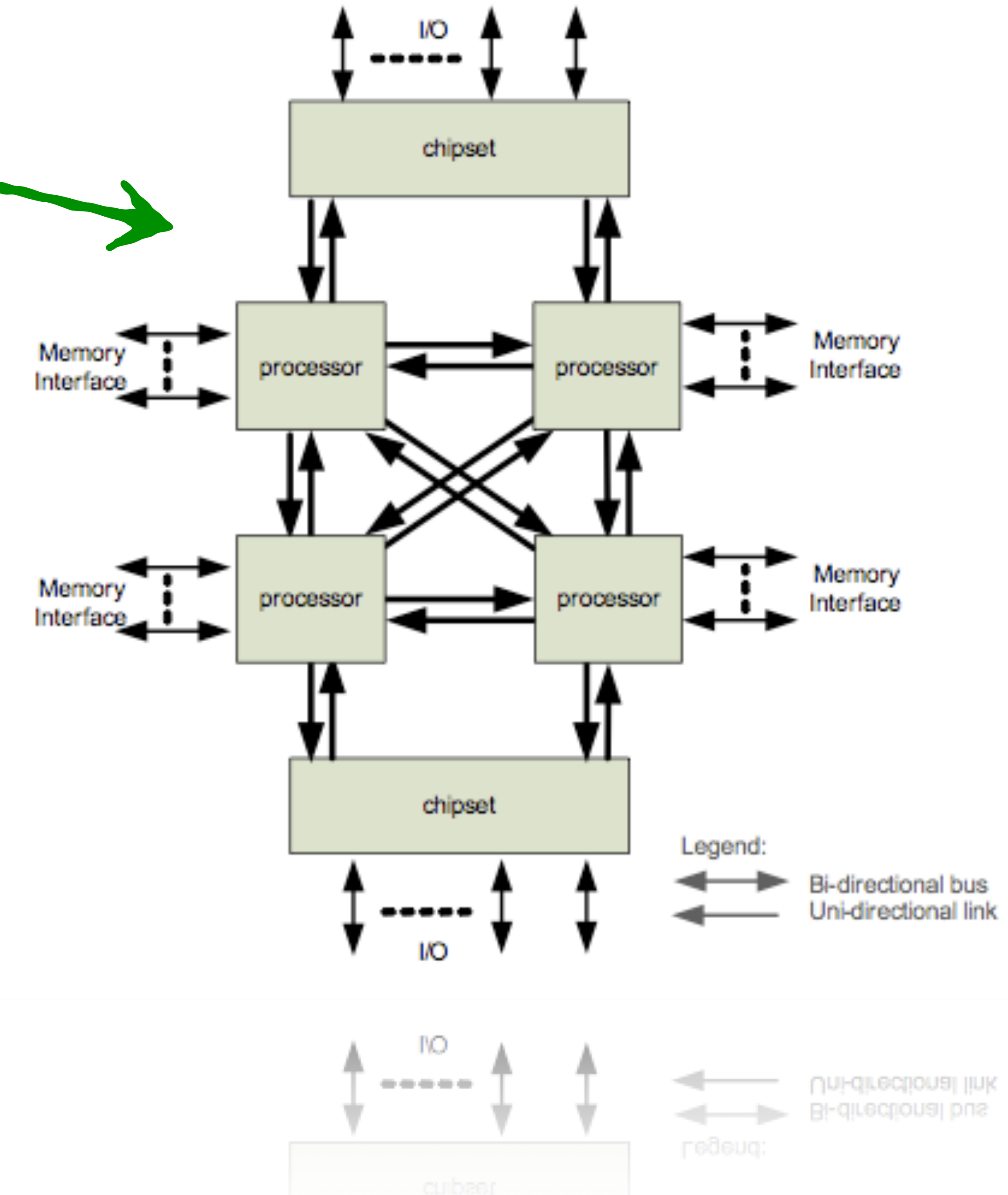




- M. Drocco. Parallel stochastic simulators in systems biology: the evolution of the species. Master's thesis, University of Torino, Italy, 2013.
- M. Aldinucci et al. On designing multicore-aware simulators for biological systems. PDP 2011. IEEE.
- M. Aldinucci et al. On parallelizing on-line statistics for stochastic biological simulations. Euro-Par 2011 Workshops. Springer.
- M. Aldinucci et al. Exercising high-level parallel programming on streams: a systems biology use case. ICDCS 2014. IEEE.

Heterogenous platforms: Multicores, accelerators, FPGA, PGAS …

- Multicore
  - E.g. Intel SandyBridge, AMD Opteron
  - cache-coherent
  - 10 or more core per socket (20 contexts)
  - **cc-NUMA** (as matter of a fact)
- NVidia/AMD GPGPU
  - SIMD, no global synch
  - performance only with proper and **not automatic** memory hierarchy management
- Intel MIC CPU/GPGPU
  - ring-based interconnection, variable coherency
  - NUMA
- FPGAs
  - general purpose cores
  - specialised cores
  - local and NUMA shared-memory (via PCI express)
- PGAS
  - Partitioned Global Address Space
  - cluster of multicores with (very) NUMA address space



Multicore, the simplest …

- 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)

On coherence and consistency

Subplot
(on programming models)

- Shared-memory access in multiprocessor platforms

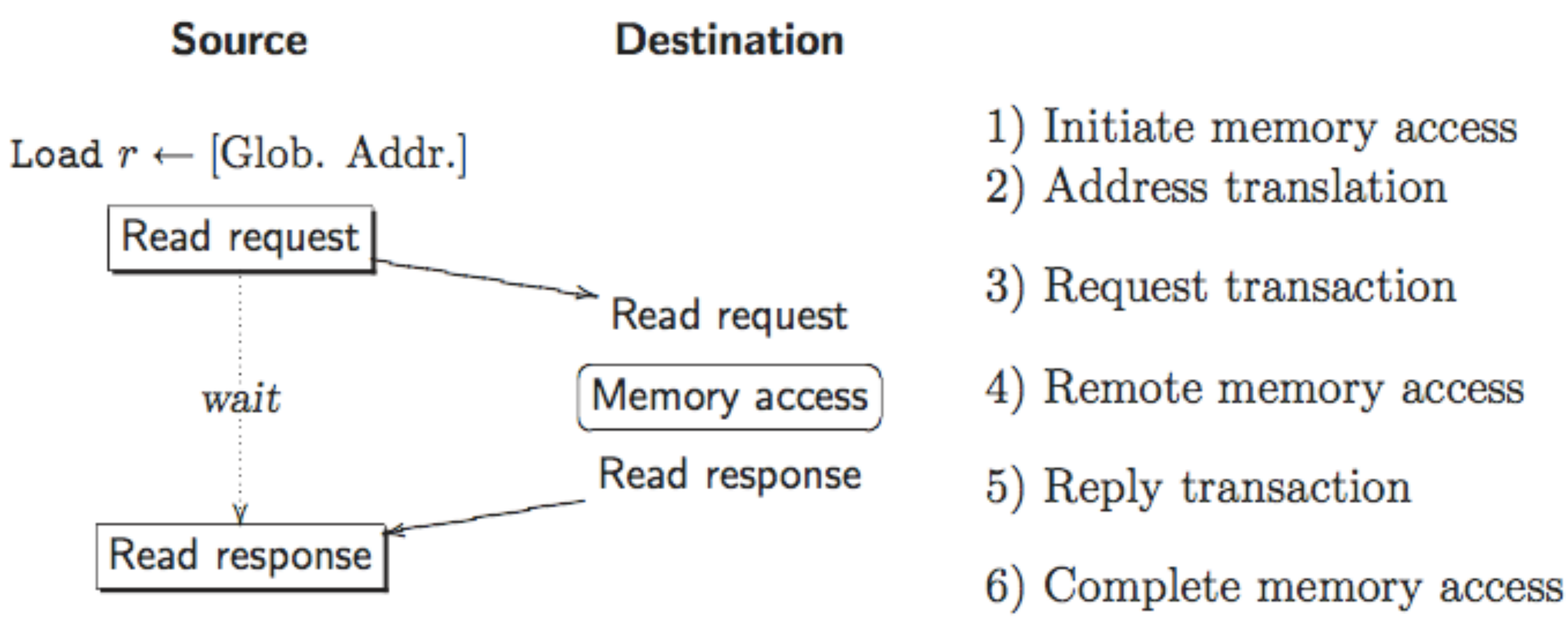
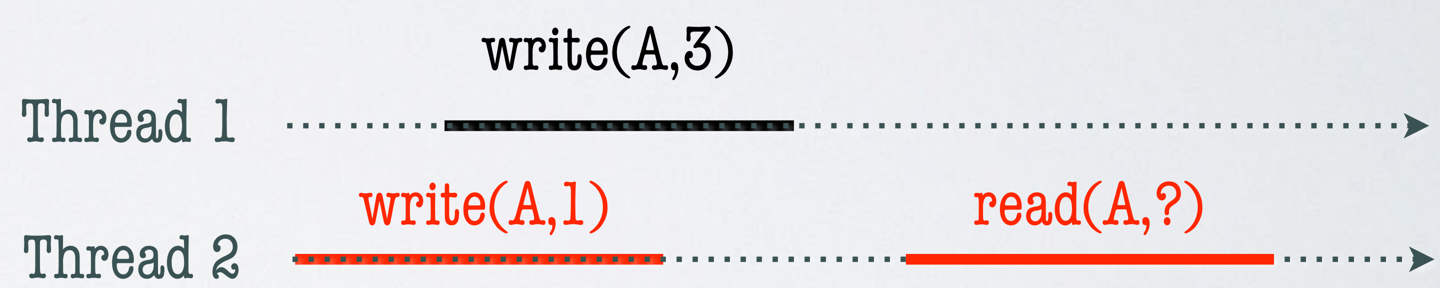


Figure 3.1: Shared address space abstraction:two-way request-response protocol.

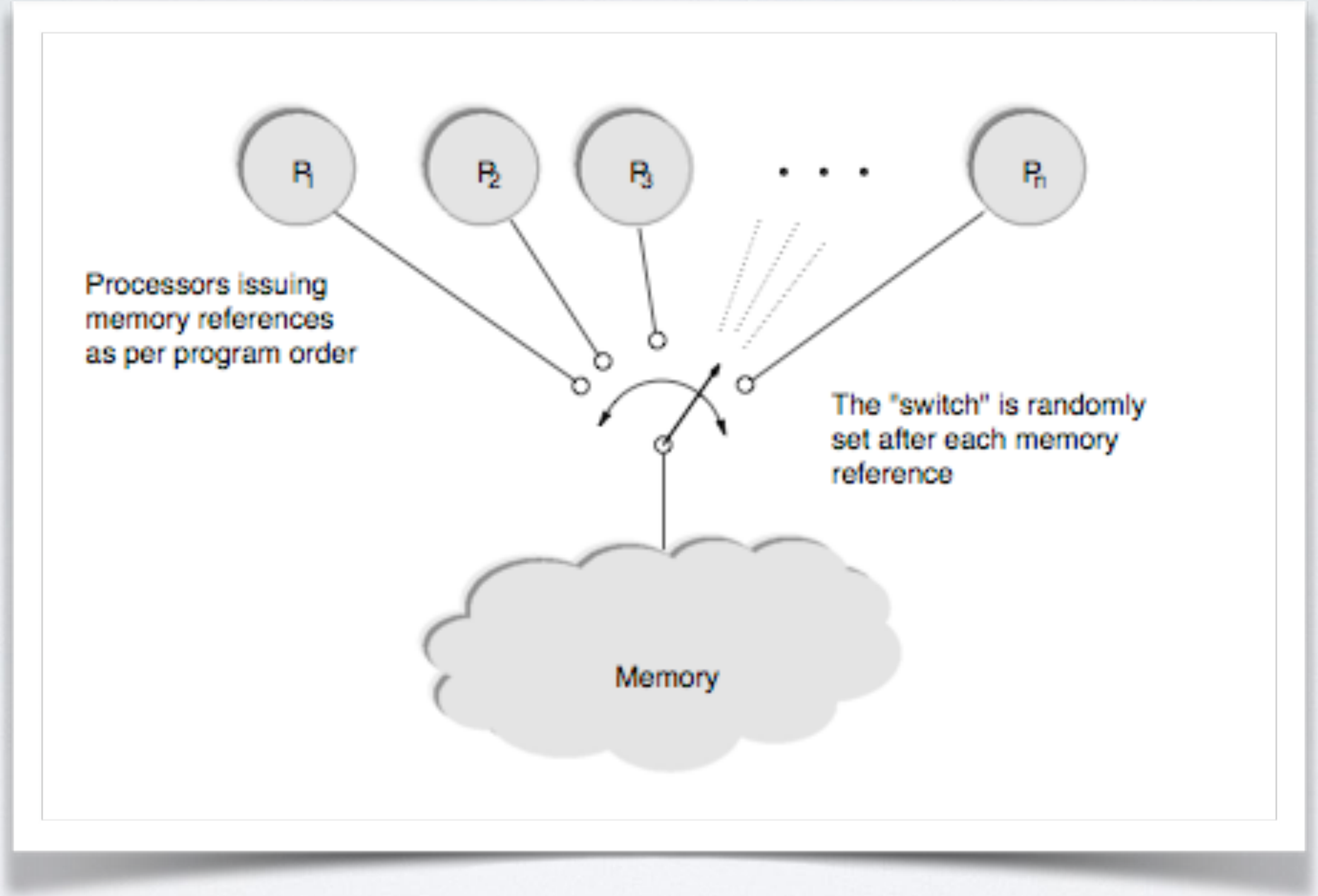
Coherence & Consistency

- Memory/Cache Coherence
  - Deal with multiple replicas of the same location in different caches
- Memory Consistency



  - Operations in memory take time and are "filtered" by caches
  - 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)
- These two phenomena mingle together ...

