EuroPar 2011
Bordeaux - France
1st Sept 2001

Accelerating code on multi-cores with FastFlow

Marco Aldinucci
Computer Science Dept. - University of Torino (Turin) - Italy
Massimo Torquati and Marco Danelutto
Computer Science Dept. - University of Pisa - Italy
Massimiliano Meneghin
IBM Research, Ireland
Peter Kilpatrick
Queen's University Belfast, U.K.

Outline

- Porting of existing sequential codes onto multi-core
  - a motivational example (edge-preserving denoiser)
- Offloading and FastFlow accelerators
  - methodology and programming framework supporting fine-grain parallel codes
- FastFlow programming model
  - Design and Implementation
  - Experimental evaluation
- Demo & Conclusion

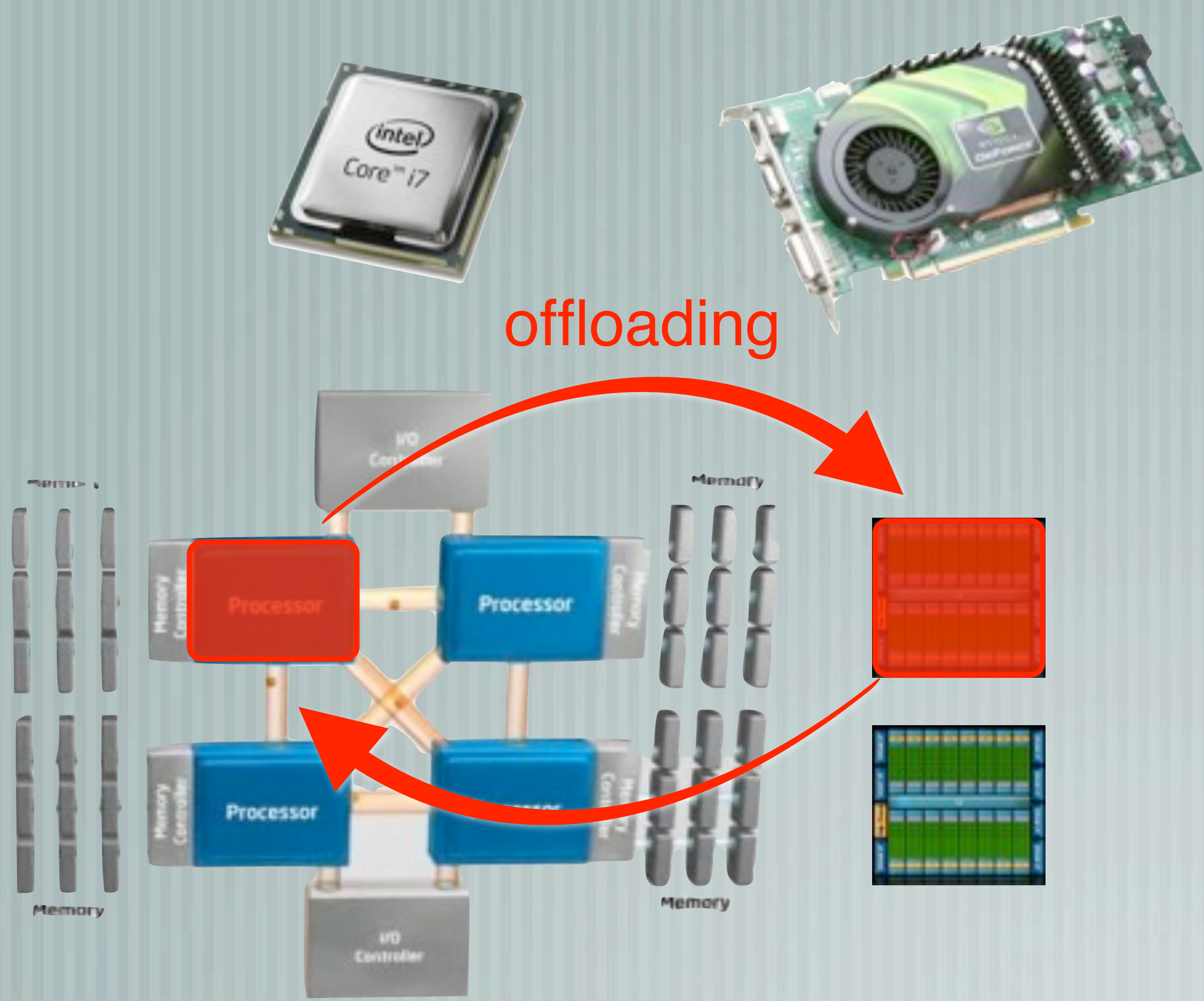
Running example: edge-preserving denoiser

```
#include <opencv/highgui.h>
#include <opencv/cv.h>

int main(int argc, char *argv[]) {
  CvCapture *capture;
  IplImage * frame,clean_frame;
  char key;
  vector<noisy_t> noisy;
  cvNamedWindow("Video", CV_WINDOW_AUTOSIZE);
  capture = cvCreateCameraCapture(CV_CAP_ANY);
  //capture = cvCreateFileCapture("/path/to/your/video/test.avi");
  while(true) {
    frame = cvQueryFrame(capture);        // get a frame from device
    noisy = myDetect(frame);              // detect noisy pixels
    clean_frame = myDenoise(frame,noisy); // denoise the frame
    cvShowImage( "Video", clean_frame);   // show the denoised frame
    key = cvWaitKey(100);
  }
  cvReleaseCapture(&capture);
  cvDestroyWindow("Video");
}
```

