An improvement of OpenMP pipeline parallelism with the BatchQueue algorithm

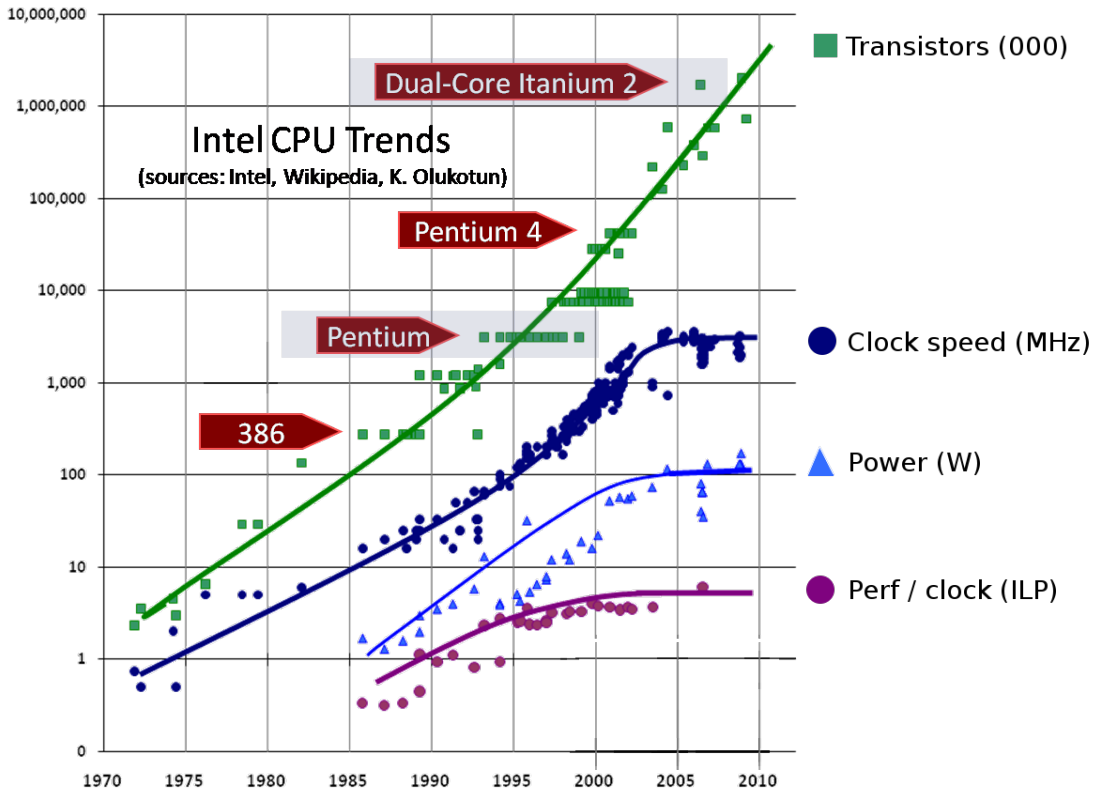
Thomas Preud'homme

Team REGAL
Advisors: Julien Sopena et Gaël Thomas
Supervisor: Bertil Folliot

June 10, 2013

Moore's law in modern CPU

Moore's law: Number of transistors on chips doubles every 2 years



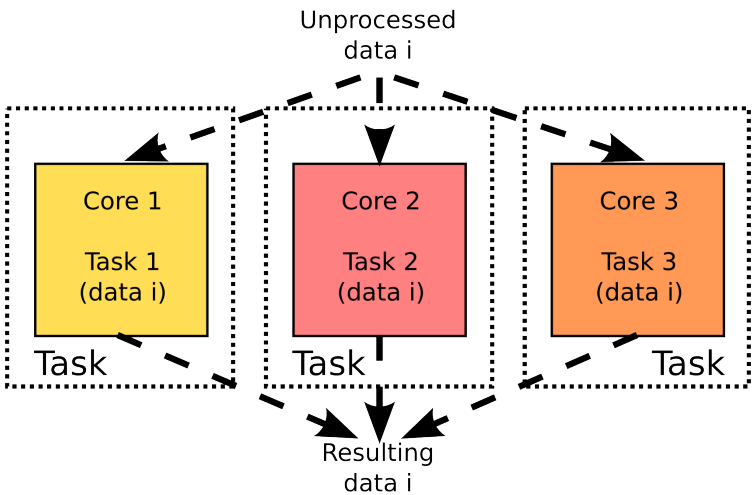
Now: CPU frequency stagnate, number of cores increases

⇒ parallelism is needed to take advantage of multi-core systems

Classical paradigms of parallel programming

Several paradigms of parallel programming already exist:

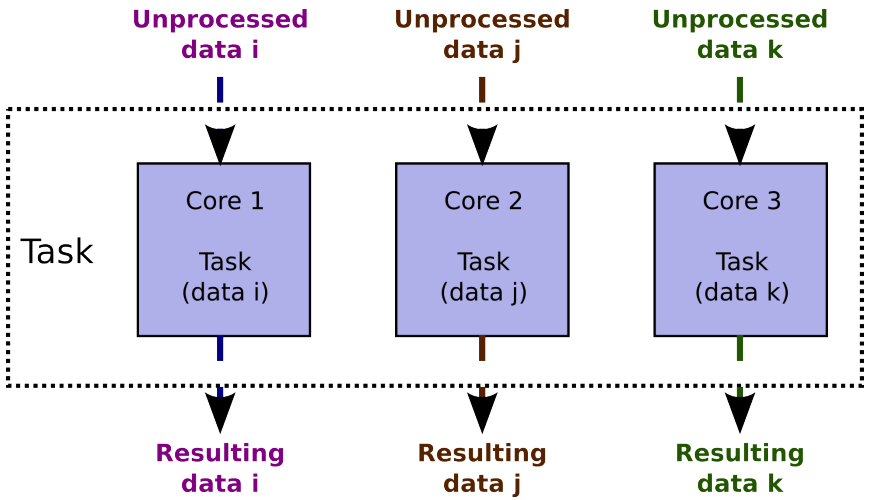
Task parallelism



E.g.: multitasking

Limit: needs **independent** tasks

Data parallelism



E.g.: array/matrix processing

Limit: needs **independent** data

Task and data dependencies: video edition example

Some modern applications require complex computation but cannot use task or data parallelism due to dependencies.
⇒ eg. audio and video processing

Example of video edition:

1. decode a frame into a bitmap image
2. rotate the image
3. trim the image

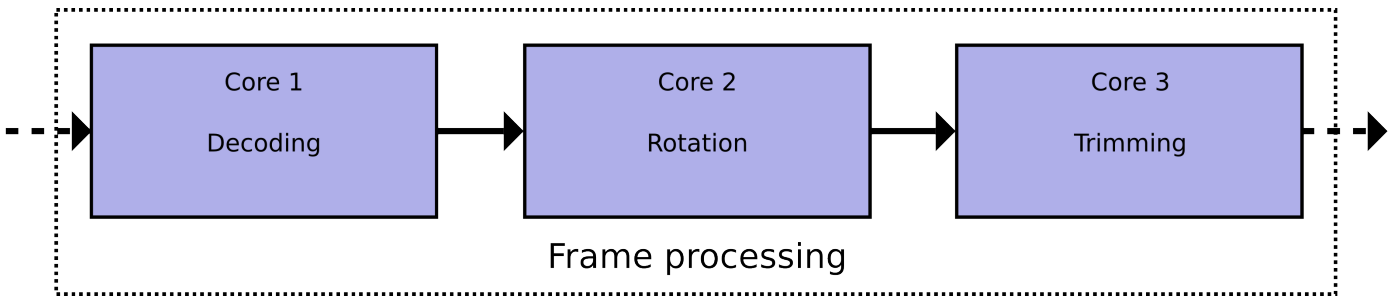
dependencies

- "task": transformations **depend** on result of previous transformations in the chain
- "data": frame decoding **depends** on previously decoded frames