Sequential Consistency (Lamport 79)

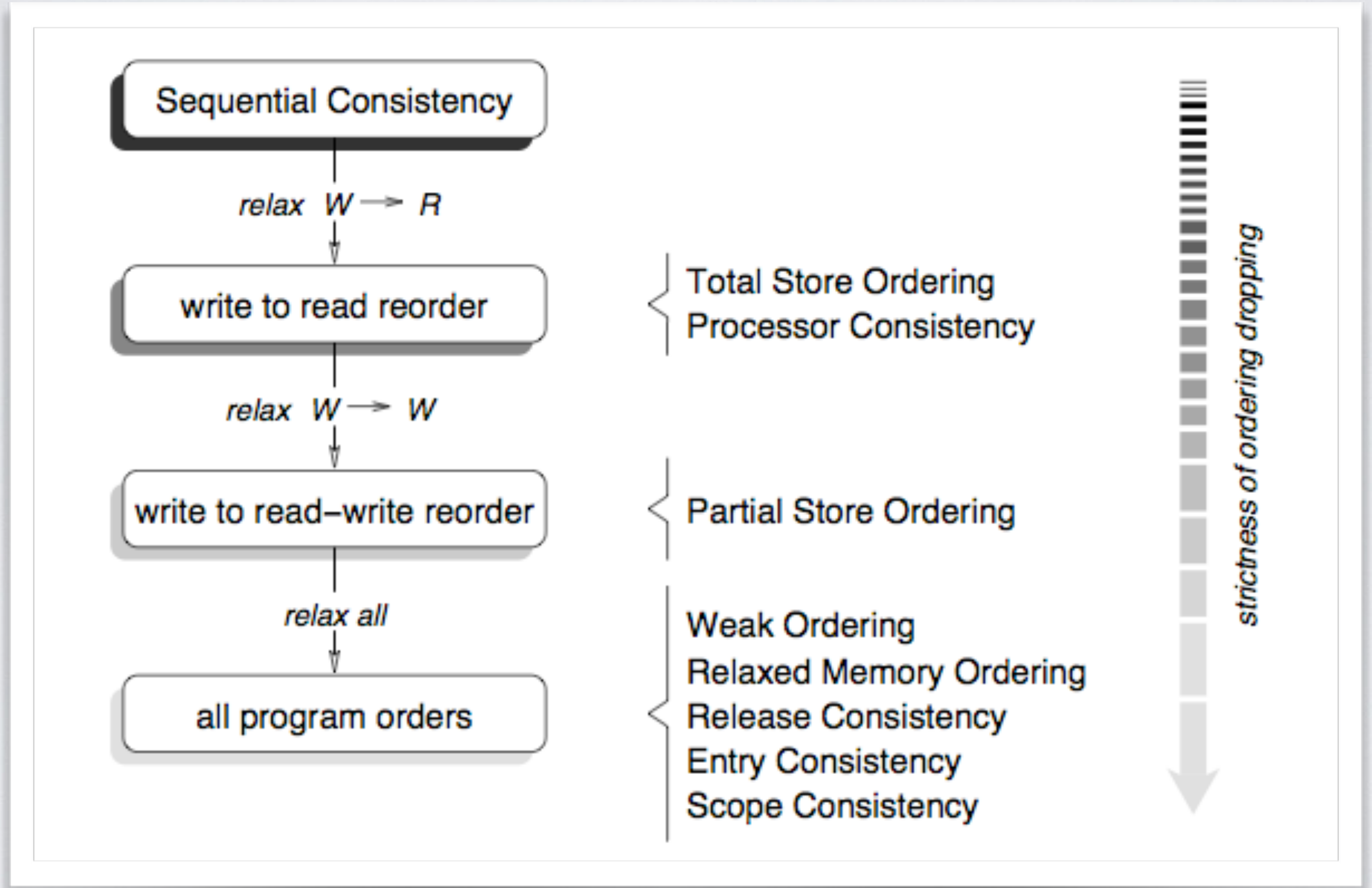


- Writes and reads are atomic
- In each thread, they are executed in the program order

Should not be confused with

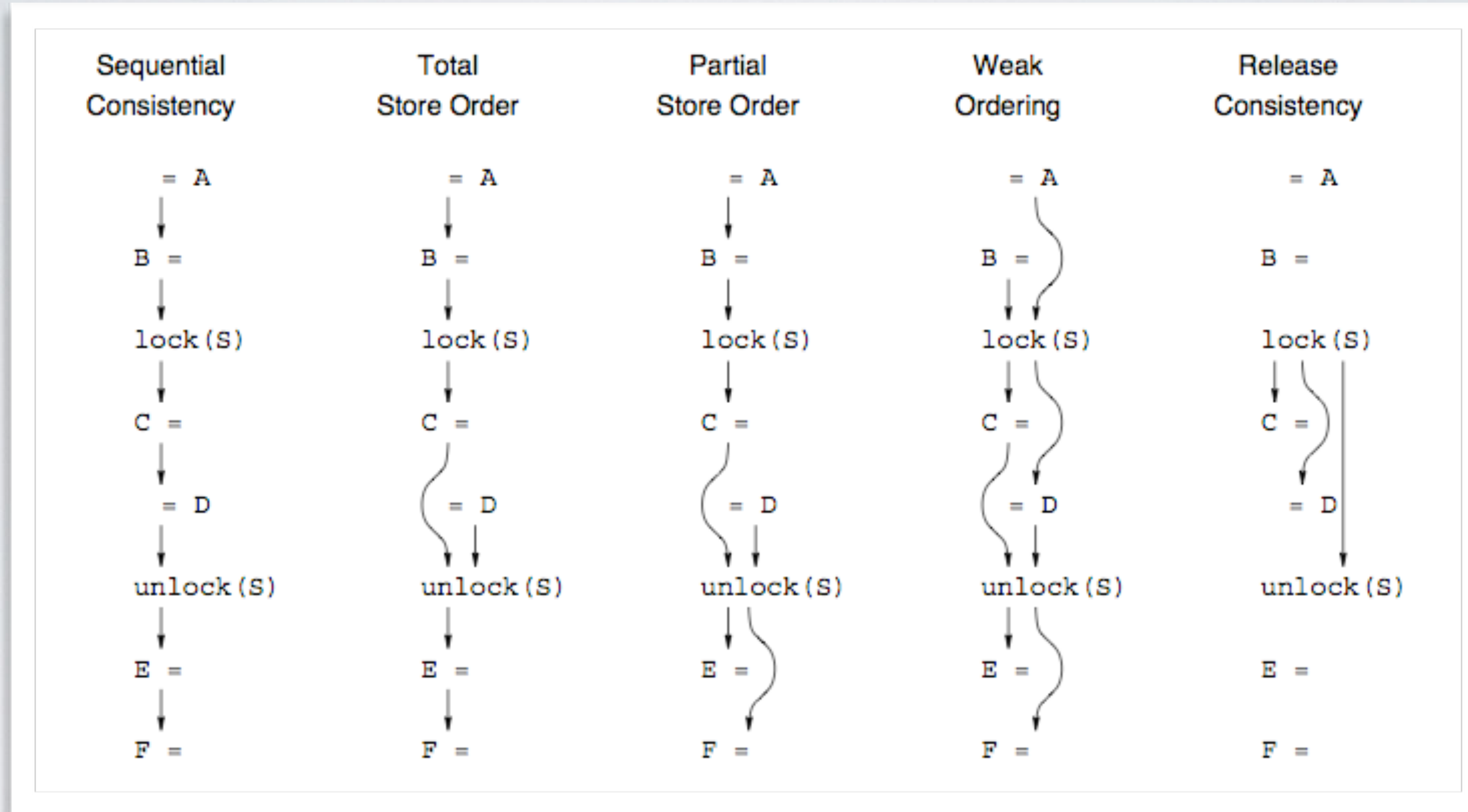
- **Dynamic scheduling** or **out-of-order scheduling** means that instructions are fetched and decoded in program order as presented by the compiler, but they are executed by the functional units in the order in which the operands become available at run time. Examples: scoreboarding or Tomasulo's algorithm.
- **Speculative execution** allows the processor to look at and schedule for execution instructions that are not necessarily going to be useful to the program's execution. Instruction after the speculation point (e.g. branch) continue to be decoded, issued and executed, but these are not allowed to commit their values until all prior speculation have been resolved.
- **Out-of-order execution does not mean that the results of instructions is made visible out-of-order at memory system level.**

HW consistency rationale



ParaPhrase

Effect of assignment in memory



- Effect in memory does not necessarily follow the order they are issued
- There not necessarily exist a total order across different processors

Mem Consistency: examples

- Current Processors:
  - x86: Total Store Order (at least)
  - Arm, PowerPC: Weak Ordering
  - Alpha: Release Consistency
- Any Sequential Consistency?
  - No
  - It is not efficient

Relaxed consistency ...

A=B=0

Pi	Pj
write(B,1) if (A==0) ...	write(A,1) if (B==0) ...

```
A=B=0, B=1, A==0, A=1 , B==0     (TRUE, FALSE)
A=B=0, B=1, A=1 , A==0, B==0     (FALSE, FALSE)
A=B=0, B=1, A=1 , B==0, A==0     (FALSE, FALSE)
A=B=0, A=1, B==0, B=1 , A==0     (FALSE, TRUE)
A=B=0, A=1, B=1 , A==0, B==0     (FALSE, FALSE)
A=B=0, A=1, B=1 , B==0, A==0     (FALSE, FALSE)
```

Ideally NO, under Sequential Consistency NO

Can both ifs be evaluated to TRUE?

Under weaker models, YES

Memory Consistency: TSO

int A, *B;

Pi	Pj
write(B,NULL) write(A,1) write(B,&A)	write(B,NULL) write(A,2) ... if (B!=NULL) PRINT read(A)

- Which is the printed value?
  - Under Sequential Consistency 1, under Total Store Order 1 or no print
  - Under more relaxed models, either 1 or 2 or no print

Again on atomic operations

- In concurrent programming, an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously.
  - Atomicity is a guarantee of isolation from concurrent processes.
  - Additionally, atomic operations commonly have a succeed-or-fail definition, they either successfully change the state of the system, or have no apparent effect.
- Atomic operation really does not actually occur instantaneously. The system behaves as if each operation occurred instantly, separated by pauses.

Linearizability

- Linearizability [Herlihy 87] is more restrictive w.r.t. atomic operation (cannot be interrupted), which are usually vague about when an operation is considered to begin and end
  - Atomicity of sequences are usually enforced via mutexes

Linearizability

- A history is a sequence of invocations and responses made of an object by a set of threads. Each invocation of a function will have a subsequent response
- A sequential history is one in which all invocations have immediate responses
- A history is serializable if
  - its invocations and responses can be reordered to yield a sequential history
  - that sequential history is correct according to the sequential definition of the object
- A history is linearizable if serializable and
  - if a response preceded an invocation in the original history, it must still precede it in the sequential reordering

Serialization & Linearization

A: ➥ lock
← lock outcome

B: ➥ lock
← lock outcome

	A ➥ lock	B ➥ lock	A← lock failed	B←lock success
1	B ➥ lock	B←lock success	A ➥ lock	A ←lock failed
2	A ➥ lock	A←lock failed	B ➥ lock	B ←lock success

2. Not a valid history. A should have get the lock
1. Valid history. Also a linearization.

An object (as opposed to a history) is linearizable if all valid histories of its use can be linearized.

Much harder to prove!

Serialization & Linearization (example)

Assume B initially holds the lock

A ➥ lock	A←lock ok	B ➥ unlock	B unlock ok	A ➥ unlock	A←unlock ok

Not a valid history
There is a point in which A and B hold the lock
Cannot be linearized

B ➥ unlock	B←unlock ok	A ➥ lock	A←lock ok	A ➥ unlock	A←unlock ok

When relaxing ordering between invocation and responses
it can be reordered to a valid history
(it is serializable)

Serialization & Linearization (example)

response invocation

Assume B initially holds the lock

A ➥ lock	A←lock ok	B ➥ unlock	B unlock ok	A ➥ unlock	A←unlock ok

Not a valid history

There is a point in which A and B hold the lock

Cannot be linearized

B ➥ unlock	B←unlock ok	A ➥ lock	A←lock ok	A ➥ unlock	A←unlock ok

When relaxing ordering between invocation and responses it can be reordered to a valid history

(it is serializable)

Linearizability (alternative def)

- The definition of linearizability is equivalent to the following:
  - All function calls have a linearization point at some instant between their invocation and their response
  - All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition

Linearizability (alternative def)

- This alternative is usually much easier to prove. It is also much easier to reason about as a user, largely due to its intuitiveness.
  - This property of occurring instantaneously, or indivisibly, leads to the use of the term atomic as an alternative to the longer "linearizable".
- In the examples, the linearization point of the counter built on CAS is the linearization point of the first (and only) successful CAS update.
  - A counter built using locking can be considered to linearize at any moment while the locks are held, since any potentially conflicting operations are excluded from running during that period

Does all this affect the programming?