Application performance

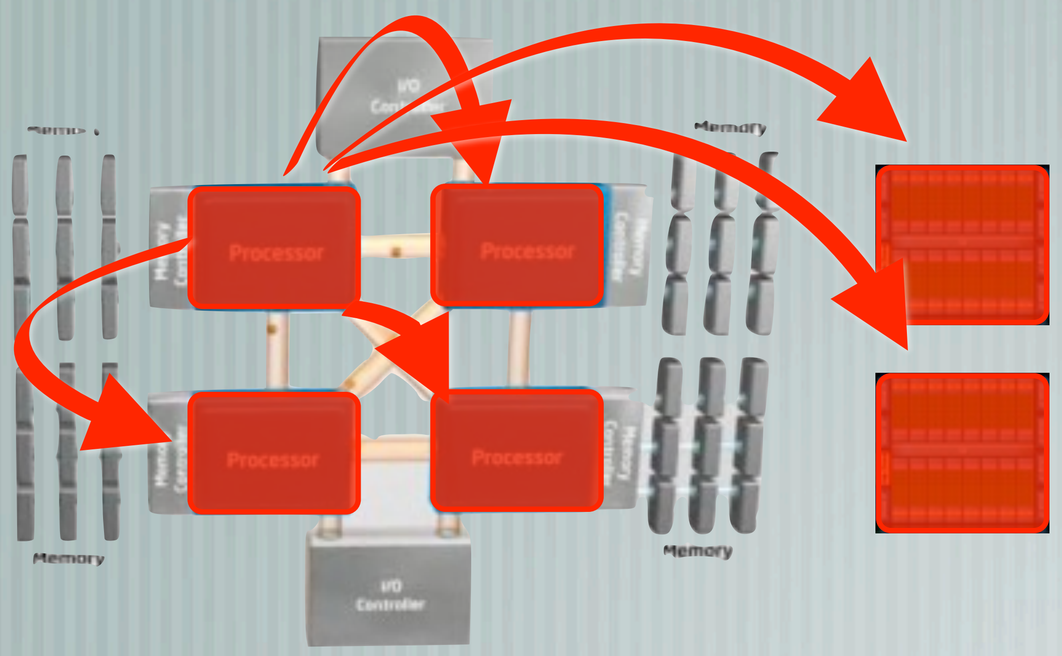
- myDetect and myDenoise filters are slow and exhibits a different computing time
  - Detect is O(size), Denoise is O(size*noise%*image_complexity)
- **in principle** the app can be parallelised exploiting
  - data parallelism on filters
  - pipeline/stream parallelism among capture, the two filters, and display
- **in practice** Intel's opencv is not fully thread-safe (signals)
  - capture and display should be run in the main thread
  - e.g. embedding the app in a TBB not easy (highgui cannot be used), the manual parallelisation via pthreads crash, ...

Easy parallelisation of existing codes



- offloading
  - onto HW accelerator

Easy parallelisation of existing codes



- offloading
  - onto other cores and accelerators

Parallelisation via offloading onto structured software accelerator

Accelerator & self-offloading

- Target the parallelisation of legacy code
  - No need to redesign the application; local intervention in the code
- Transform loops and D&C in streaming then offload them into dynamically created accelerators using spare cores
- Parallelising requires management of data dependency
  - True dependency: fast synchronisation via FastFlow
  - False dependency (Write-Read, Write-Write): remove
    - Variable streamization (i.e. dynamic privatisation onto a stream). More powerful than expansion (do-across

Original

```
...
for (i=0; i<N;++i) {
  temp=A[i]+2;
  B[i]=2*temp;
}
```

Privatization

```
...
for (i=0;i<N;++i) {
  private temp=A[i]+2;
  B[i]=2*temp;
}
```

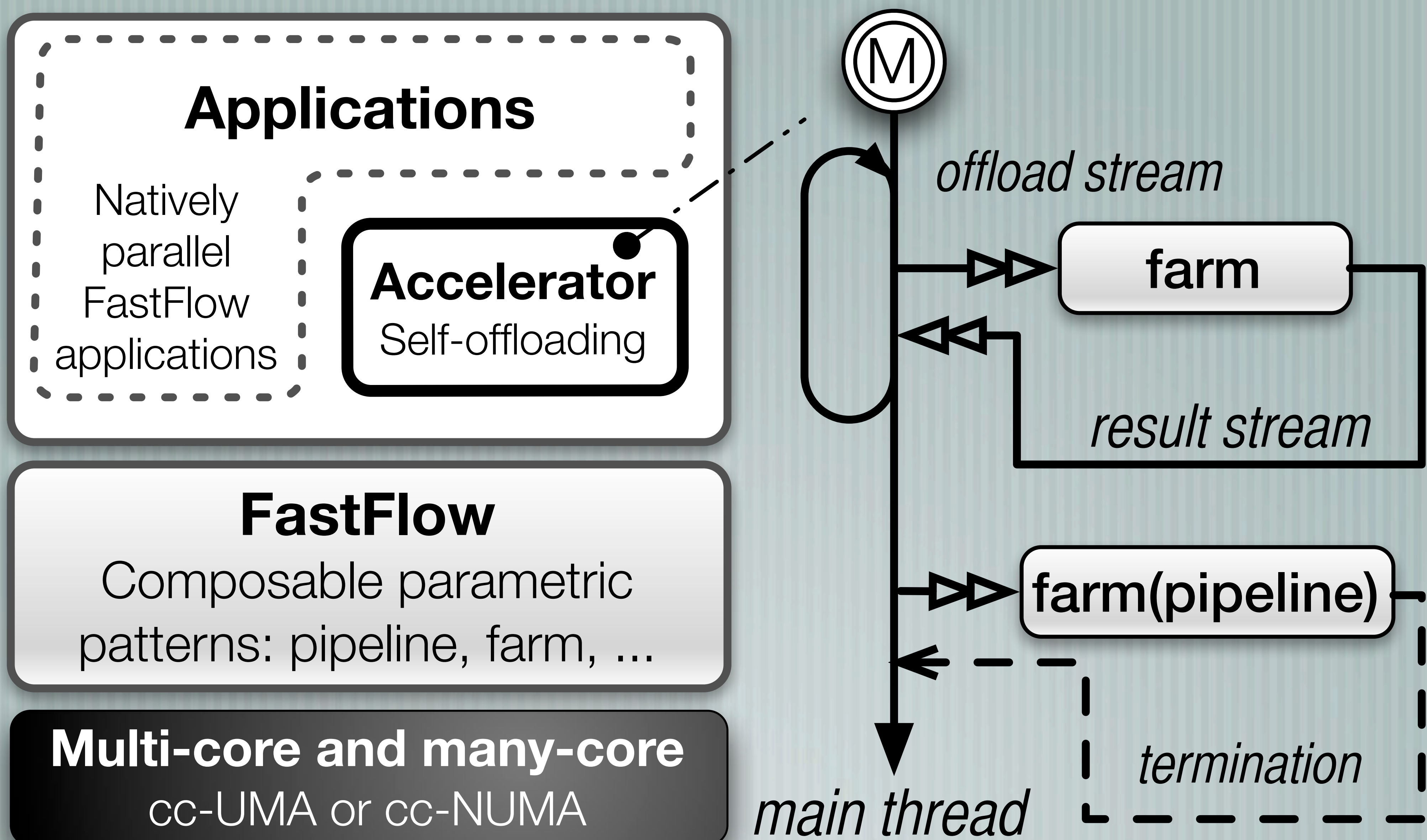
Expansion

```
...
for (i=0;i<N;++i) {
  temp[i]=A[i]+2;
  B[i]=2*temp[i];
}
```