Pipeline parallelism to the rescue

Method to increase the number of images processed per second:

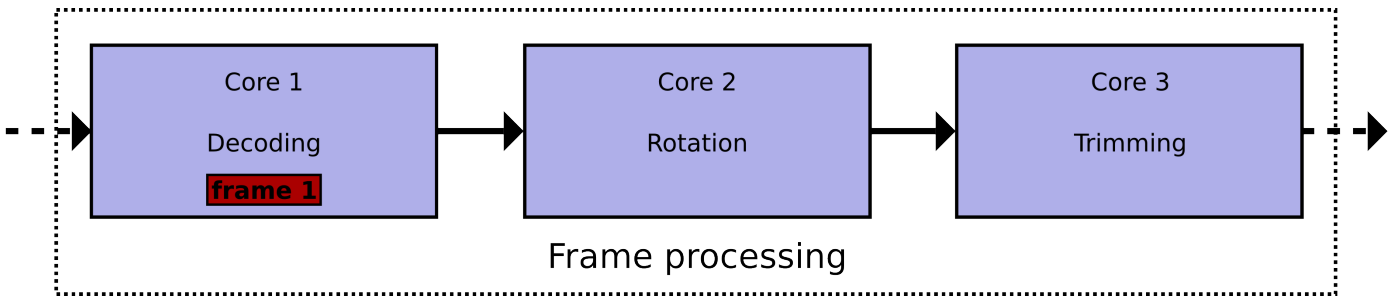
- Split frame processing in 3 sub-tasks:
  1. decoding
  2. rotation
  3. trimming
- Perform each sub-task on different cores
- Make images **flow** from one sub-task to another
  $\Rightarrow$ Sub-tasks performed in parallel for different images



Pipeline parallelism to the rescue

Method to increase the number of images processed per second:

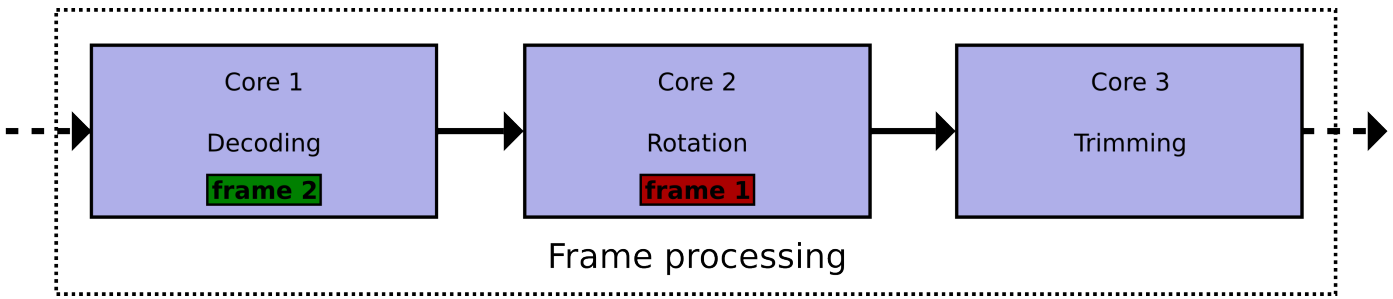
- Split frame processing in 3 sub-tasks:
  1. decoding
  2. rotation
  3. trimming
- Perform each sub-task on different cores
- Make images **flow** from one sub-task to another
  $\Rightarrow$ Sub-tasks performed in parallel for different images



Pipeline parallelism to the rescue

Method to increase the number of images processed per second:

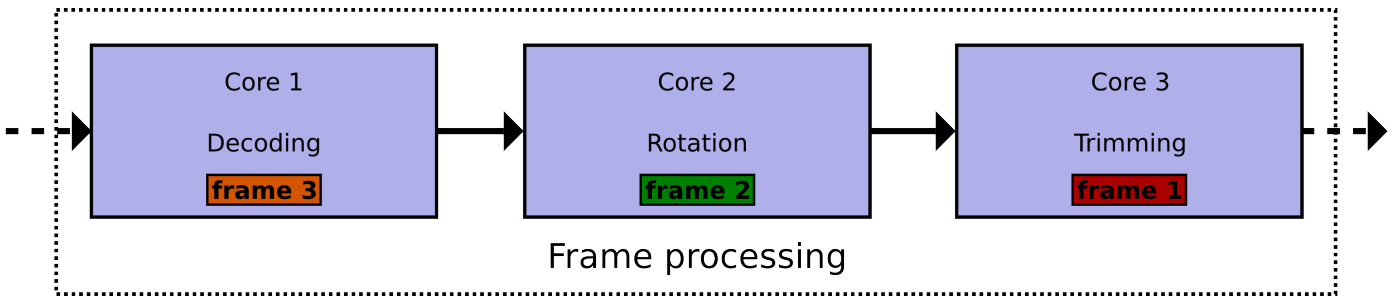
- Split frame processing in 3 sub-tasks:
  1. decoding
  2. rotation
  3. trimming
- Perform each sub-task on different cores
- Make images **flow** from one sub-task to another
  $\Rightarrow$ Sub-tasks performed in parallel for different images



Pipeline parallelism to the rescue

Method to increase the number of images processed per second:

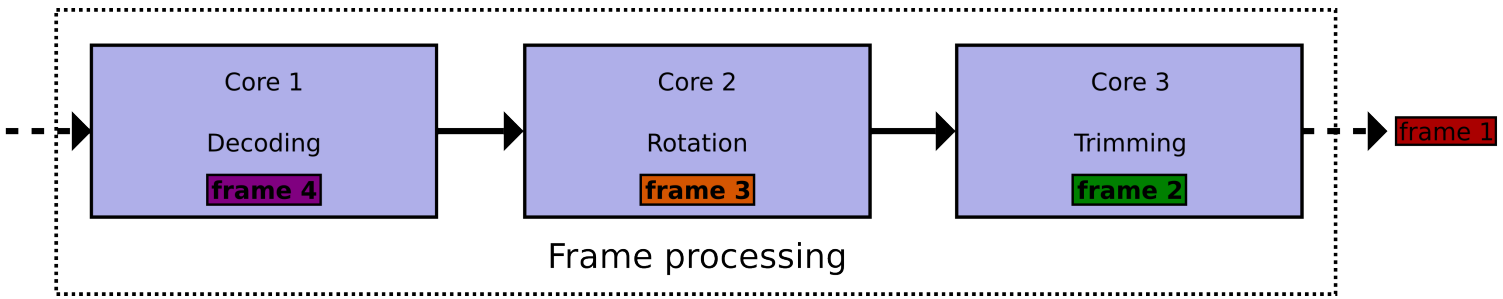
- Split frame processing in 3 sub-tasks:
  1. decoding
  2. rotation
  3. trimming
- Perform each sub-task on different cores
- Make images **flow** from one sub-task to another
  $\Rightarrow$ Sub-tasks performed in parallel for different images