- Let us focus on two typical low-level synchronisation paradigms for the shared memory model
  - Mutual Exclusion (mutex)
  - Producer Consumer
  - there are more, clearly, but they are crucially important for this talk
    - M.Herlihy, N. Shavit. The art of multiprocessor programming. Elsevier

Mutex is a quite powerful mechanism

- Mutual exclusion
  - Mutex algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.
  - enforced via locks/unlocks
  - requires deadlock-freedom
  - typically used as the foundation of higher level mechanisms, such as semaphores, monitors, ...
  - Classic algorithms: Peterson, Lamport, Dekker, ...
    - in the "register" model (i.e. a read-write memory)

Peterson's mutex (Peterson 1981)

```java
class Peterson implements Lock {
  // thread-local index, 0 or 1
  private volatile boolean[] flag = new boolean[2];
  private volatile int victim;
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true;               // I'm interested
    victim = i;                   // you go first
    while (flag[j] && victim == i) {}; // wait
  }
  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;              // I'm not interested
  }
}
```

Figure 2.6 The Peterson lock algorithm.

- Works for 2 threads, require SC (or PRAM Consistency)
- Starvation-free, Deadlock-free

Bakery (Lamport 1976)

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Figure 2.9 The Bakery lock algorithm.

- Works for n threads, require SC (or PRAM Consistency)

Are they working on a x86?

- No!
  - try them, they are going to fail half of the times
- So, what can we do?
- "transactional" operations (CAS)
  - extend the "register" model with "transactional" operations (CAS)
    - Compare-And-Swap, Test-And-Set, Load-Linked-Store-Conditional
  - what do they do?
    - execute a read AND a write as an atomic operation
    - acts a memory fences, all in-flight operations are committed before proceeding

Lock with CAS? Easy job.

```
volatile int lock = 0;

void Critical() {

  while (TestAndSet(&lock) == 1);    // acquire lock

  critical section //only one thread can be in this section at a time

  lock = 0                                        // release lock

}
```

So, what is the problem?

- Atomic operations are memory fences
  - each atomic operation requires the reconciliation of caches
- They do affect performance!

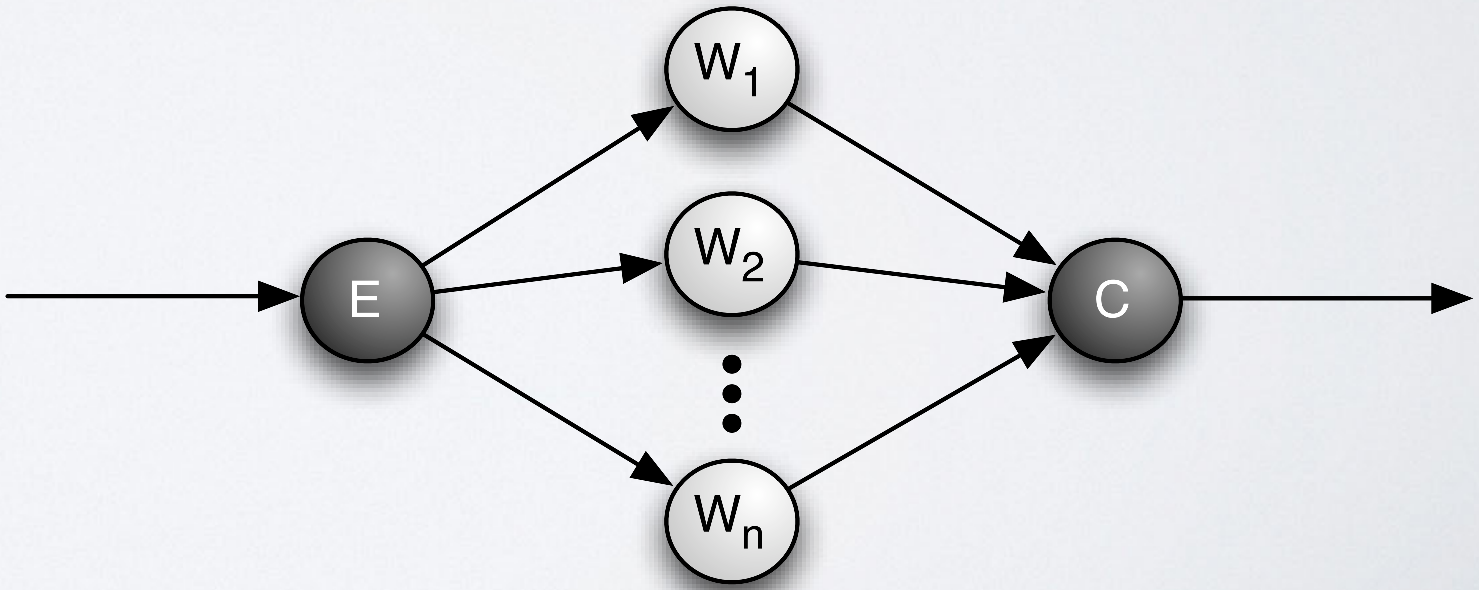
Micro-benchmarks: farm of tasks

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

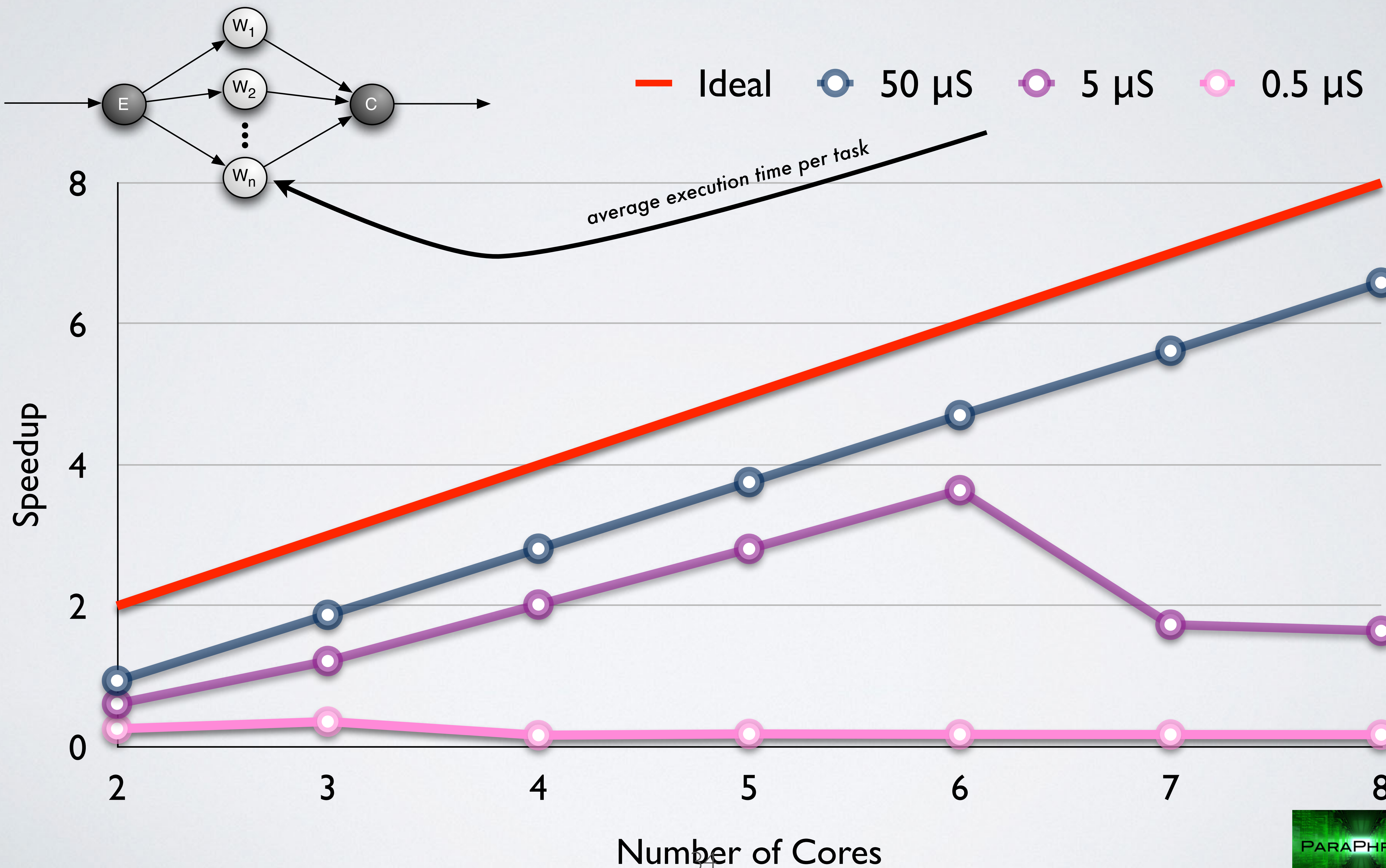
```
void Emitter () {
  for ( i =0; i <streamLen;++i){
    task = create_task ();
    queue=SELECT_WORKER_QUEUE();
    queue ->PUSH(task);
  }
}
```

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

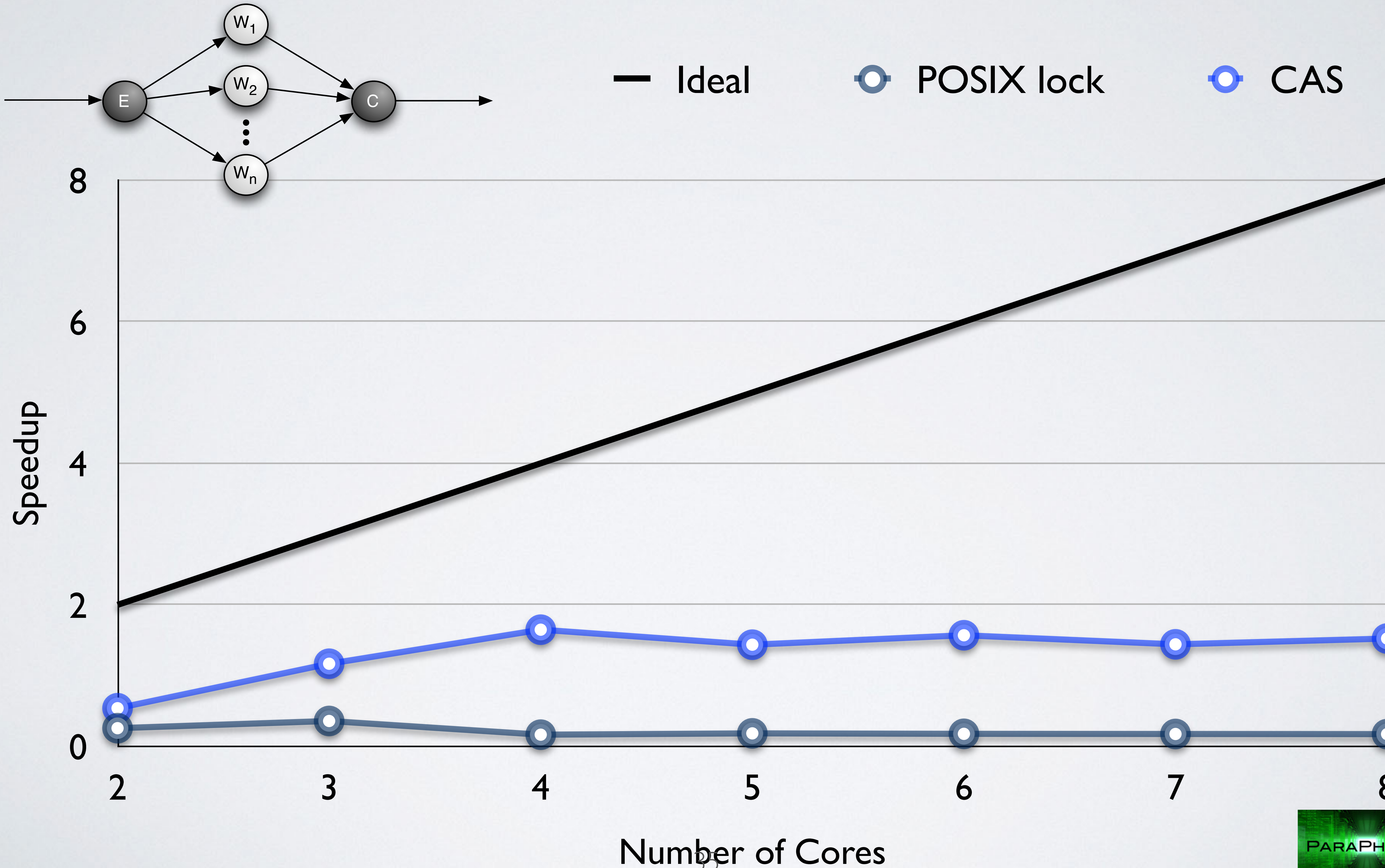
```
void Worker() {
  while (!end_of_stream){
  myqueue ->POP(&task);
  do_work(task) ;
  }
}
```



Task farm with POSIX lock/unlock



Lock vs Nonblocking CAS (fine grain 0.5μS)

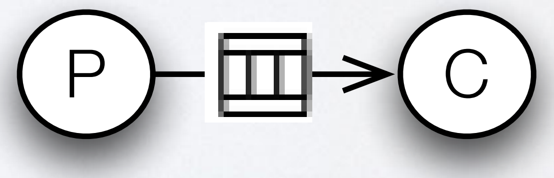


Can we avoid locks?