Accelerator & self-offloading

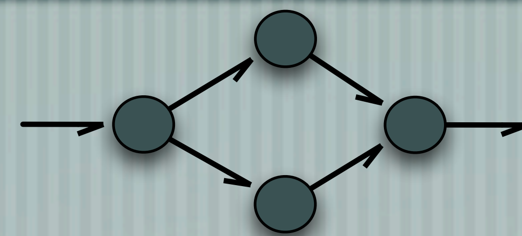
- FastFlow-based stream-oriented parallel accelerators
- Source of stream parallelism
  - Natural
    - external devices, network: antenna, audio/video, medical instruments, ...
  - Produced via streamization from recursion/iteration
    - streamization of recursion
    - streamization of loops without dependencies (do-independent)
    - streamization of loops with dependencies (do-across)

Accelerator architecture

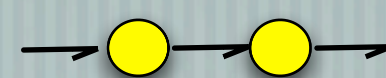


Accelerator patterns and their composition

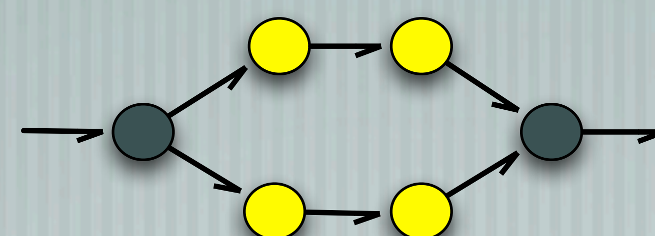
- farm



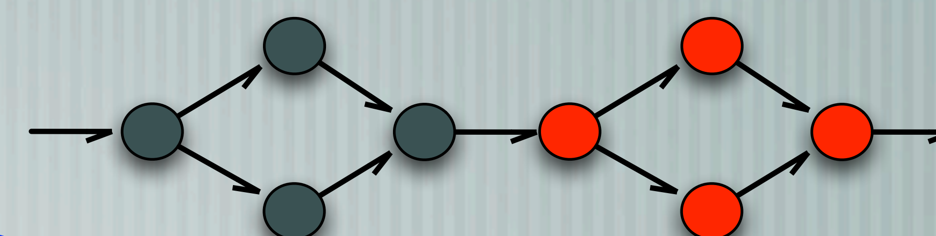
- pipe



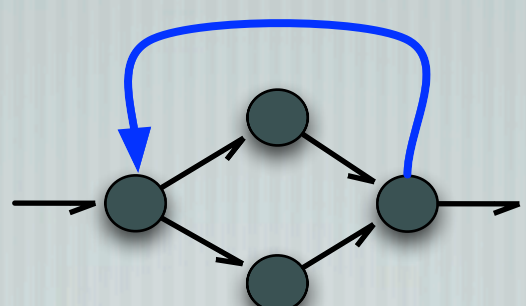
- farm{ pipe }



- farm ; farm

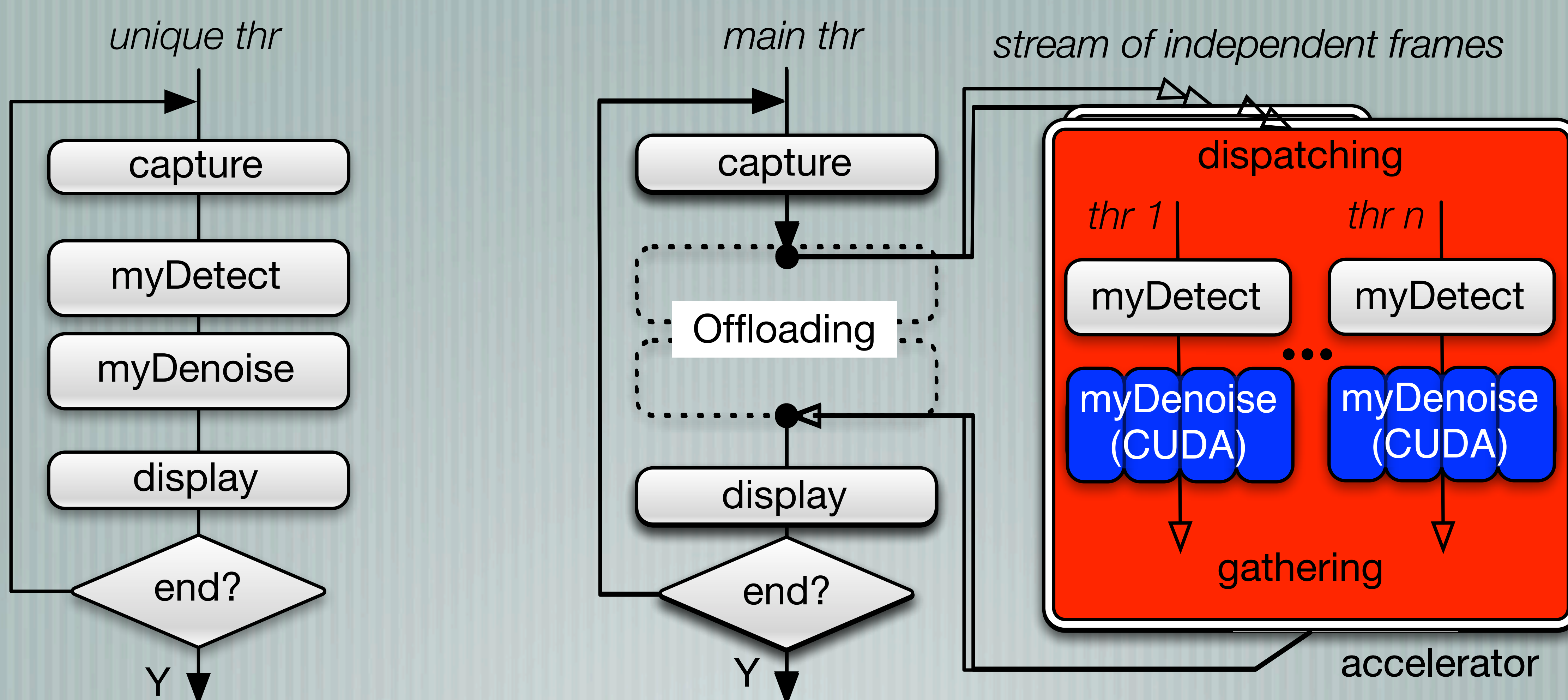


- D&C = farm + wrap



- any variation of them requiring additional synch ...

Running example: denoiser variants



Sequential

Changing the accelerator structure does not require re-writing business code (gray)

Self-offloading methodology

```
// Original code
#define N 1024
long A[N][N],B[N][N],C[N][N];
int main() {
  // < init A,B,C>

  for(int i=0;i<N;++i) {
    for(int j=0;j<N;++j) {

      int _C=0;
      for(int k=0;k<N;++k)
        _C += A[i][k]*B[k][j];
      C[i][j]=_C;

    }
  }
}
```

Original

```
// FastFlow accelerated code
#define N 1024
long A[N][N],B[N][N],C[N][N];
int main() {
  // < init A,B,C>

  ff::ff_farm<> farm(true /* accel */);
  std::vector<ff::ff_node *> w;
  for(int i=0;i<PAR_DEGREE;++i)
    w.push_back(new Worker);
  farm.add_workers(w);
  farm.run_then_freeze();

  for (int i=0;i<N;i++) {
    for(int j=0;j<N;++j) {
      task_t * task = new task_t(i,j);
      farm.offload(task);
    }
  }
  farm.offload((void *)ff::FF_EOS);
  farm.wait();  // Here join
}

// Includes
struct task_t {
  task_t(int i,int j):i(i),j(j) {}
  int i; int j;};

class Worker: public ff::ff_node {
public: // Offload target service
  void * svc(void *task) {
    task_t * t = (task_t *)task;
    int _C=0;
    for(int k=0;k<N;++k)
      _C += A[t->i][k]*B[k][t->j];
    C[t->i][t->j] = _C;
    delete t;
    return GO_ON;
  }
};
```