Pipeline parallelism to the rescue

Method to increase the number of images processed per second:

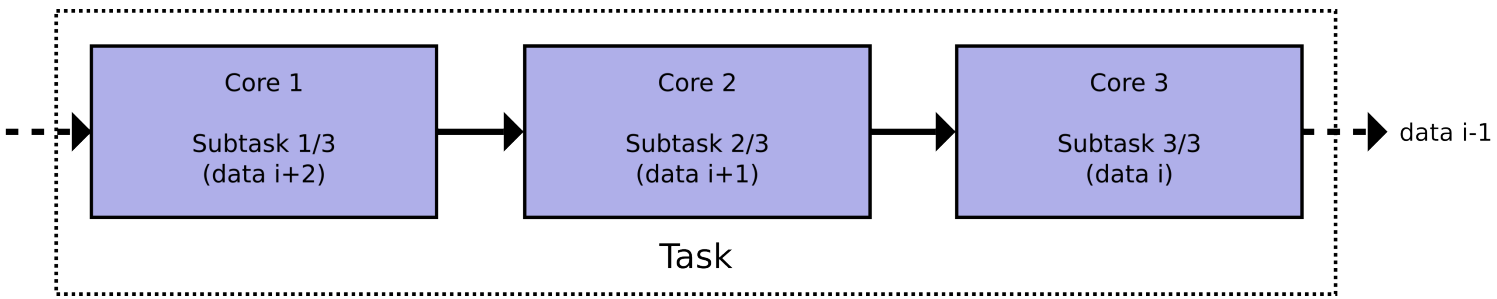
- Split frame processing in 3 sub-tasks:
  1. decoding
  2. rotation
  3. trimming
- Perform each sub-task on different cores
- Make images **flow** from one sub-task to another
  $\Rightarrow$ Sub-tasks performed in parallel for different images



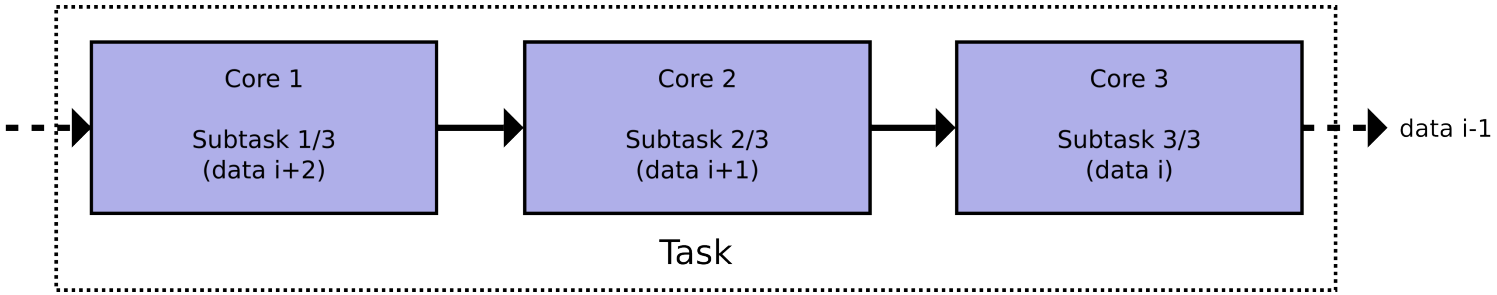
Pipeline parallelism: general case

General principle

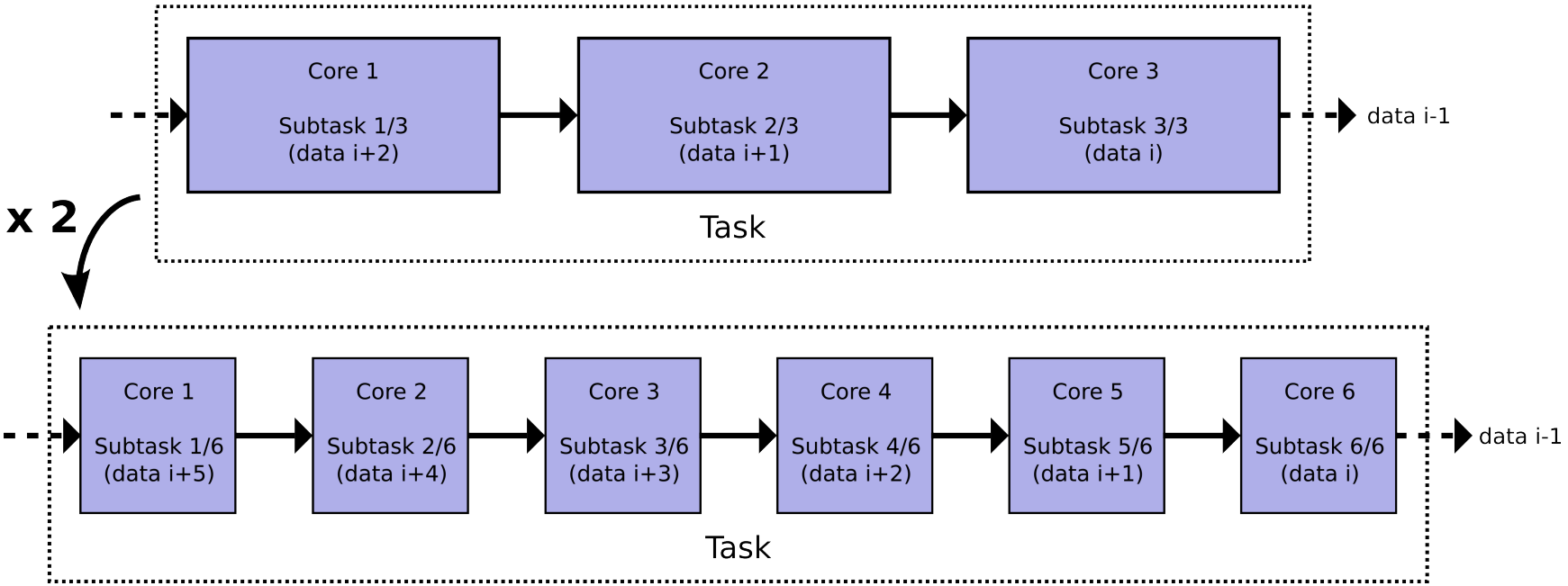
- Divide a sequential code in several sub-tasks
- Execute each sub-task on different cores
- Make data **flow** from one sub-task to another
  ⇒ Sub-tasks run in parallel on different parts of the flow