- Yes, in many ways using CAS (under relaxed memory models)
  - actually building concurrent data structures accessed via CAS
  - they perform better than locks-based, but still they fence the memory
- and what about lock-free, CAS-free?
  - Mutex cannot, Producer Consumer can be done
    - also under some relaxed memory model, not all of them, however
    - notice that Producer Consumer is inherently weaker with respect to Mutex because it does requires the cooperation of partners whereas Mutex is required to be deadlock-free

In designing FastFlow we re-started 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
- The FIFO queue



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

I'll focus here

- In the general case:
  - Stronger properties are harder to maintain

ParaPhrase

Nonblocking algorithms

- An algorithm is **obstruction-free** if at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation.
- An algorithm is **lock-free** if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress (for some sensible definition of progress). All wait-free algorithms are lock-free.
  - Lock-freedom allows individual threads to starve but guarantees system-wide throughput.
- An algorithm is **wait-free** if every operation has a bound on the number of steps the algorithm will take before the operation completes.
  - Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom.

Related Work: Lock-free, CAS-

- 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 ➠ Impossible
  - Nonblocking without CAS - with mediator thread ➠ FastFlow

First attempt: Lamport FIFO

```
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;
}
```

- Works under SC
- Doesn't work under weaker models
  - Because of the need to serialise data and head updates
- Even if it were working it pushing lot of pressure on coherence system because both producer and consumer need to share both head and tail index of the queue

Finally, FastFlow-like SPSC queues

```
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

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

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

FastFlow FIFO

Finally, FastFlow-like SPSC queues

```
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

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

cking(data) {
uffer[tail];
 == data) {
 EWOULDBLOCK;

il] = NULL;
XT(tail);
```

astFlow FIFO

WMB enforce store ordering on successive cells/indexes. Also it enforces transitivity in pointer traversal.

ParaPhrase

Finally, FastFlow-like 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

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

        cking(data) {
        uffer[tail];
         == data) {
         EWOULDBLOCK;

        il] = NULL;
        XT(tail);
```

astFlow FIFO

WMB enforce store ordering on successive cells/indexes. Also it enforces transitivity in pointer traversal.

ParaPhrase

FastFlow queues tolerate TSO

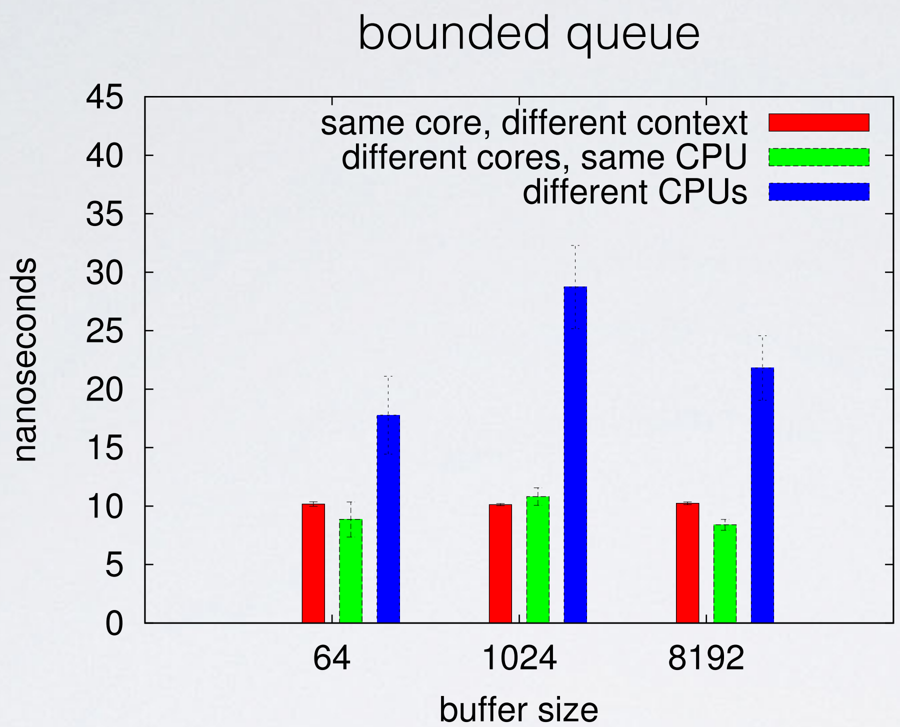
- Do they eventually work?
  - Yes, under SC and Total Store Order (TSO), and we will see they are very efficient
    - TSO is required because we should enforce the transitivity of updates, as shown in a previous example
  - That is important because x86 is TSO and on all architectures it can be emulated by using a single write barrier
  - J. Giacomoni et al. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. PPoPP 2008. ACM.
- The result cannot be directly extended to Multiple Producer and Multiple Consumer

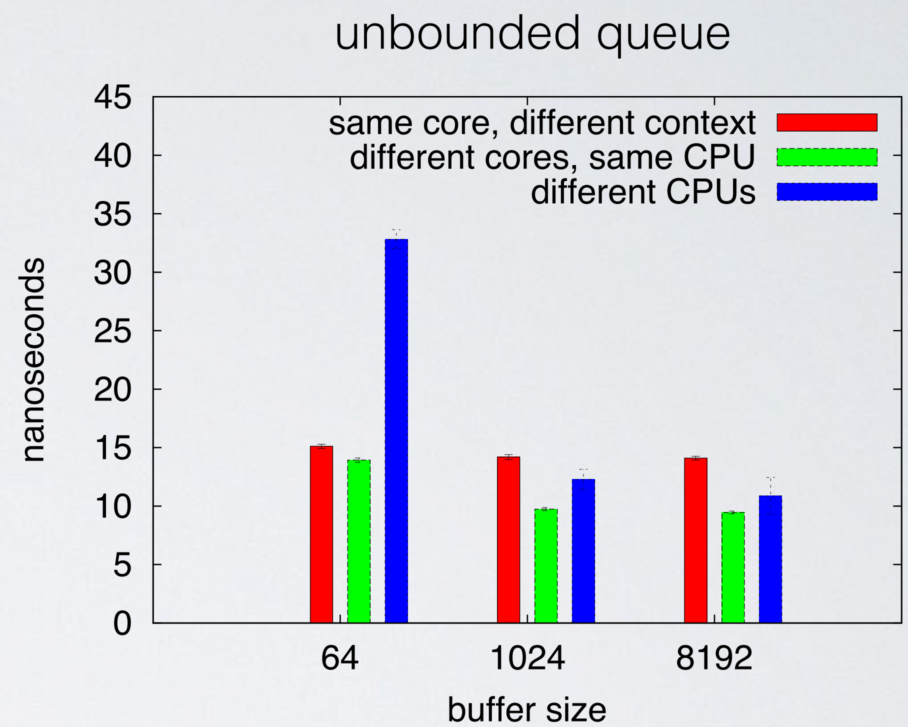
MPI shmem impl is ~190 ns at best (D.K. Panda)

High quality channels

ShMem latency

M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi core systems. Euro-Par 2012.



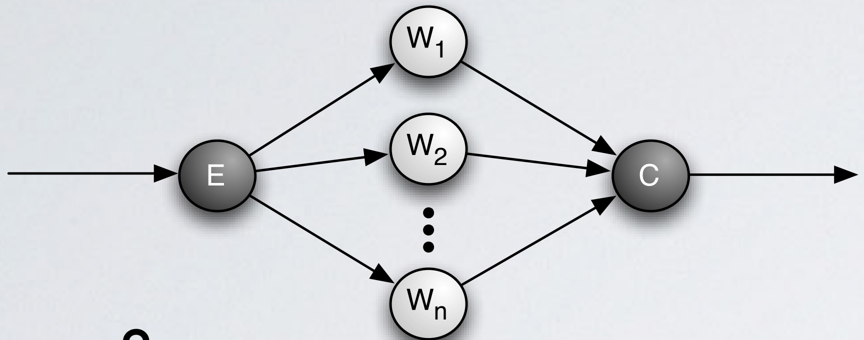


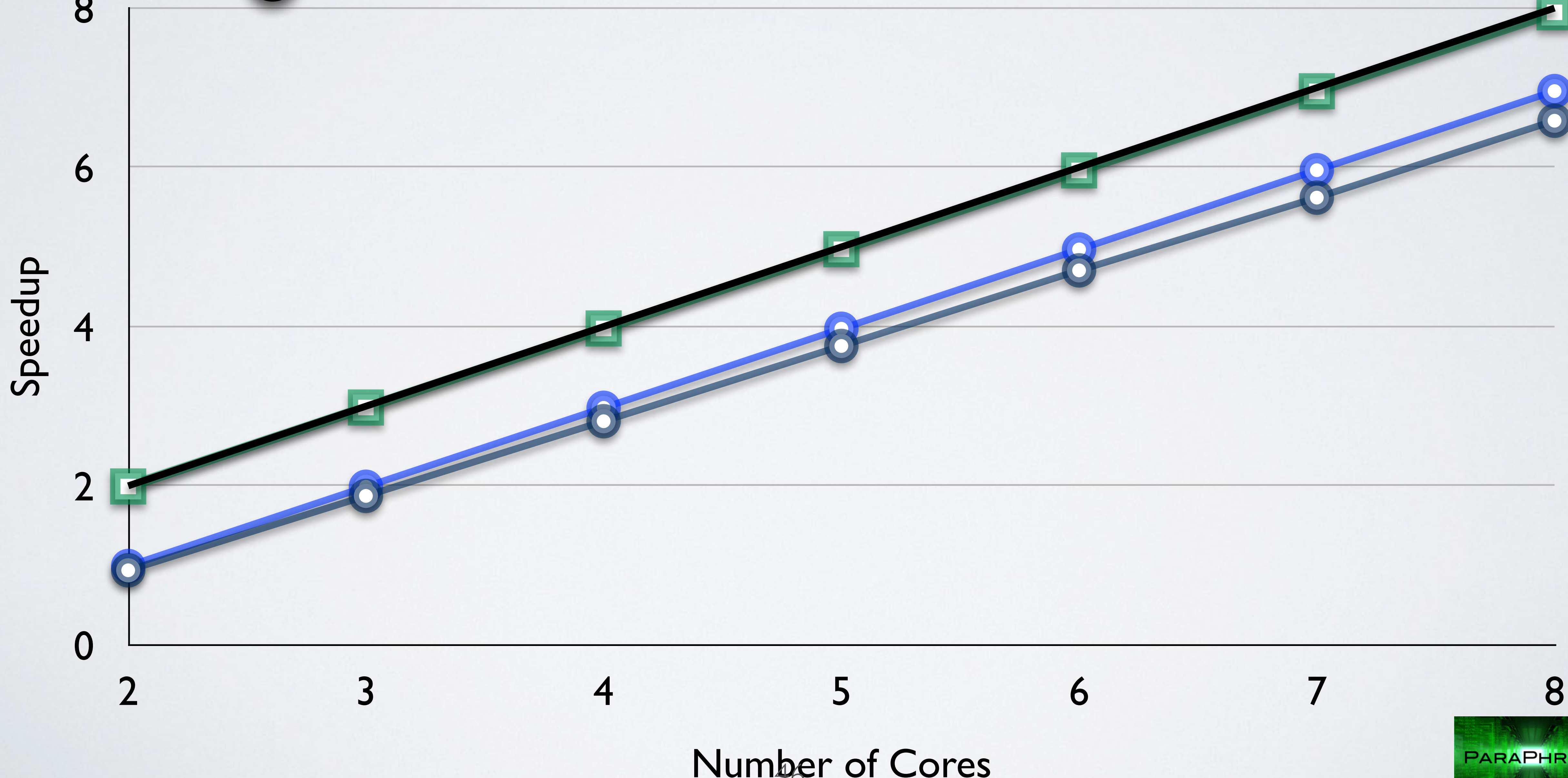
Message size (bytes)	ib_write_bw (Mb/s)	MPI (Mb/s)	FastFlow /IB (Mb/s)	FastFlow/ZMQ /IPoIB (Mb/s)
10	300	192	129	0.7
100	3,600	1,816	1,300	7.0
1,024	22,900	13,936	10,591	70.0
5,000	25,200	23,880	19,761	300.0
10,000	25,500	25,128	20,479	500.0
25,000	25,700	12,408	20,051	1,100.0
50,000	25,800	16,232	21,019	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