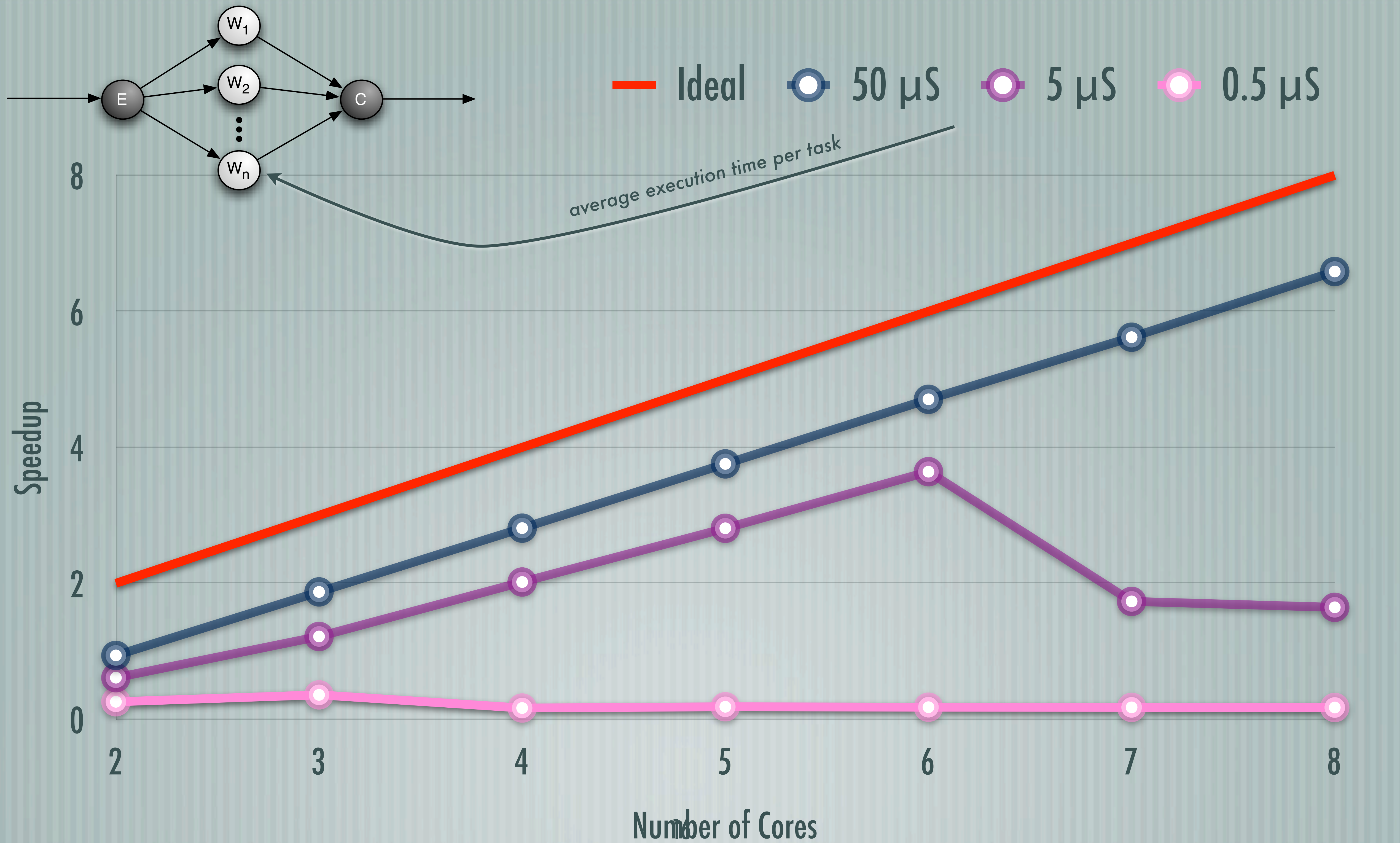
Accelerated

FastFlow as a programming model for multi-core

ISSUES FOR EXASCALE ERA FROM P. BECKMAN KEYNOTE

- “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”

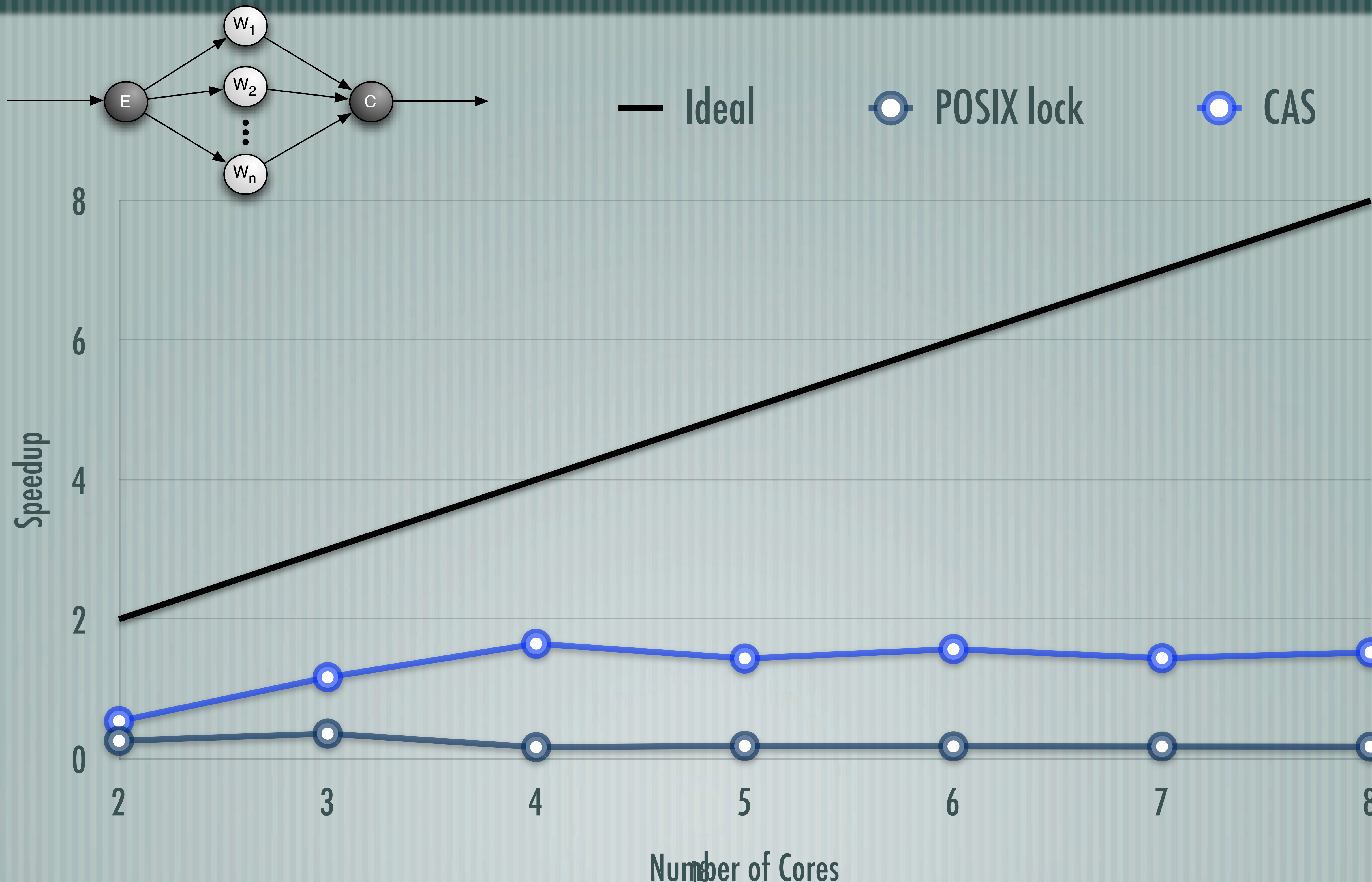
Grain: task farm with POSIX lock/unlock



Can we avoid locks?