Efficiency of pipeline parallelism



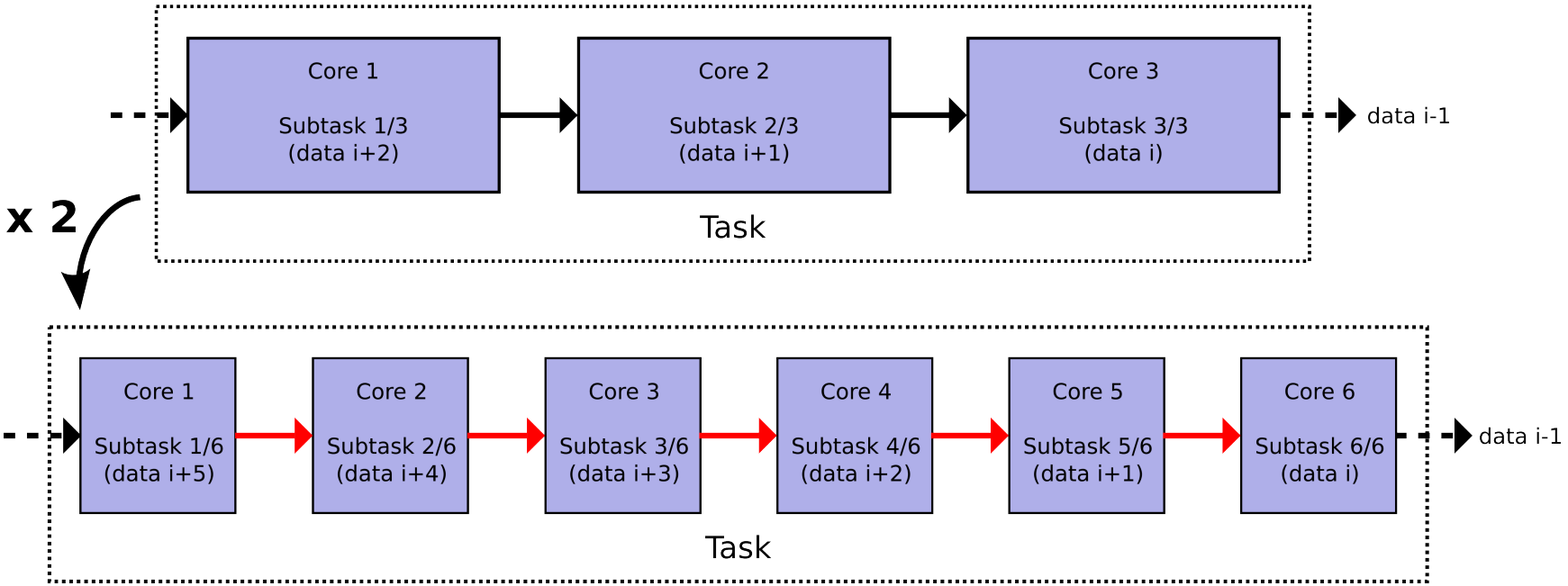
Efficiency of pipeline parallelism



Performance improvement with 6 cores instead of 3:

- Latency: **slower** by 3 T_{comm}
- Throughput: about 2 times **faster**

Efficiency of pipeline parallelism



In the general case, performance for n cores is:

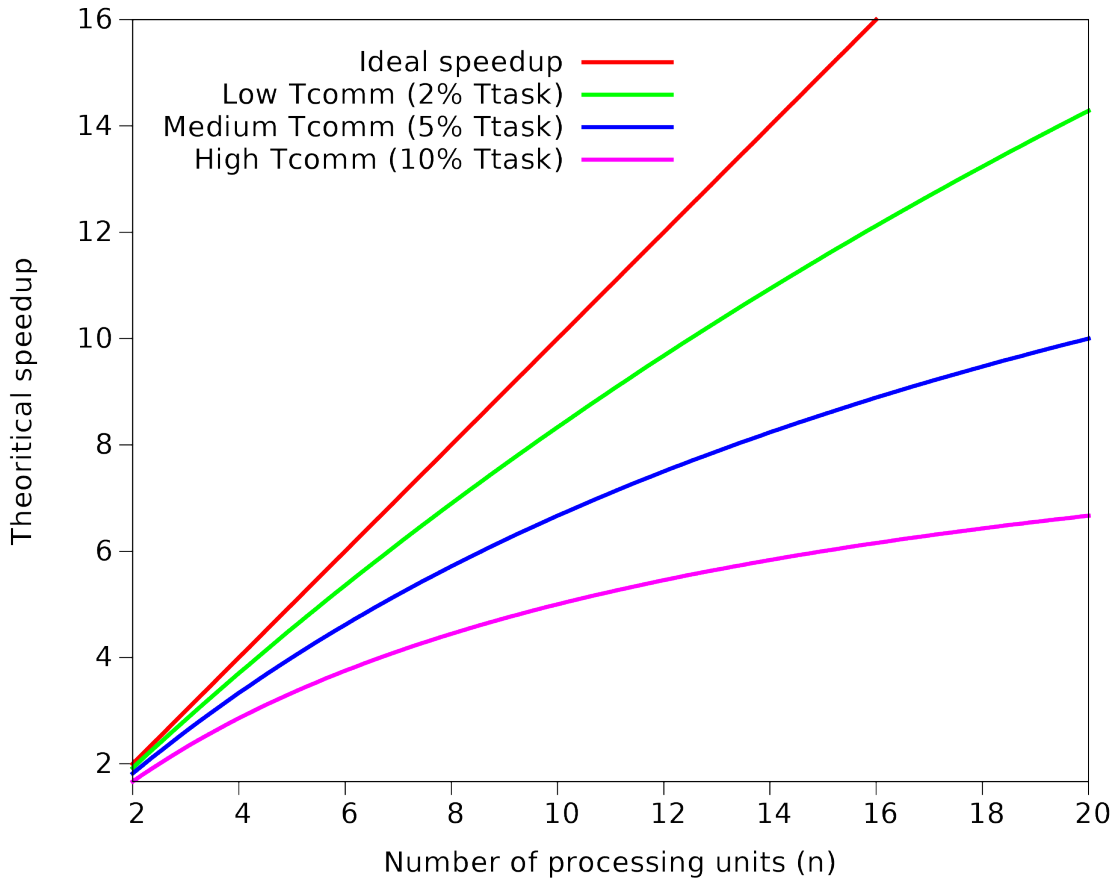
- Latency: $T_{task} + (n-1)T_{comm}$
- Throughput: 1 output every $T_{subtask} + T_{comm}$
  $\Rightarrow$ 1 output every $\frac{T_{task}}{n} + T_{comm}$

Problem

Communication time limits the speedup

Pipeline parallelism: limits

On n cores, one processing done every $\frac{T_{task}}{n} + T_{comm}$



Communication time limits the speedup !

$\Rightarrow$ Need for efficient inter-core communication

Problem statement

Problem 1

Current communication algorithms perform badly for inter-core communication

Problem 2

Changing the communication algorithm of all/many programs doing pipeline parallelism is impractical

Contributions

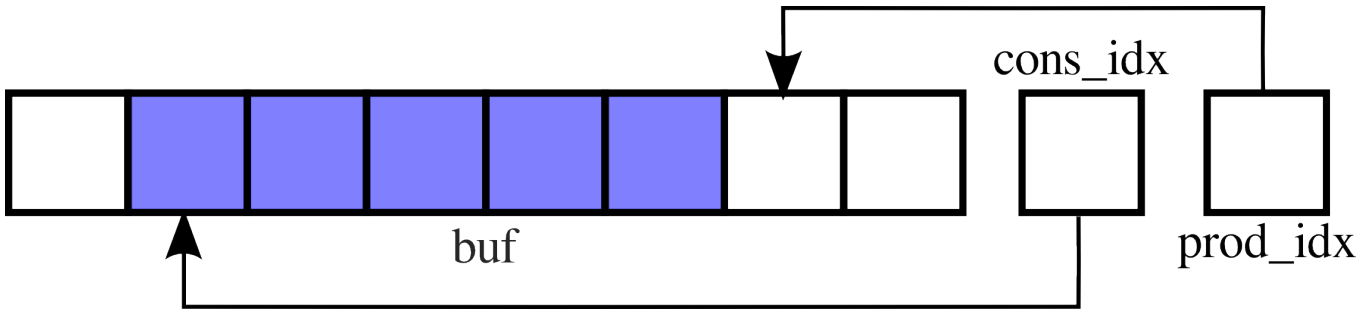
Two-fold solution:

- BatchQueue: queue optimized for inter-core communication
- Automated usage of BatchQueue for pipeline parallelism

Contribution 1

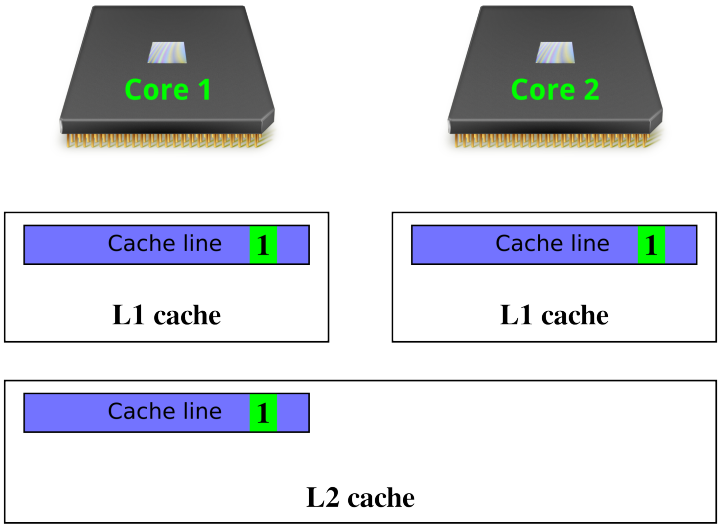
BatchQueue: queue optimized for inter-core communication

Lamport: principle



- Data exchanged by reads and writes in a shared buffer
  ⇒ data read/written sequentially, cycling at end of buffer
- 2 indices to memorize where to read/write next in the buffer
  ⇒ filling of buffer detected via indices comparison

Cache consistency



- Caches with same data must be kept consistent
- Consistency maintained by a hardware component: MOESI

MOESI cache consistency protocol

- Memory in caches divided in lines
  ⇒ Consistency enforced at cache line level
- Lines in each cache have a consistency status:
  Modified, **O**wned, **E**xclusive, **S**hared, **I**nvalid
- MOESI ensures only one line is in Modified or Owned state
  ⇒ Implements a **Read/Write exclusion**.

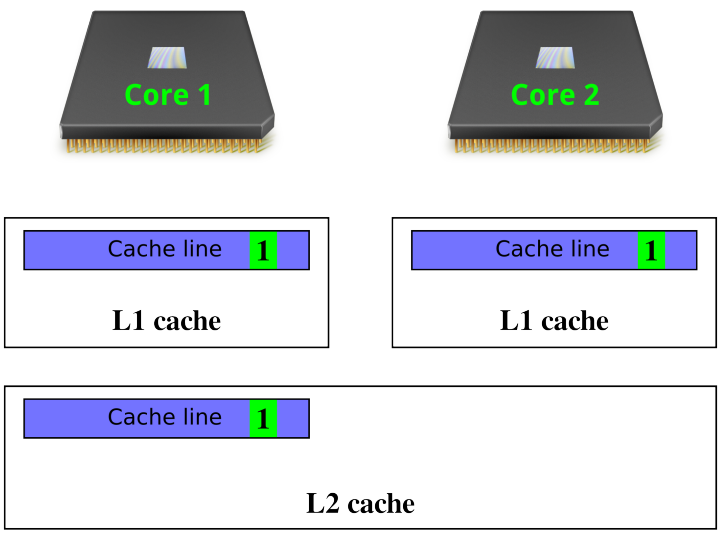
⚠ 3 problems of performance arise from using MOESI

Cache consistency protocol: cost

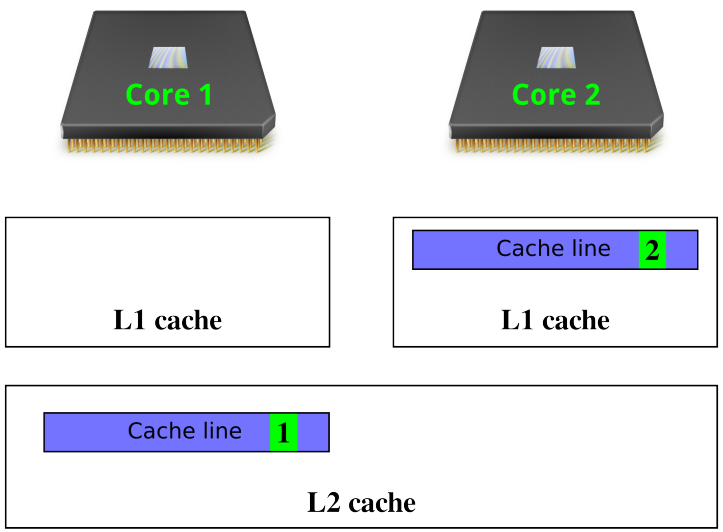
Communication required to update cache lines and their status
⇒ **Cache consistency = slowdown**

2 sources of communication

- Write from Shared or Owned: **invalidate** remote cache lines
- Read from Invalid: **broadcast** to find up-to-date line