Distributed throughput

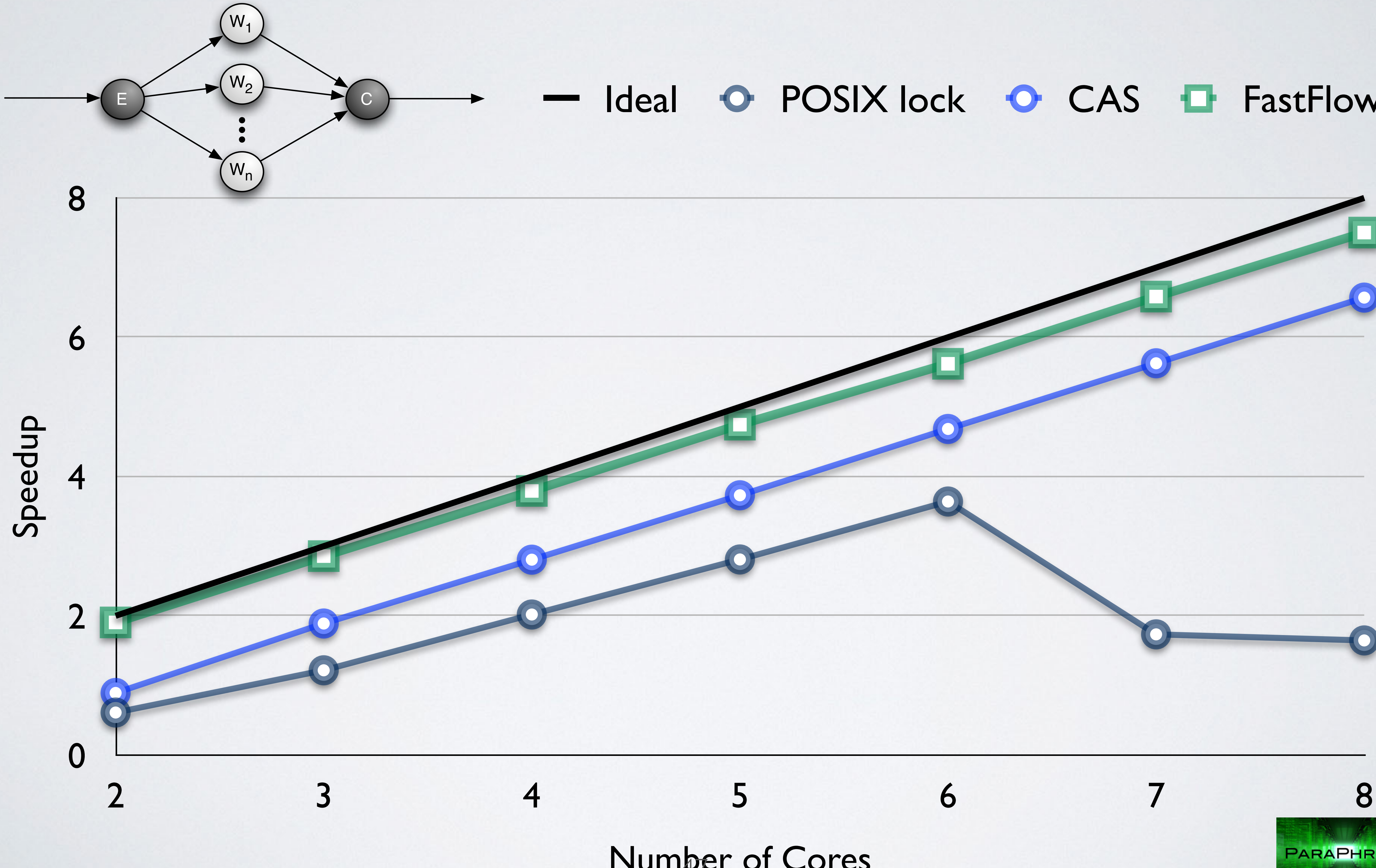
A. Secco, I. Uddin, G. Peretti Pezzi, M. Torquati. Message passing on InfiniBand RDMA for parallel run-time supports. IEEE PDP 2014.

Lock vs CAS vs SPSC/FastFlow

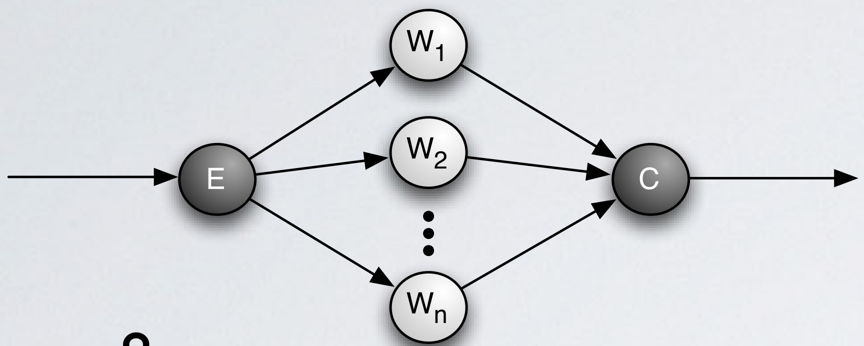


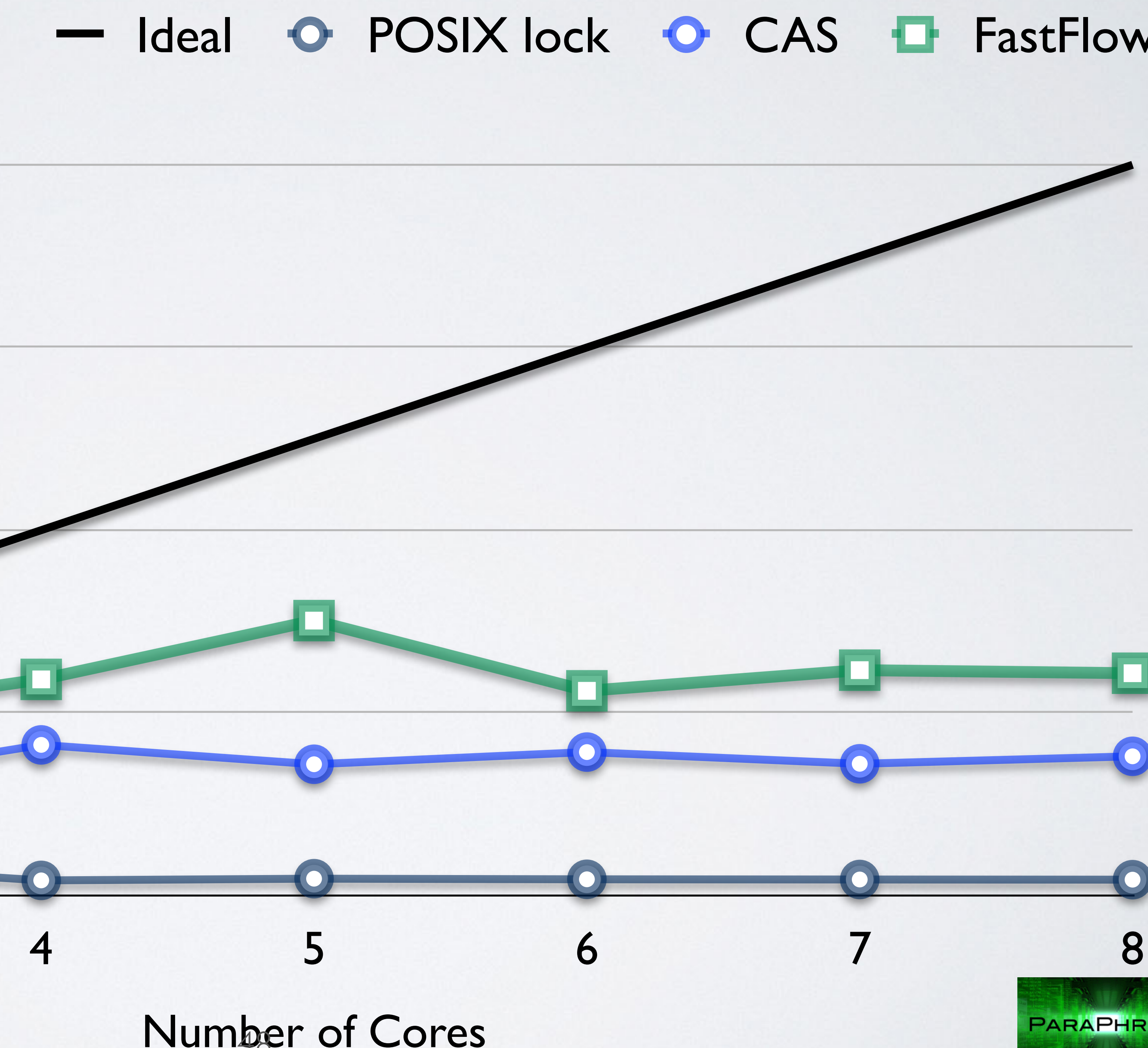


Lock vs CAS vs SPSC/FastFlow (5



Lock vs CAS vs SPSC/FastFlow



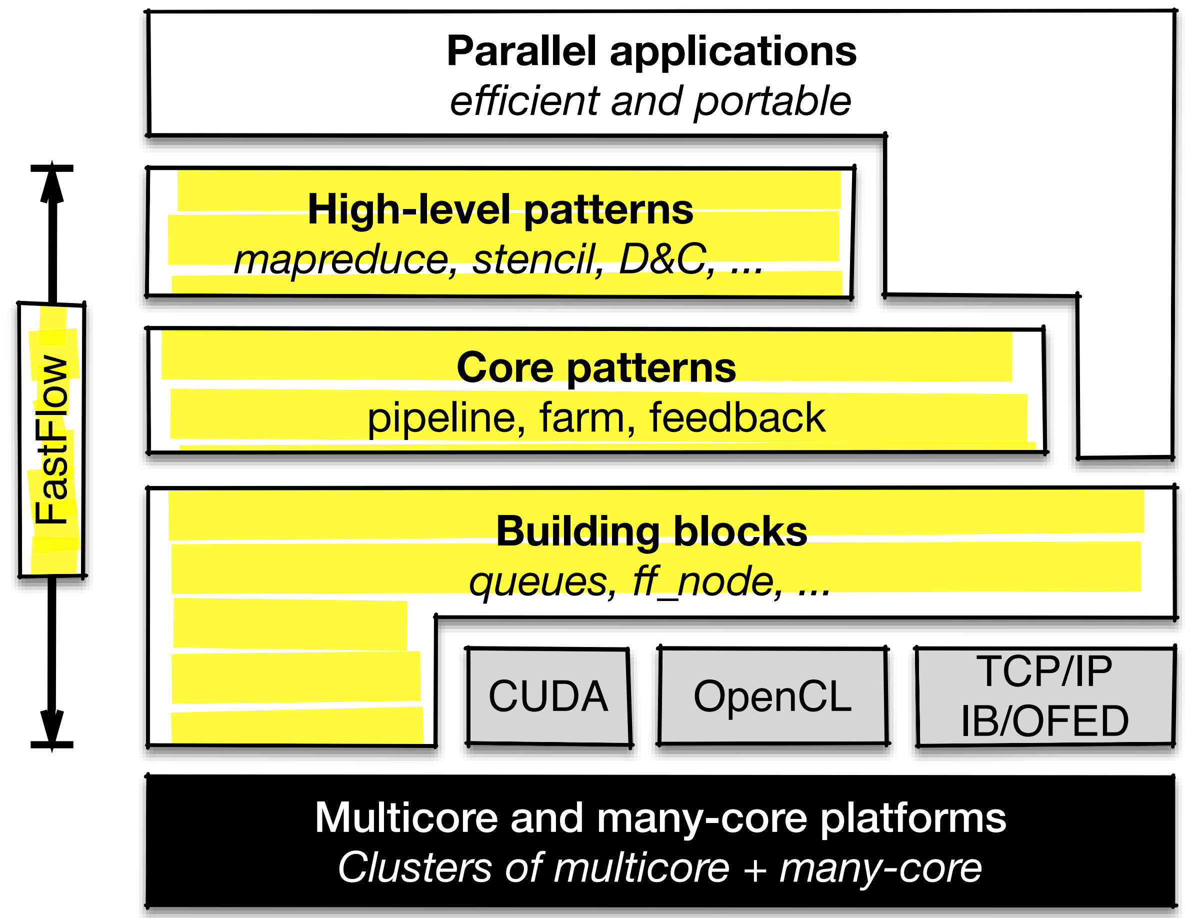


FastFlow

Lock-free and CAS-free?

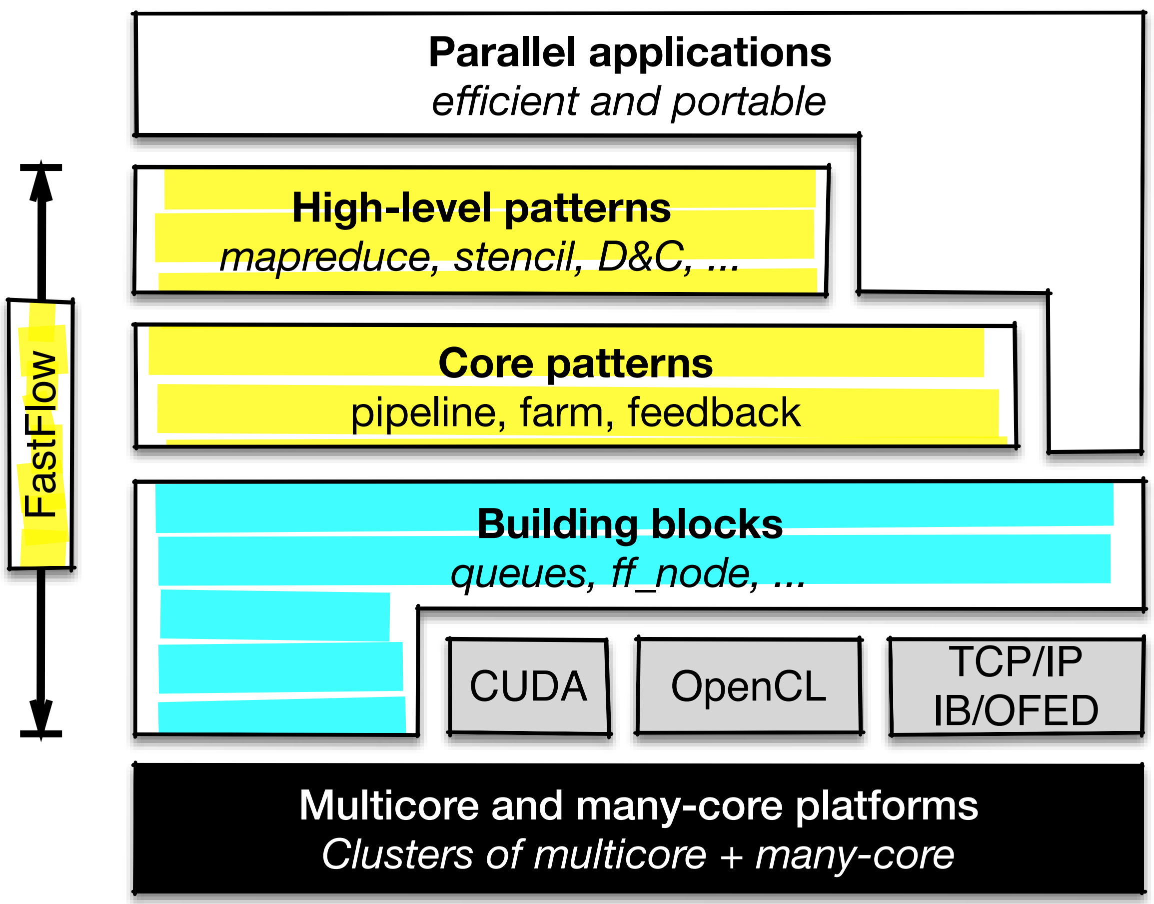
- Mutex cannot be done
  Single-Producer-Single-Consumer (SPSC) can be done
  - Producer-Consumer is inherently weaker with respect to Mutex
  - It does require the cooperation of partners whereas Mutex does not
- Expressive enough to build a streaming (or dataflow) programming framework
  - MPMC = SPSC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