- Under relaxed memory models, using CAS/atomic ops
  - lock-free data structures
  - they perform better than lock-based
  - they fence the memory and pay cache coherency reconciliation overhead

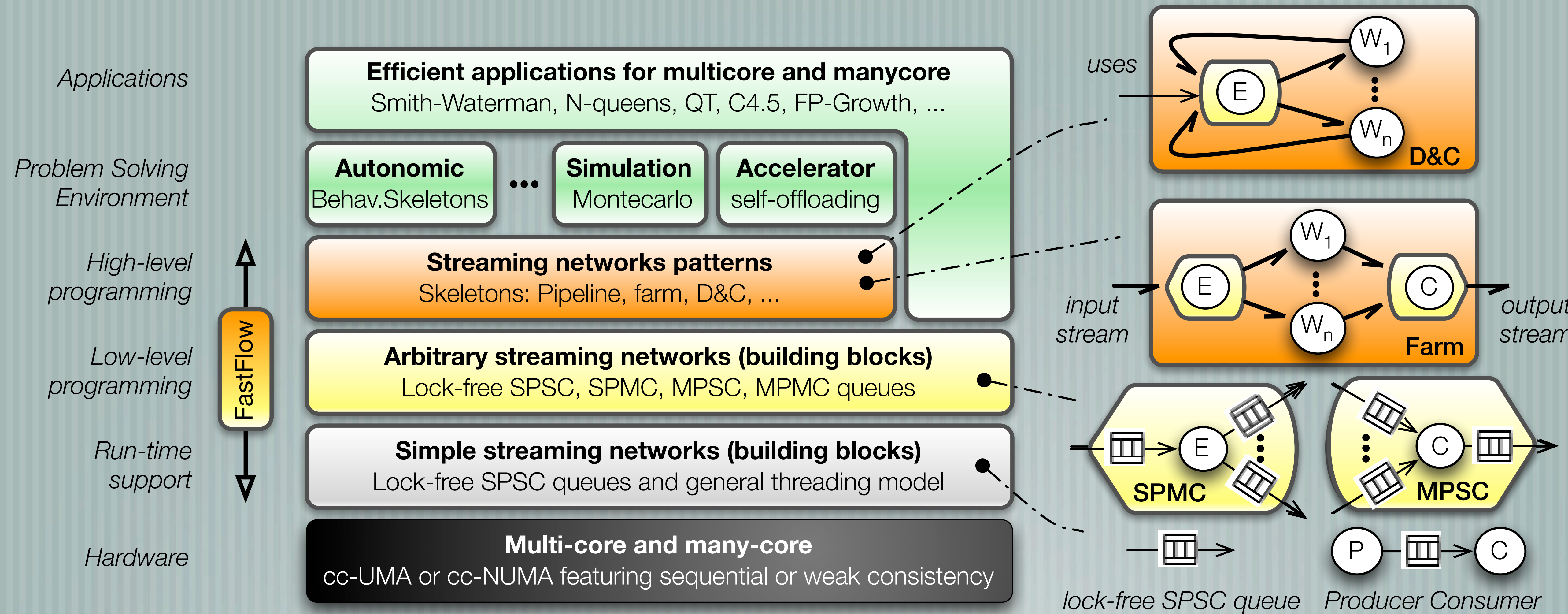
Lock vs CAS at fine grain (0.5 μS)



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

FastFlow is based on producer-consumer



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