Modify line in shared state



Read line in exclusive state

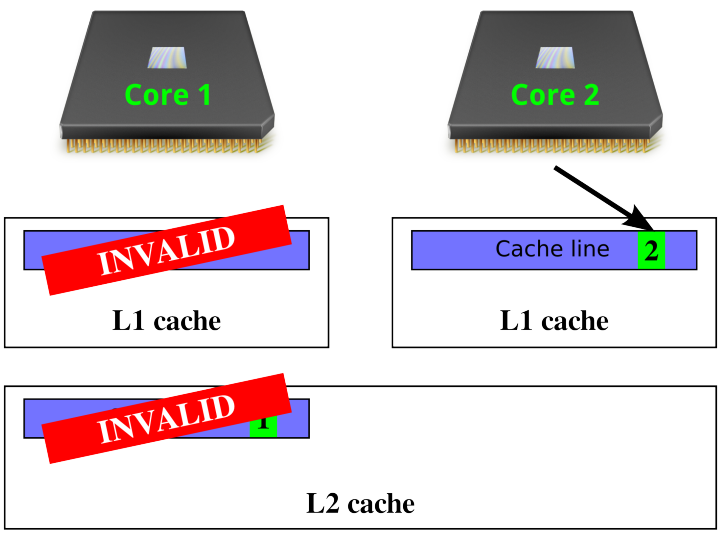
Cache consistency protocol: cost

Communication required to update cache lines and their status

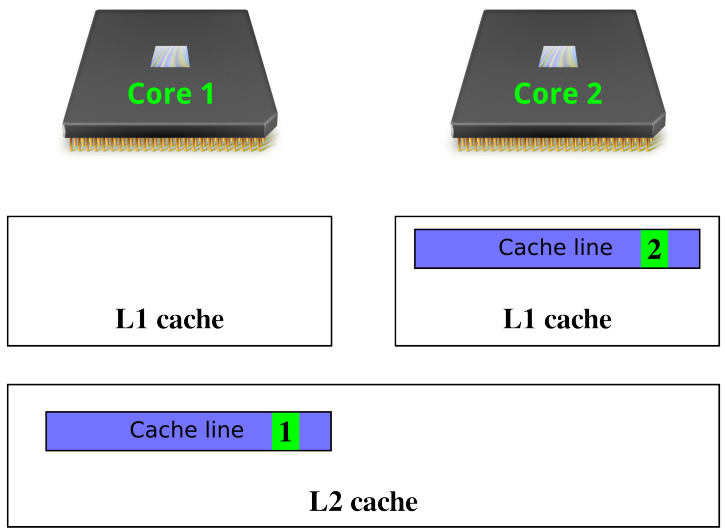
⇒ **Cache consistency = slowdown**

2 sources of communication

- Write from Shared or Owned: **invalidate** remote cache lines
- Read from Invalid: **broadcast** to find up-to-date line



Modify line in shared state



Read line in exclusive state

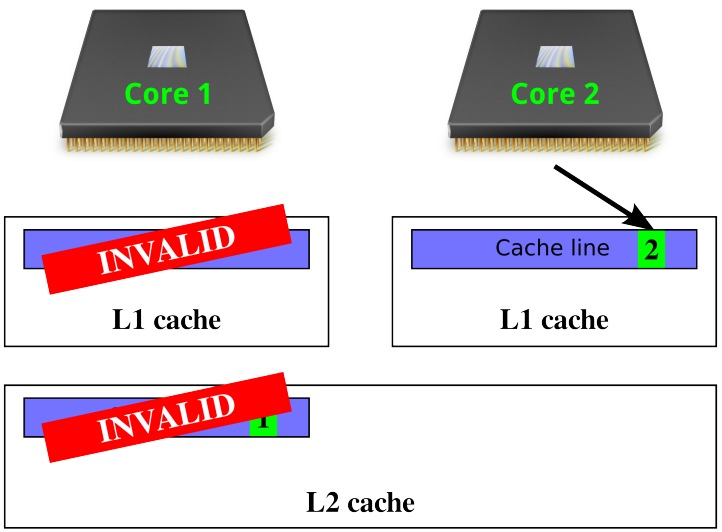
Cache consistency protocol: cost

Communication required to update cache lines and their status

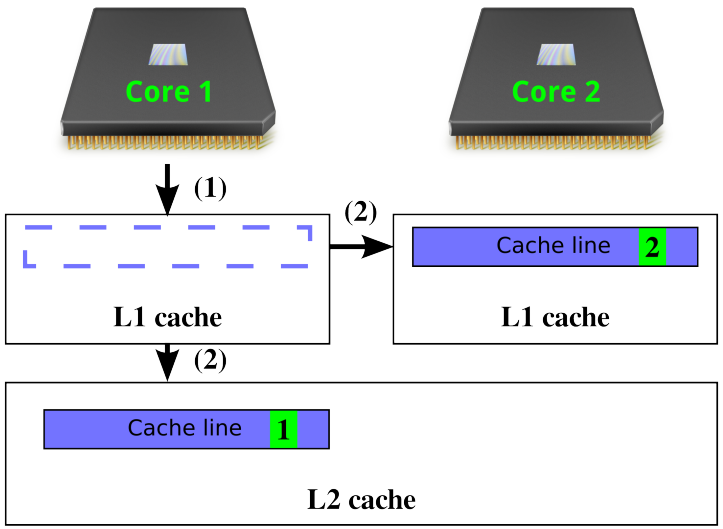
⇒ **Cache consistency = slowdown**

2 sources of communication

- Write from Shared or Owned: **invalidate** remote cache lines
- Read from Invalid: **broadcast** to find up-to-date line



Modify line in shared state



Read line in exclusive state

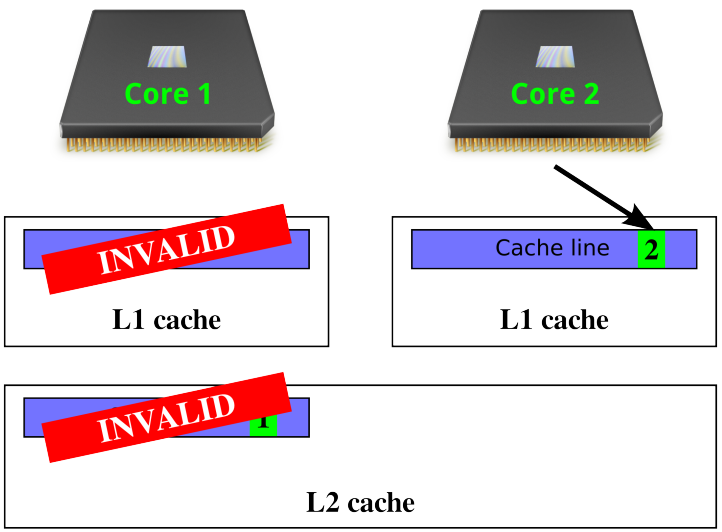
Cache consistency protocol: cost

Communication required to update cache lines and their status

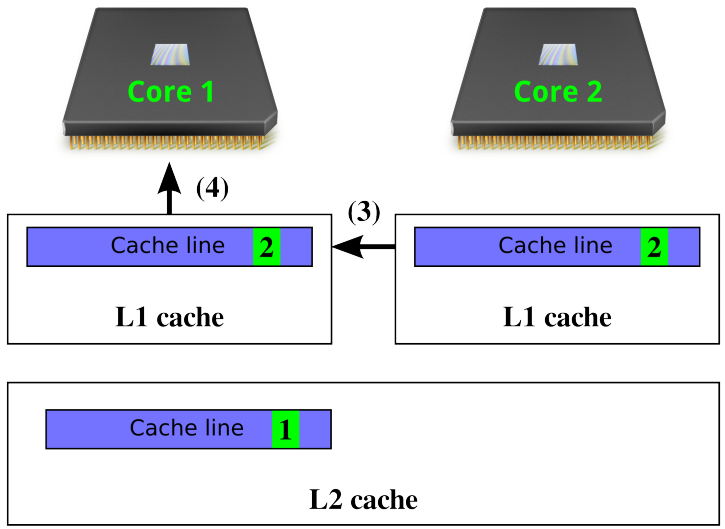
⇒ **Cache consistency = slowdown**

2 sources of communication

- Write from Shared or Owned: **invalidate** remote cache lines
- Read from Invalid: **broadcast** to find up-to-date line