FastFlow layered architecture



- 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)

Building blocks



Foundations
(of any concurrent model actually)

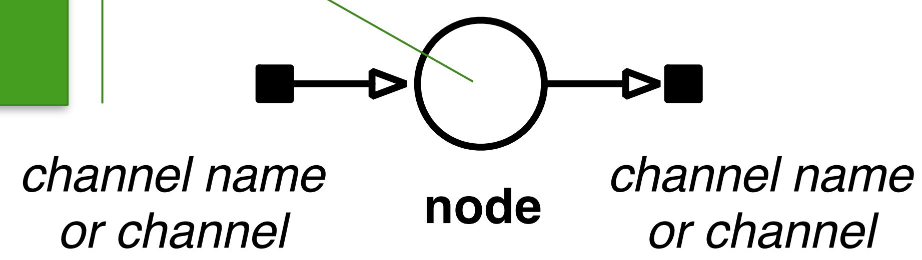
- Concurrent activities
  - actors, processes, threads
- Synchronisations
  - channels, semaphores, condVars, …
- Data movements
  - memcopy, reference, …

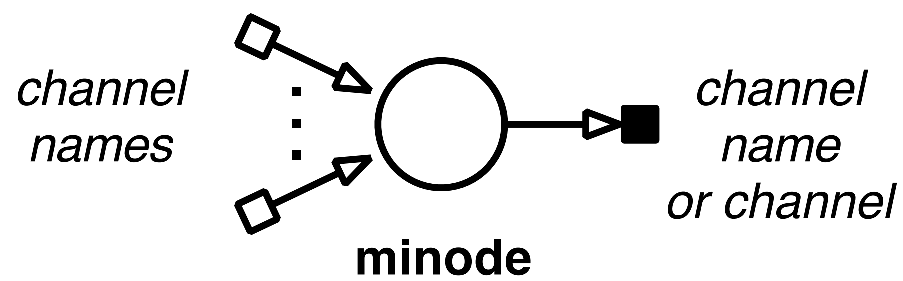
FastFlow concepts

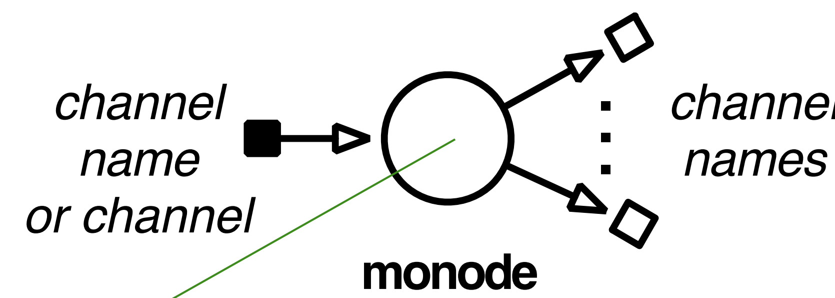
- Implementation based around the concept of node (ff_node class abstraction)
- A node is a concurrent abstraction with an input and output SPSC queue
- Queues are lock/CAS-free and can be bounded or unbounded
- nodes are connected through queues
- A node can be sequential or parallel
  - pipeline, farm, map are parallel ff_node(s)

Building blocks

Nonblocking thread, can be turned into blocking mode via extra-functional interface







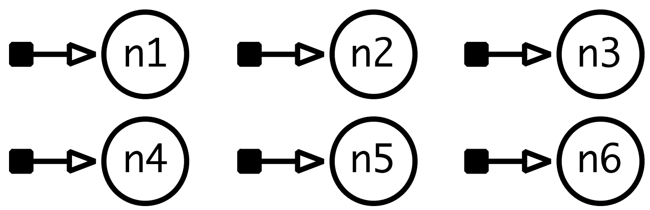
M-nodes are mediators, M-side of the interface always has channel names, i.e. should be bound with a channel

ff_node

```
struct myNode: ff::ff_node {
  int svc_init() { // optional
       // initialization part if needed
       // called only once after the thread is started
       return 0;  //  < 0  means that initialization failed
  }
  void *svc(void *task) {  // mandatory
       // do some computation on the input task
       // called each time an input task is available

       return task;  // see next slides....
  }

  void svc_end() {  // optional
      // termination part, if needed
      // called only once if the svc method returns NULL
      // or if EOS has arrived from the input stream
  }
} ;
```

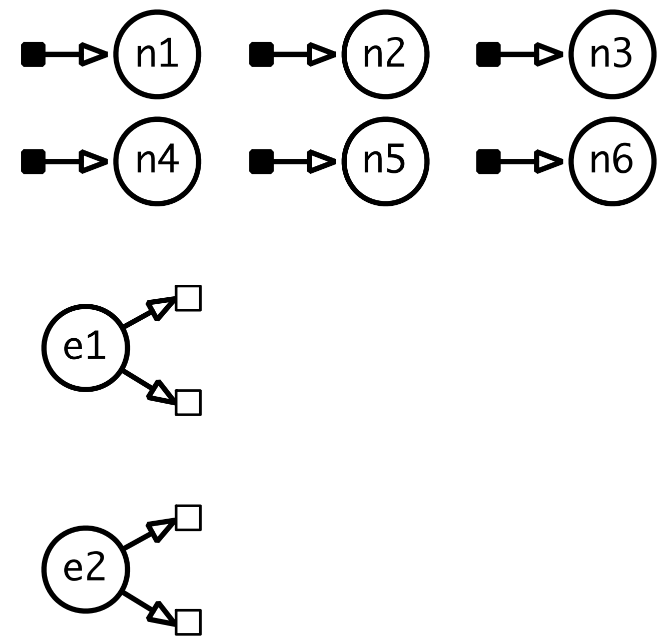
Create a DAG/graph

```
// Create 6 generic nodes - max 1 in channel 1 out
channel
N n1(1),n2(2),n3(3),n4(4),n5(5),n6(6);
n1.create_input_buffer(100); …
n6.create_input_buffer(100);

// 2 emitters: e1->n1, e1->n2, e2->n3, e2->n4
std::vector<ff_node*> we1, we2;
we1.push_back(&n1); we1.push_back(&n2);
we2.push_back(&n3); we2.push_back(&n4);
E e1(we1,1,ntasks), e2(we2,2,ntasks);

// link n1->n5 and n4->n6
n1.set_output(n5.get_in_buffer());
n4.set_output(n6.get_in_buffer());

// 1 collector + linking: n2->c1, n3->c1
n2.create_output_buffer(100);
n3.create_output_buffer(100);
std::vector<ff_node*> wc1;
wc1.push_back(&n2);
wc1.push_back(&n3);
C c1(wc1,1 /* id */);
c1.create_input_buffer(100);

// run all nodes and wait end
n1.run(); … n6.run(); e1.run(); e2.run(); c1.run();
n1.wait(); … n6.wait(); e1.wait(); e2.wait(); c1.run();
```



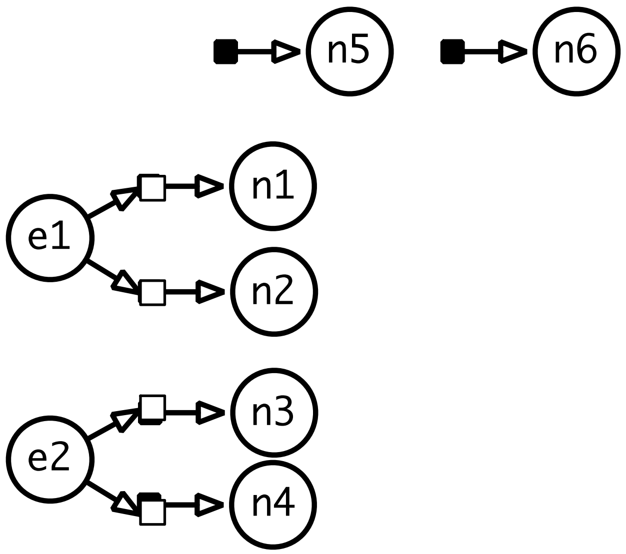
Create a DAG/graph

```
// Create 6 generic nodes - max 1 in channel 1 out
channel
N n1(1),n2(2),n3(3),n4(4),n5(5),n6(6);
n1.create_input_buffer(100); …
n6.create_input_buffer(100);

// 2 emitters: e1->n1, e1->n2, e2->n3, e2->n4
std::vector<ff_node*> we1, we2;
we1.push_back(&n1); we1.push_back(&n2);
we2.push_back(&n3); we2.push_back(&n4);
E e1(we1,1,ntasks), e2(we2,2,ntasks);

// link n1->n5 and n4->n6
n1.set_output(n5.get_in_buffer());
n4.set_output(n6.get_in_buffer());

// 1 collector + linking: n2->c1, n3->c1
n2.create_output_buffer(100);
n3.create_output_buffer(100);
std::vector<ff_node*> wc1;
wc1.push_back(&n2);
wc1.push_back(&n3);
C c1(wc1,1 /* id */);
c1.create_input_buffer(100);

// run all nodes and wait end
n1.run(); … n6.run(); e1.run(); e2.run(); c1.run();
n1.wait(); … n6.wait(); e1.wait(); e2.wait(); c1.run();
```



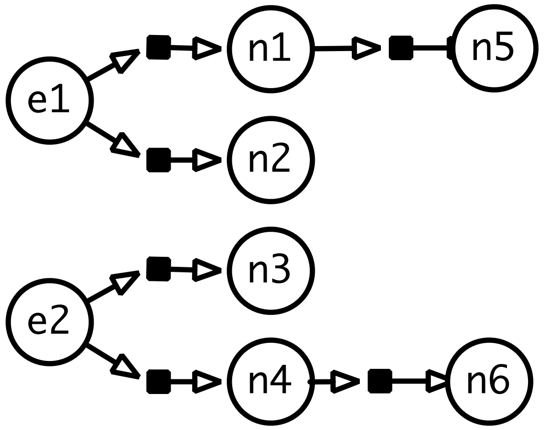
Create a DAG/graph

```
// Create 6 generic nodes - max 1 in channel 1 out
channel
N n1(1),n2(2),n3(3),n4(4),n5(5),n6(6);
n1.create_input_buffer(100); …
n6.create_input_buffer(100);

// 2 emitters: e1->n1, e1->n2, e2->n3, e2->n4
std::vector<ff_node*> we1, we2;
we1.push_back(&n1); we1.push_back(&n2);
we2.push_back(&n3); we2.push_back(&n4);
E e1(we1,1,ntasks), e2(we2,2,ntasks);

// link n1->n5 and n4->n6
n1.set_output(n5.get_in_buffer());
n4.set_output(n6.get_in_buffer());

// 1 collector + linking: n2->c1, n3->c1
n2.create_output_buffer(100);
n3.create_output_buffer(100);
std::vector<ff_node*> wc1;
wc1.push_back(&n2);
wc1.push_back(&n3);
C c1(wc1,1 /* id */);
c1.create_input_buffer(100);

// run all nodes and wait end
n1.run(); … n6.run(); e1.run(); e2.run(); c1.run();
n1.wait(); … n6.wait(); e1.wait(); e2.wait(); c1.run();
```



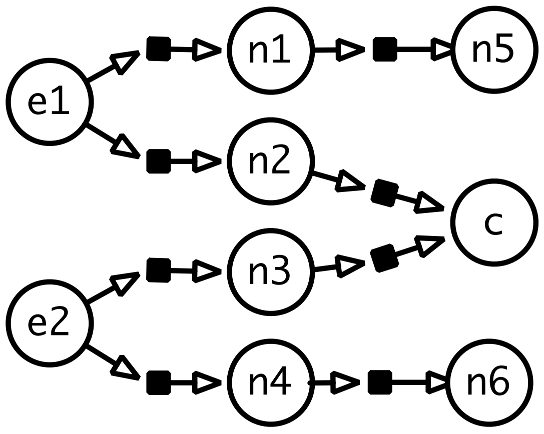
Create a DAG/graph

```
// Create 6 generic nodes - max 1 in channel 1 out
channel
N n1(1),n2(2),n3(3),n4(4),n5(5),n6(6);
n1.create_input_buffer(100); …
n6.create_input_buffer(100);

// 2 emitters: e1->n1, e1->n2, e2->n3, e2->n4
std::vector<ff_node*> we1, we2;
we1.push_back(&n1); we1.push_back(&n2);
we2.push_back(&n3); we2.push_back(&n4);
E e1(we1,1,ntasks), e2(we2,2,ntasks);

// link n1->n5 and n4->n6
n1.set_output(n5.get_in_buffer());
n4.set_output(n6.get_in_buffer());

// 1 collector + linking: n2->c1, n3->c1
n2.create_output_buffer(100);
n3.create_output_buffer(100);
std::vector<ff_node*> wc1;
wc1.push_back(&n2);
wc1.push_back(&n3);
C c1(wc1,1 /* id */);
c1.create_input_buffer(100);

// run all nodes and wait end
n1.run(); … n6.run(); e1.run(); e2.run(); c1.run();
n1.wait(); … n6.wait(); e1.wait(); e2.wait(); c1.run();
```