FastFlow 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;
}
```

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

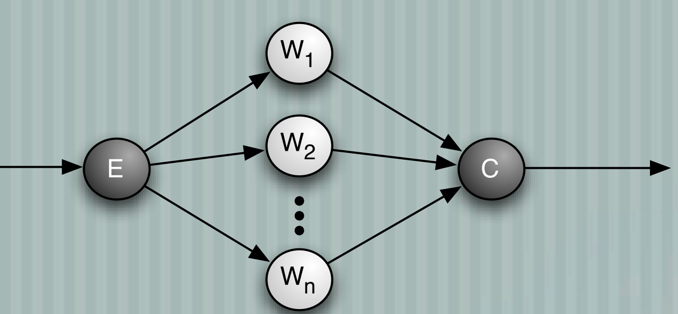
Lamport FIFO
1983

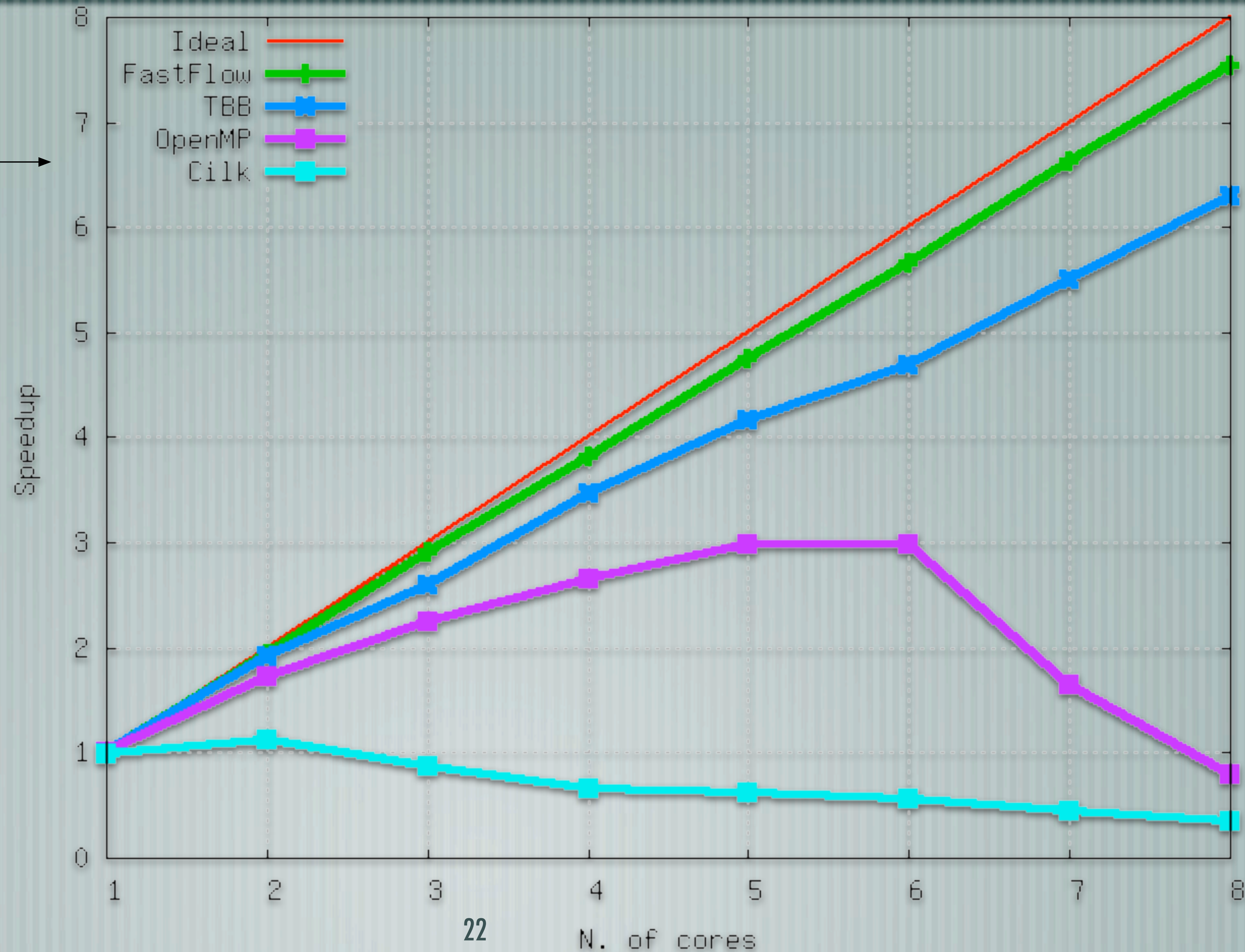
```
push_nonbocking(data) {
  if (NULL != buffer[head]) {
    return EWOULDBLOCK;
  }                                    (WMB)
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