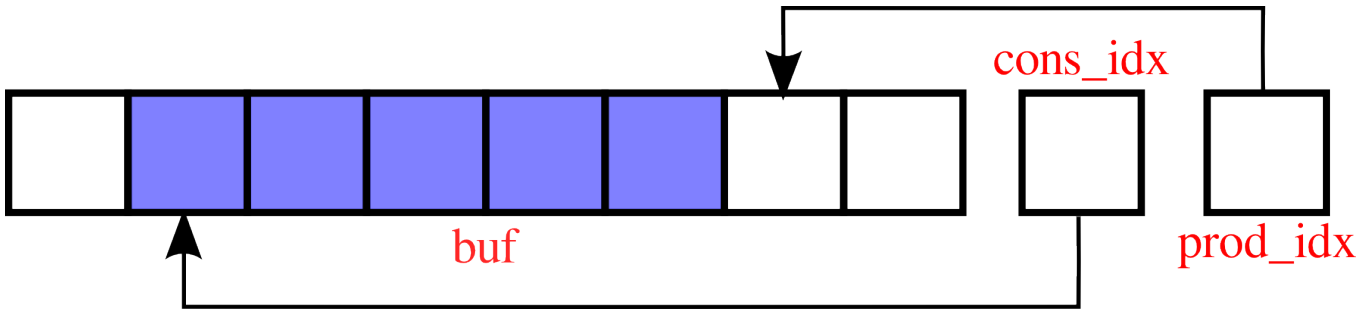


Modify line in shared state



Read line in exclusive state

Lamport: cache friendliness



3 shared variables: buf, prod_idx and cons_idx

Lockless algorithm tailored to **single core** systems

1. high reliance on memory consistency
   - synchronization for each production and consumption
   - 2 variables needed for synchronization