Create a DAG/graph

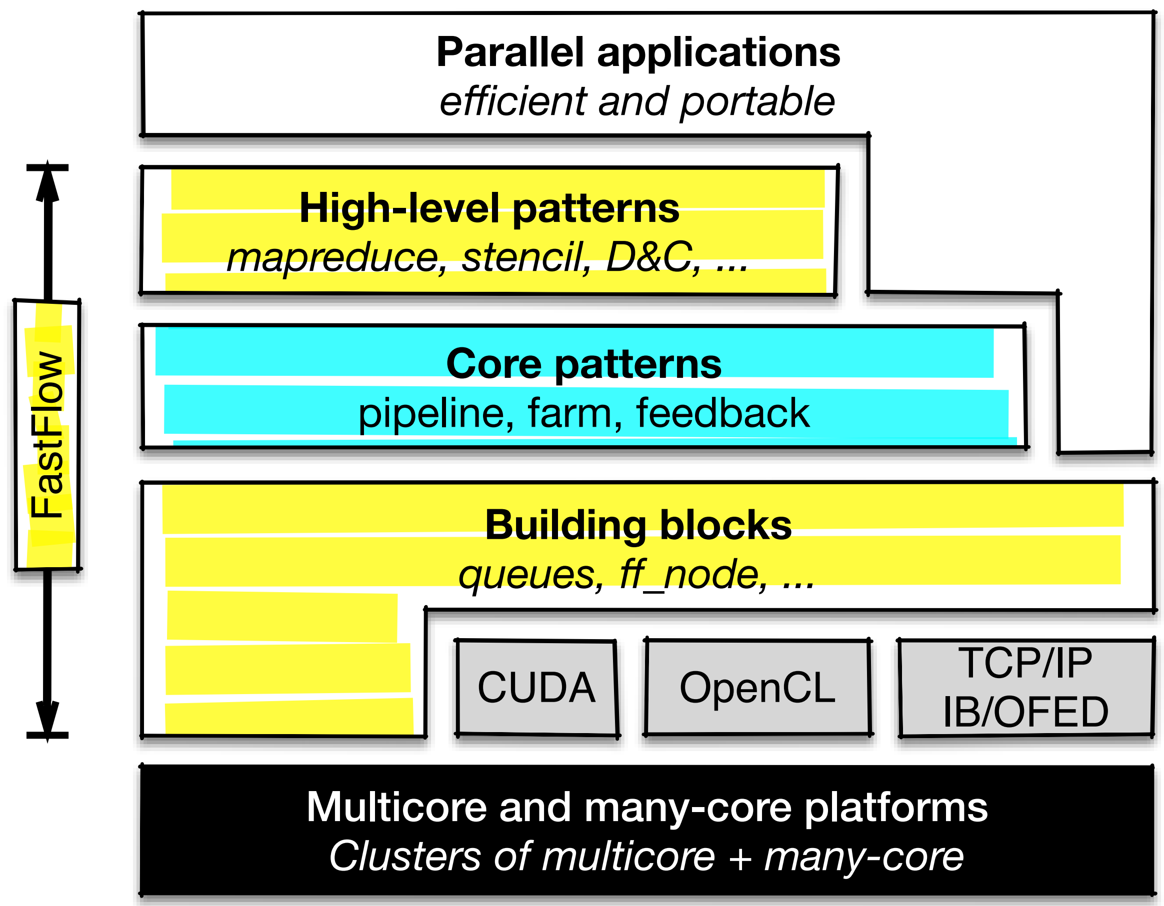
```
// Create 6 generic nodes - max 1 in channel 1 out
channel
N n1(1),n2(2),n3(3),n4(4),n5(5),n6(6);
n1.create_input_buffer(100); …
n6.create_input_buffer(100);

// 2 emitters: e1->n1, e1->n2, e2->n3, e2->n4
std::vector<ff_node*> we1, we2;
we1.push_back(&n1); we1.push_back(&n2);
we2.push_back(&n3); we2.push_back(&n4);
E e1(we1,1,ntasks), e2(we2,2,ntasks);

// link n1->n5 and n4->n6
n1.set_output(n5.get_in_buffer());
n4.set_output(n6.get_in_buffer());

// 1 collector + linking: n2->c1, n3->c1
n2.create_output_buffer(100);
n3.create_output_buffer(100);
std::vector<ff_node*> wc1;
wc1.push_back(&n2);
wc1.push_back(&n3);
C c1(wc1,1 /* id */);
c1.create_input_buffer(100);

// run all nodes and wait end
n1.run(); … n6.run(); e1.run(); e2.run(); c1.run();
n1.wait(); … n6.wait(); e1.wait(); e2.wait(); c1.run();
```



Not task processing

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

Core patterns



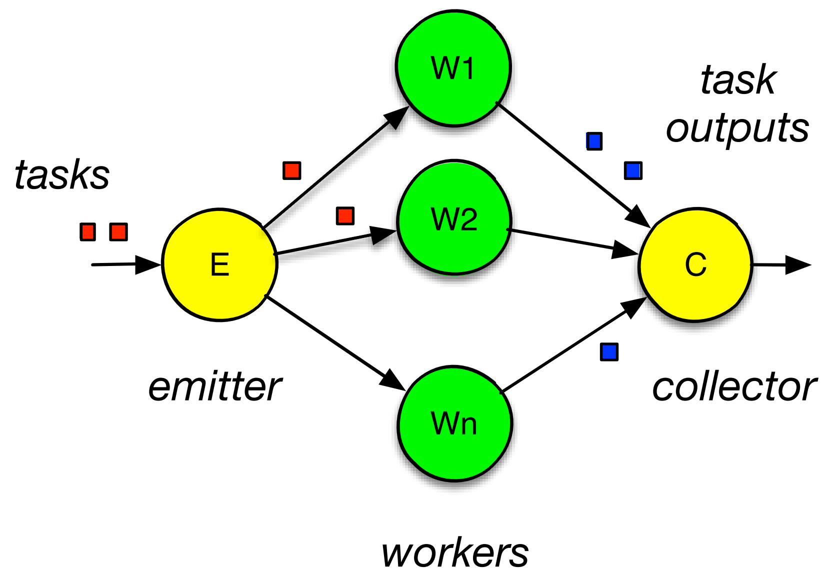
Core patterns: compositional pipeline, farm and feedback qualifier

Enough to build any useful streaming network, i.e. the implementation of all other patterns

For GPGPU: + stenciReduce - evaluation ongoing

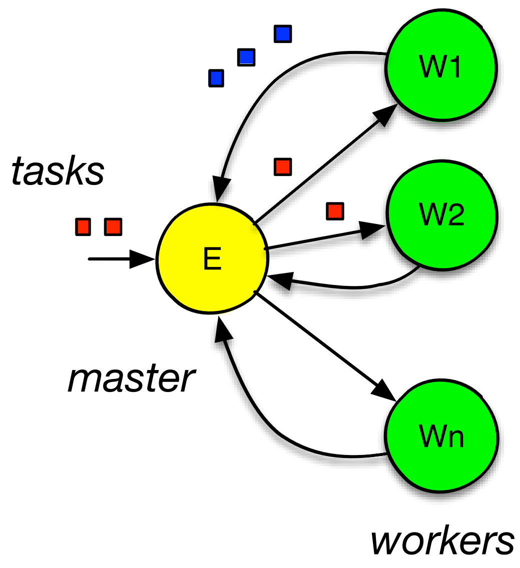
Farm

- Exploit parallelism on a stream (or a set) of independent tasks
  - Master-worker on a stream of tasks
  - No other data dependencies are enforced on tasks beyond task scheduling on workers and task outputs collection
  - They can enforced in the worker business code (e.g. via mutex)
  - *emitter* is a scheduler
  - *collector* is a gatherer
  - all actors are ff_nodes



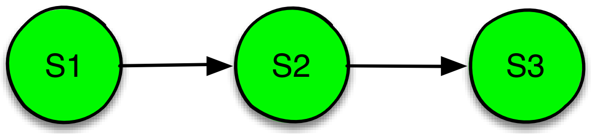
Farm + feedback

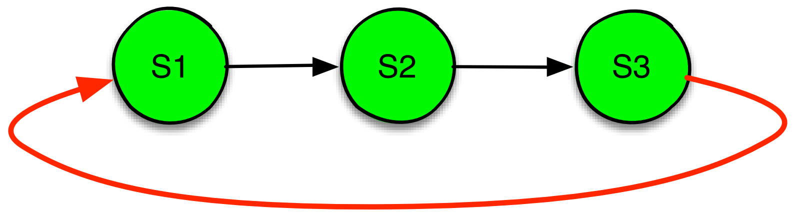
* Master-worker
  * just a variant of farm
  * farm.wrap_around()



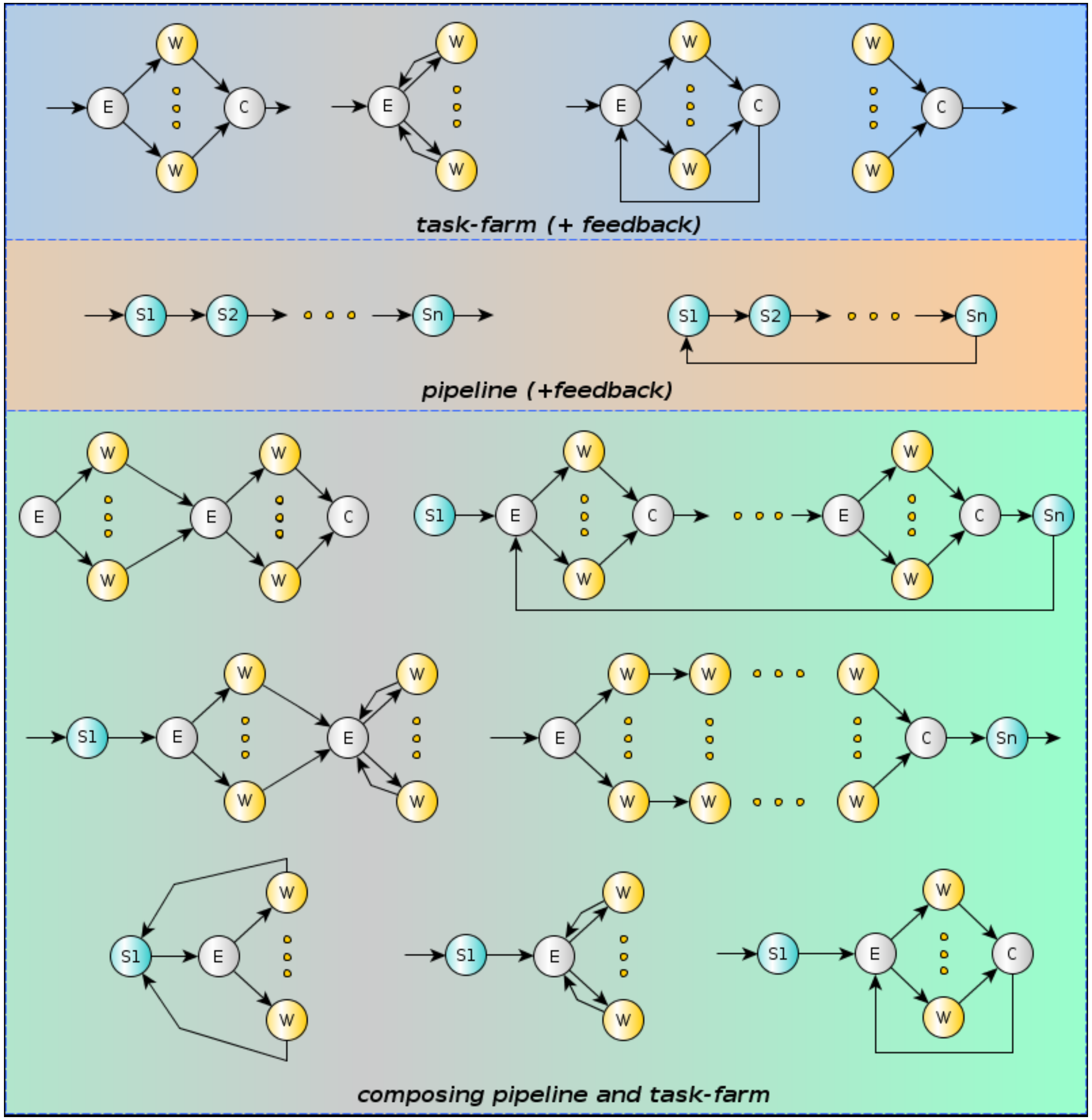
Pipeline

- Pipeline
  - functional composition
- pipeline + wrap_around
  - functional composition wither recursion