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

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

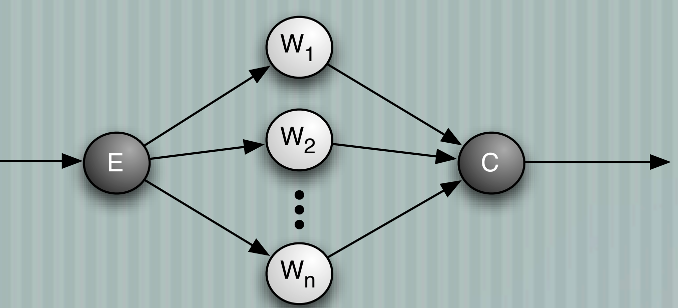
FastFlow FIFO
(derived from P1C1)

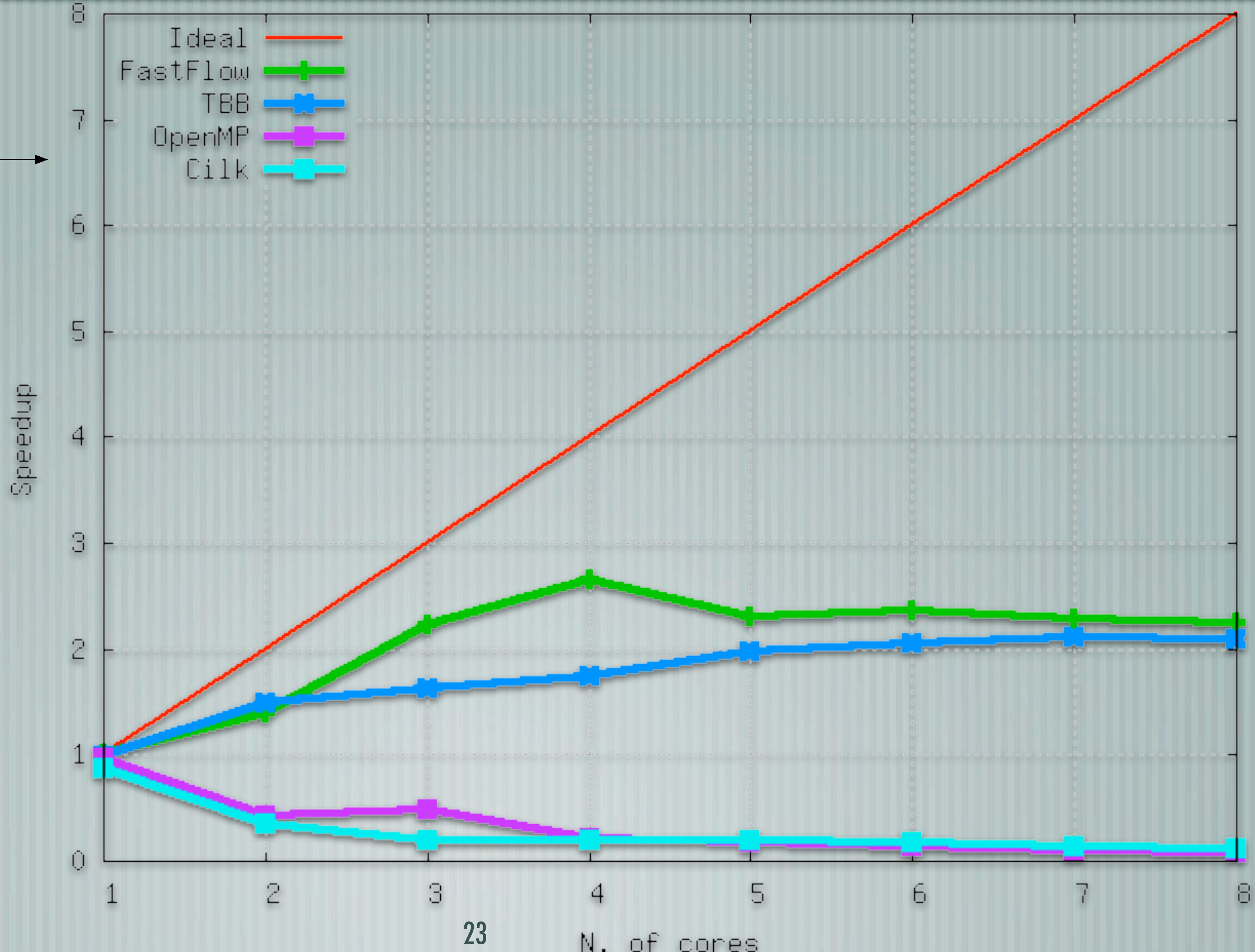
Medium grain (5 μS workload)



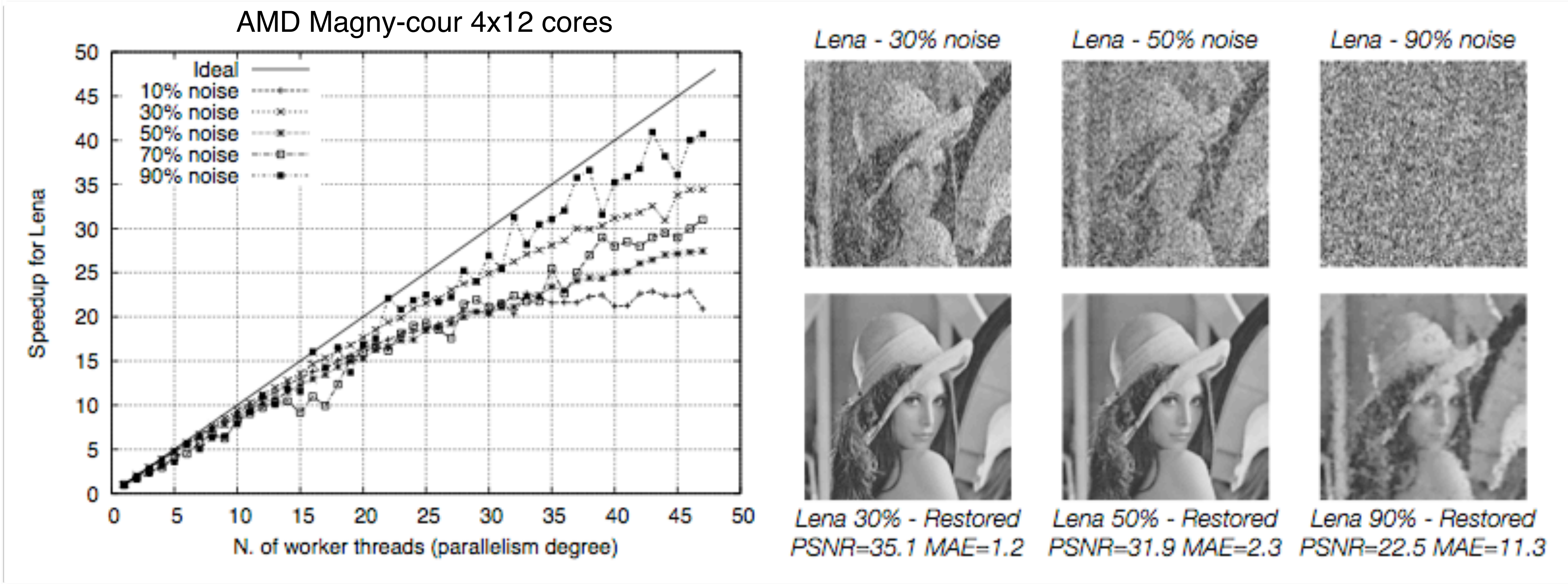


Fine grain (0.5 μS workload)





Good scalability (see the paper)



Lena* with 90% of noise is restored in 4 seconds
Next best result in literature is about 180 seconds

Lena Lombardy. Standard test image, Playboy magazine centerfold (top). Nov. 1972

BioBITs

ISSUES FOR EXASCALE ERA FROM P. BECKMAN KEYNOTE

- “Coarse grain concurrency is nearly exhausted”
  - more scalable than OpenMP, TBB, Cilk at fine grain
- “It is not about Flops, it is about data movement”
  - inter-core communication latency ~7 ns on core2 2Ghz
- “Programming systems should be designed to support fast data movement and enforce locality”
  - FastFlow memory allocator is specifically designed to enhance locality in streaming applications.
- “Variable coherency & inter-socket messaging”
  - SPSC queue is designed to selectively disarm coherency. Queues can be used to both pass pointers (zero-copy) o to copy messages in a message passing fashion.

ISSUES FOR EXASCALE ERA FROM P. BECKMAN KEYNOTE

- “A computer language is not a computing model.”
  “A library is not a computing model.”
  “System programmers should use the techniques they advocate”
  - Data communication happen via both shared-memory and messages. Synchronisations are realised via message-passing (FIFO queues).
  - Synchronisation are local (no barriers) and determined by high-level algorithmic patterns. Data races are identified and solved at design time.
  - FastFlow memory allocator has been developed in FastFlow

Running example demo

Thank you! Questions?

HPC
ADVISORY COUNCIL
NETWORK OF EXPERTISE

- FastFlow: an open source project
  - http://mc-fastflow.sourceforge.net
  - Many contributes from the open source community worldwide
  - Over 25K website visits, 6K downloads form 120 different countries in 1 year and half
  - HPC advisoryboard academic award 2011 (announced at Intl. Supercomputing 2011)
  - ParaPhrase STREP (FWP7 - starting Oct 2011, 3 years)
  - tested on Linux, Mac, Windows XP/7 - Intel core2, AMD Opteron, PPC, ARM (ongoing)
- Many existing benchmarks and applications
  - C4.5, k-means, pbzip-ff, smith-waterman, Stochkit-ff, Parallel MonteCarlo, N-queens ...
  - Many on my laptop, just ask if you are interested

Related Work: Lock-free and CAS-free

- Single-Producer-Single-Consumer FIFO queues
  - Lamport et al. 1983 Trans. PLS (Sequential consistency only - passive)
  - Higham and Kavalsh. 1997 ISPAN (P1C1 - TSO + proof - passive)
  - Giacomoni et al. 2008 PPoPP (TSO + cache slipping - passive)
- Multiple-Producers-Multiple-Consumers FIFO queues
  - with CAS (two of them) - Michael and Scott (PODC96)
    - Also implemented in FastFlow, require deferred reclamation/hazard pointers to avoid ABA problem
  - without CAS - passive ➠ Cannot be done
  - without CAS - active ➠ FastFlow
- Extending the taxonomy with locking algorithms is clearly useless ....