Composing them



Much more

* lock-less parallel memory allocator
* scheduling
* pinning
* memory affinity
* passive mediators
* load-balancing
* offloading on soft and hardware accelerators (OpenCL and CUDA)
* distributed

http://mc-fastflow.sourceforge.net/

Code examples

```
/**
 First FastFlow farm.

 clang++ -I ./fastflow test_farm.cpp -o test_farm

Alternative versions (removing some of the commented lines)

 - simple farm
 - -DTRACE_FASTFLOW
 - emit task via ff_send_out
 - without collector
 - initialise workers/nodes via constructor
 - ondemand policy (or user-defined)
*/

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

using namespace ff;

struct ff_task_t {
    ff_task_t(int c): payload(c) {}

    int payload;
};


// generic worker
class Worker: public ff_node {
public:
    // Constructor
    //Worker() {};

    // Service method - to hold the business code - will called each received ta
sk
    void * svc(void * task) {
        ff_task_t * t = (ff_task_t *)task;
        //std::cout << "Worker " << ff_node::get_my_id()
        //          << " received task " << t->payload << "\n";
        //std::fflush(stdout);
        // business code
        t->payload++;
        usleep(500);
        // end business code
        return task;
    }

    // init the worker - executed before first task - optional
    //int   svc_init() { return 0; }

    // finalise the worker - executed after last task - optional
    //void  svc_end() {}

};

// Emitter - no input channel in this case - will start spontaneusly
class Emitter: public ff_node {
public:
    Emitter(int max_task):ntask(max_task) {};


    void * svc(void *) {
        --ntask;
        // on return NULL the node terminate. Termination is propagated to all w
orkers
        if (ntask<0) return NULL;
        ff_task_t * task = new ff_task_t(ntask);
        return task;
```



```
        // alternatively
        //ff_send_out(task); // ff_send_out(worker_id,task);
        //task = new ff_task_t(100+ntask);
        //ff_send_out(task);
        //return GO_ON;
    }
private:
    int ntask;
};

// the gatherer filter
class Collector: public ff_node {
public:
    void * svc(void * task) {
        ff_task_t * t = (ff_task_t *) task;
        std::cout << "Collector received " << t->payload << "\n";
        //std::fflush(stdout);
        delete t;
        return GO_ON;
    }
};

int main(int argc, char * argv[]) {
    int nworkers = 3;
    int streamlen = 1000;

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

    //farm.set_scheduling_ondemand(4);

    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;
}
```

```
/**
FastFlow farm accelerator.

 clang++ -I ./fastflow test_farm.cpp -DTRACE_FASTFLOW -o test_farm

 - load results NB and B
 - freezing
*/

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

using namespace ff;

struct ff_task_t {
    ff_task_t(int c): payload(c) {}

    int payload;
};


// generic worker
class Worker: public ff_node {
public:
    // Constructor
    //Worker() {};

    // Service method - to hold the business code - will called each received ta
sk
    void * svc(void * task) {
        ff_task_t * t = (ff_task_t *)task;
        //std::cout << "Worker " << ff_node::get_my_id()
        //          << " received task " << t->payload << "\n";
        //std::fflush(stdout);
        // business code
        t->payload++;
        usleep(random() % 1000);
        // end business code
        return task;
    }

    // init the worker - executed before first task - optional
    //int   svc_init() { return 0; }

    // finalise the worker - executed after last task - optional
    //void  svc_end() {}

};

int main(int argc, char * argv[]) {
    int nworkers = 3;
    int streamlen = 1000;

    if (argc>1) {
        if (argc!=3) {
            std::cerr << "use: "
                      << argv[0]
                      << " nworkers streamlen\n";
            return -1;
        }
        nworkers=atoi(argv[1]);
        streamlen=atoi(argv[2]);
    }

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



```
    ff_farm<> farm(true /* acc mode */); // farm object

    // using standard emitter
    //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

    // and standard collector
    //Collector C;
    farm.add_collector(NULL);

    farm.run();


    //for (int j=0;j<4;++j) {
    //farm.run_then_freeze();
    void *result;
    for (int i=0;i<streamlen;i++) {
        ff_task_t * ii = new ff_task_t(i);
        std::cout << "[Main] Offloading " << i << "\n";
        // Here offloading computation onto the farm
        farm.offload(ii);

        std::cout << "[Main] Do something else\n";
        usleep(100); // change me to check nonblocking result get

        // possibly get results (nonblocking)
        if (farm.load_result_nb(&result)) {
            std::cerr << "[Main] NB result= " << ((ff_task_t*)result)->payload << "\n
";
            delete ((ff_task_t*)result);
        }
    }
    std::cout << "[Main] EOS arrived\n";
    farm.offload((void *)FF_EOS);

    // get all remaining results syncronously.
    while(farm.load_result(&result)) {
        std::cerr << "[Main] B result= " << ((ff_task_t*)result)->payload << "\n";
        delete ((ff_task_t*)result);
    }
    farm.wait();
    //farm.wait_freezing();
    std::cout << "[Main] Farm accelerator stopped\n";
    //}
    std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
    farm.ffStats(std::cerr);

    return 0;
}
```

```
/*
  Very basic test for the FastFlow pipeline (actually a 2-stage torus).

  clang++ -I ./fastflow test_pipe.cpp  -o test_pipe
  clang++ -I ./fastflow test_pipe.cpp -DTRACE_FASTFLOW -o test_pipe

 */

#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

struct ff_task_t {
    ff_task_t() {}
    int payload;
};

// generic stage
class Stage: public ff_node {
public:
    Stage(unsigned int streamlen):streamlen(streamlen),sum(0)  {}

    void * svc(void * task) {
        unsigned int * t = (unsigned int *)task;

        if (!t) {
            t = (unsigned int*)malloc(sizeof(int));
            if (!t) abort();

            *t=0;
            task = t;
        } else { sum+=*t; *t+=1;}

        if (*t == streamlen) return NULL;
        task = t;
        return task;
    }
    void  svc_end() {
        if (ff_node::get_my_id())
            std::cout << "Sum: " << sum << "\n";
    }

private:
    unsigned int streamlen;
    unsigned int sum;
};

int main(int argc, char * argv[]) {
    int streamlen = 1000;
    // bild a 2-stage pipeline
    ff_pipeline pipe;
    pipe.add_stage(new Stage(streamlen));
    pipe.add_stage(new Stage(streamlen));

    pipe.wrap_around();

    ffTime(START_TIME);
    if (pipe.run_and_wait_end()<0) {
        error("running pipeline\n");
        return -1;
    }
    ffTime(STOP_TIME);

    std::cerr << "DONE, pipe  time= " << pipe.ffTime() << " (ms)\n";
    std::cerr << "DONE, total time= " << ffTime(GET_TIME) << " (ms)\n";
    pipe.ffStats(std::cerr);
    return 0;
}
```

```
/*
  farm with user0defined affinity scheduling policy

 */

#include <ff/farm.hpp>

using namespace ff;

#define NWORKERS 4
#define INBUF_Q_SIZE 4

//#define MNODES 2
#define CORES 4

typedef struct ff_task {
    int sourceW;
    int mnode;
        int core;
        int done;
} ff_task_t;

class emitter: public ff_node {
public:
        emitter(ff_loadbalancer* const lb, const svector<ff_node*> &workers):
        lb(lb),workers(workers) {
                completed = new bool[NWORKERS];
                for (int i = 0; i < NWORKERS; i++)
                        completed[i] = false;
        done=0;
        }

        void* svc(void *t) {
                ff_task_t* worker_task = (ff_task_t*) t;
                if (worker_task == NULL) {
            for (int j = 0; j < INBUF_Q_SIZE; j++) {
                for (int i = 0; i < NWORKERS; i++) {
                    int targetworker = workers[i]->get_my_id();
                    int targetcore = threadMapper::instance()->getCoreId(lb->get
Tid(workers[i]));
                    int targetmnode = targetcore/8;
                    // here allocate the task
                    // numa_malloc(sizeof(ff_task_t),mnode)
                    worker_task = (ff_task_t *) malloc(sizeof(ff_task_t));
                    worker_task->sourceW = targetworker;
                    // here danger of race condition
                    // only static information can be used
                    worker_task->core = targetcore;
                    worker_task->mnode = targetmnode;
                    worker_task->done = 0;
                    bool res = lb->ff_send_out_to(worker_task, targetworker);
                    if (res)
                        printf("sent to worker %d on core %d mnode %d\n",
                               worker_task->sourceW,
                               worker_task->core,
                               worker_task->mnode);
                    else printf("ERROR: send failed − should never happen − task is lost − queue are
too short\n");
                }
            }
                } else {
            if (worker_task->done<10) {

                printf("[E] recv from worker %d on core %d mnode %d\n",
                       worker_task->sourceW, worker_task->core,
                       worker_task->mnode);
                lb->ff_send_out_to(worker_task, worker_task->sourceW);
            } else {
                completed[worker_task->sourceW] = true;
```



```
            }
                }

                for (int i = 0; i < NWORKERS; i++)
            done &= completed[i];
        if (done) {
                free(worker_task);
                delete[] completed;
                return NULL;
            }
        else done = true;
        return GO_ON;
        }

private:
    ff_loadbalancer* lb;
    const svector<ff_node*> &workers;
    bool* completed;
    bool done;
};

class worker: public ff_node {
public:
    worker (): taskcount(0){};
        void* svc(void* t) {
                ff_task_t* worker_task = (ff_task_t*) t;
        ++worker_task->done;
                usleep(worker_task->sourceW * 100);
                printf("[%d] received from emitter task workerid %d on core %d\n",
                       get_my_id(), worker_task->sourceW, worker_task->core);

        ++taskcount;
        return (worker_task);
        }
    void svc_end (){
        printf("[%d] processed %d tasks\n",get_my_id(),taskcount);
    }
private:
    int taskcount;
};

int main() {
        std::vector<ff_node *> workers;
        ff_farm<> farm(false);
        for (int i = 0; i < NWORKERS; i++)
                workers.push_back(new worker());
        emitter em(farm.getlb(), farm.getWorkers());
        farm.add_emitter(&em);
        farm.add_workers(workers);
        farm.set_scheduling_ondemand(INBUF_Q_SIZE);
        farm.wrap_around();
        if (farm.run_and_wait_end() < 0) {
                error("running farm\n");
                return -1;
        }
        return 0;
}
```

```
/*
 * Mixing FastFlow pipeline and farm. The farm module has neihter the Emitter
 * nor the Collector filters.
 *
 *      |---------- 3-stage pipeline -----------|
 *      |                   |                   |
 *      |                   v                   |
 *      |            |(stage2_1->stage2_2)|     |
 *      v            |                    |     v
 *     stage1-->farm |(stage2_1->stage2_2)|-->stage3
 *                   |                    |
 *                   |(stage2_1->stage2_2)|
 *                     ^                ^
 *                     |- 2-stage pipe -|
 */

#include <iostream>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <ff/allocator.hpp>

using namespace ff;

static ff_allocator ffalloc;

class Stage1: public ff_node {
public:
    Stage1(unsigned int streamlen):streamlen(streamlen),cnt(0){}

    void * svc(void *) {
        int * t;
        t = (int*)ffalloc.malloc(sizeof(int));
        if (!t) abort();

        *t=cnt++;
        if (cnt > streamlen) {
            ffalloc.free(t);
            t = NULL; // EOS
        }
        return t;
    }

    int   svc_init() {
        if (ffalloc.registerAllocator()<0) {
            error("registerAllocator fails\n");
            return -1;
        }
        return 0;
    }

private:
    unsigned int streamlen;
    unsigned int cnt;
};

class Stage2_1: public ff_node {
public:
    void * svc(void * task) {
        return task;
    }
};

class Stage2_2: public ff_node {
public:
    void * svc(void * task) {
        return task;
    }
};
```



```
class Stage3: public ff_node {
public:
    Stage3():sum(0){}

    void * svc(void * task) {
        int * t = (int *)task;
        if (!t)  abort();
        sum +=*t;
        ffalloc.free(task);
        task = GO_ON; // we want to be sure to continue
        return task;
    }

    int   svc_init() {
        if (ffalloc.register4free()<0) {
            error("register4free fails\n");
            return -1;
        }

        return 0;
    }
    void  svc_end() {
        std::cout << "Sum: " << sum << "\n";
    }

private:
    unsigned int sum;
};

int main(int argc, char * argv[]) {
    int nworkers=3;
    int streamlen=1000;

    if (argc>1) {
        if (argc!=3) {
            std::cerr << "use: "  << argv[0] << " streamlen num-farm-workers\n";
            return -1;
        }
        streamlen=atoi(argv[1]);
        nworkers=atoi(argv[2]);
    }

    ffalloc.init(); // Init allocator ...

    ff_pipeline pipe;
    pipe.add_stage(new Stage1(streamlen));
    ff_farm<> farm; // build farm without Collector
    farm.add_collector(NULL); // standard collector

    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i) {
        ff_pipeline * pipe2 = new ff_pipeline;
        pipe2->add_stage(new Stage2_1);
        pipe2->add_stage(new Stage2_2);
        w.push_back(pipe2);
    }
    farm.add_workers(w);

    pipe.add_stage(&farm);

    pipe.add_stage(new Stage3);
    std::cerr << "Starting ...\n";
    if (pipe.run_and_wait_end()<0) {
        error("running pipeline\n");
        return -1;
    }
    pipe.ffStats(std::cout);
    return 0;
}
```