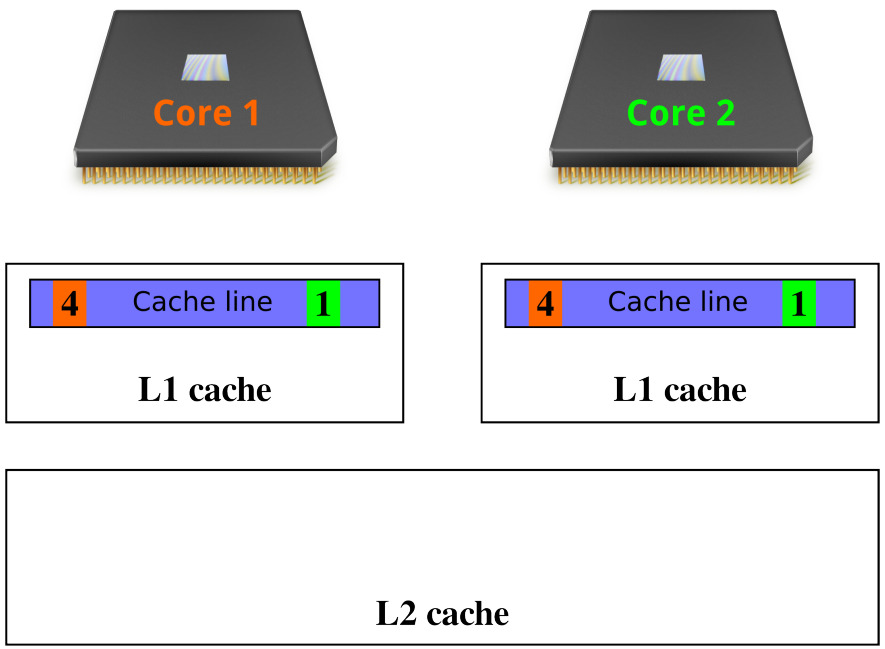
Cache consistency: further slowdown

False sharing problem

Per cache line consistency status

⇒ data sharing detected at cache line level

⇒ accesses to ≠ data from same cache line appears concurrent



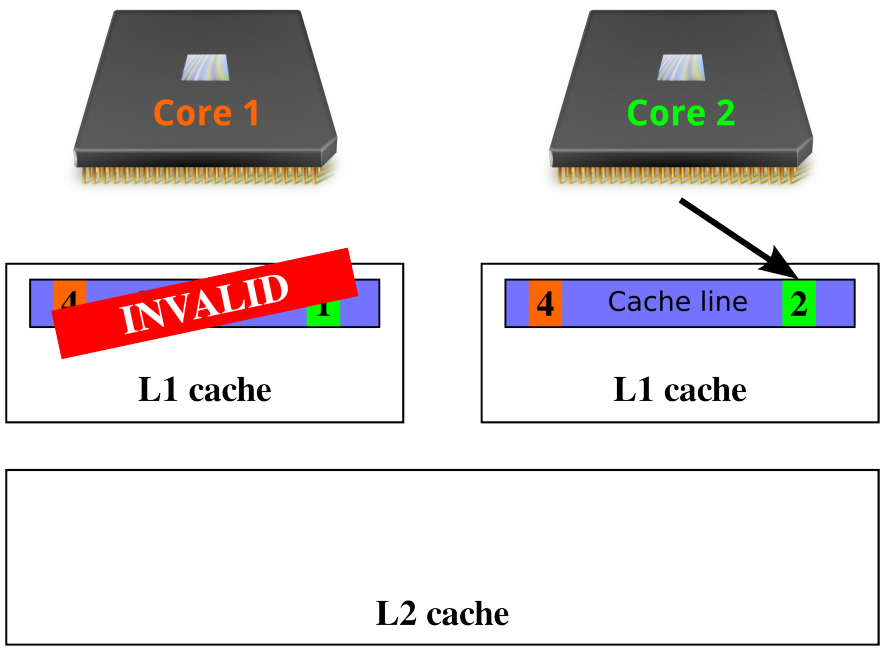
Cache consistency: further slowdown

False sharing problem

Per cache line consistency status

⇒ data sharing detected at cache line level

⇒ accesses to ≠ data from same cache line appears concurrent



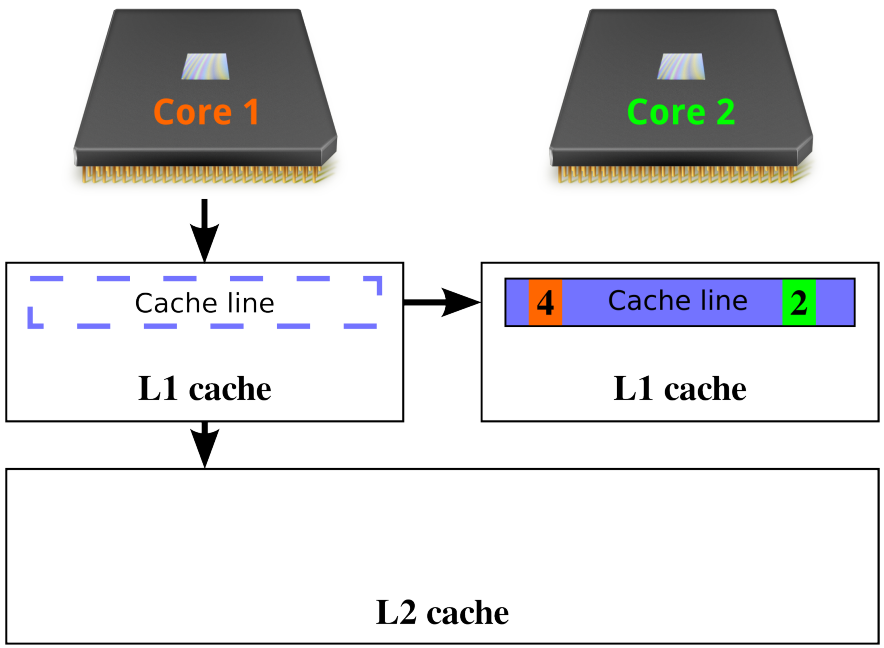
Cache consistency: further slowdown

False sharing problem

Per cache line consistency status

⇒ data sharing detected at cache line level

⇒ accesses to ≠ data from same cache line appears concurrent



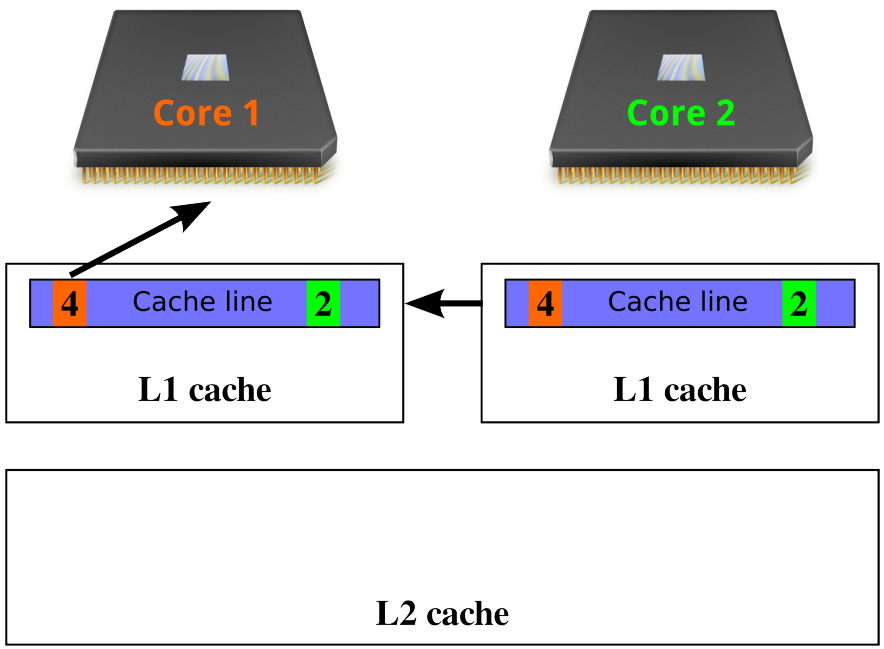
Cache consistency: further slowdown

False sharing problem

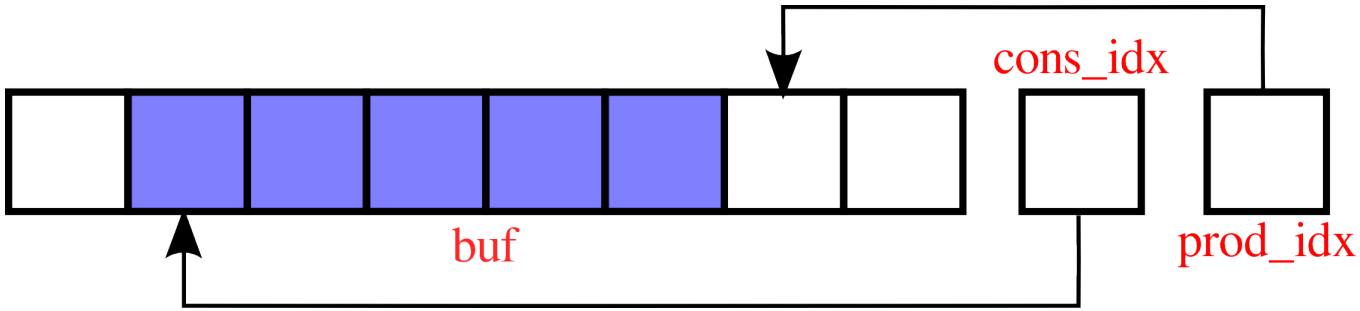
Per cache line consistency status

⇒ data sharing detected at cache line level

⇒ accesses to ≠ data from same cache line appears concurrent



Lamport: cache friendliness



prod_idx and cons_idx may point to nearby entries

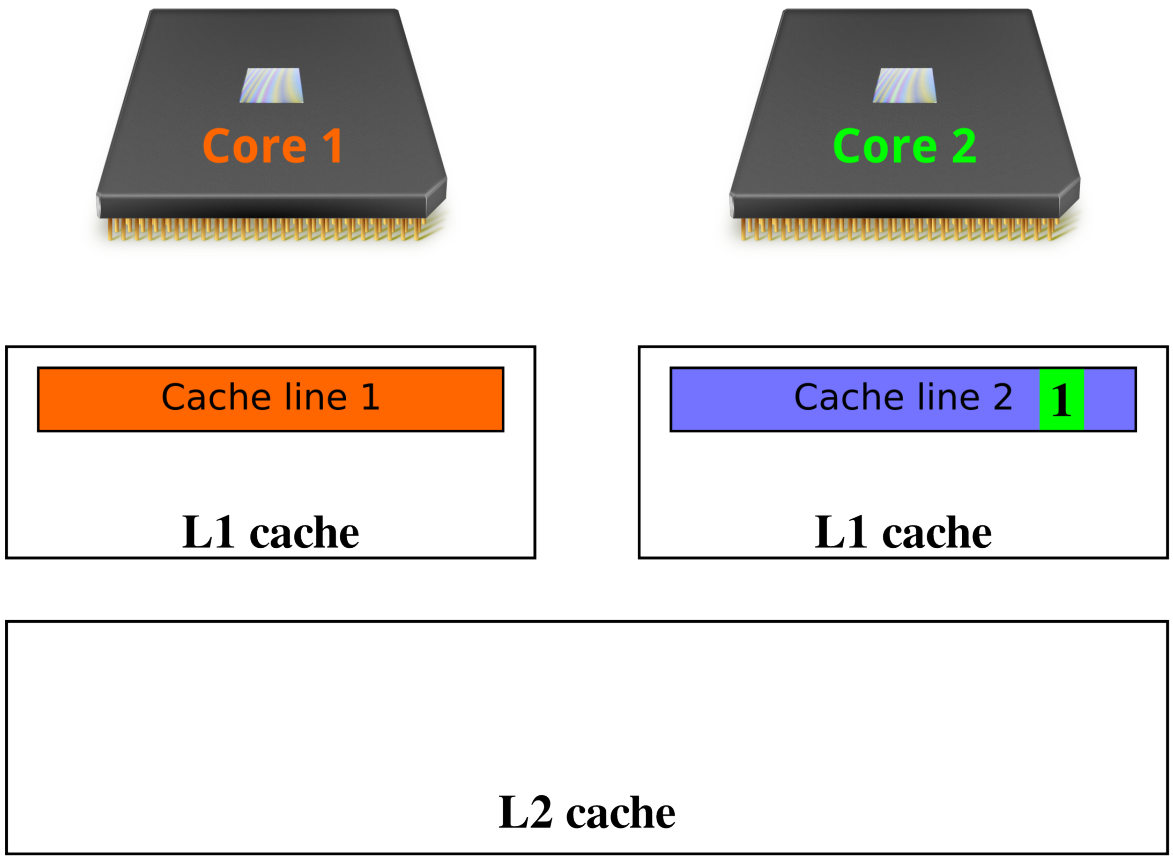
Lockless algorithm tailored to **single core** systems

1. high reliance on memory consistency
   - synchronization for each production and consumption
   - 2 variables needed for synchronization
2. false sharing
   - producer and consumer often work on nearby entries

False sharing due to prefetch

- Prefetch consists in **fetching** data before they are needed
- read + disjoint write access in same cache line = false sharing

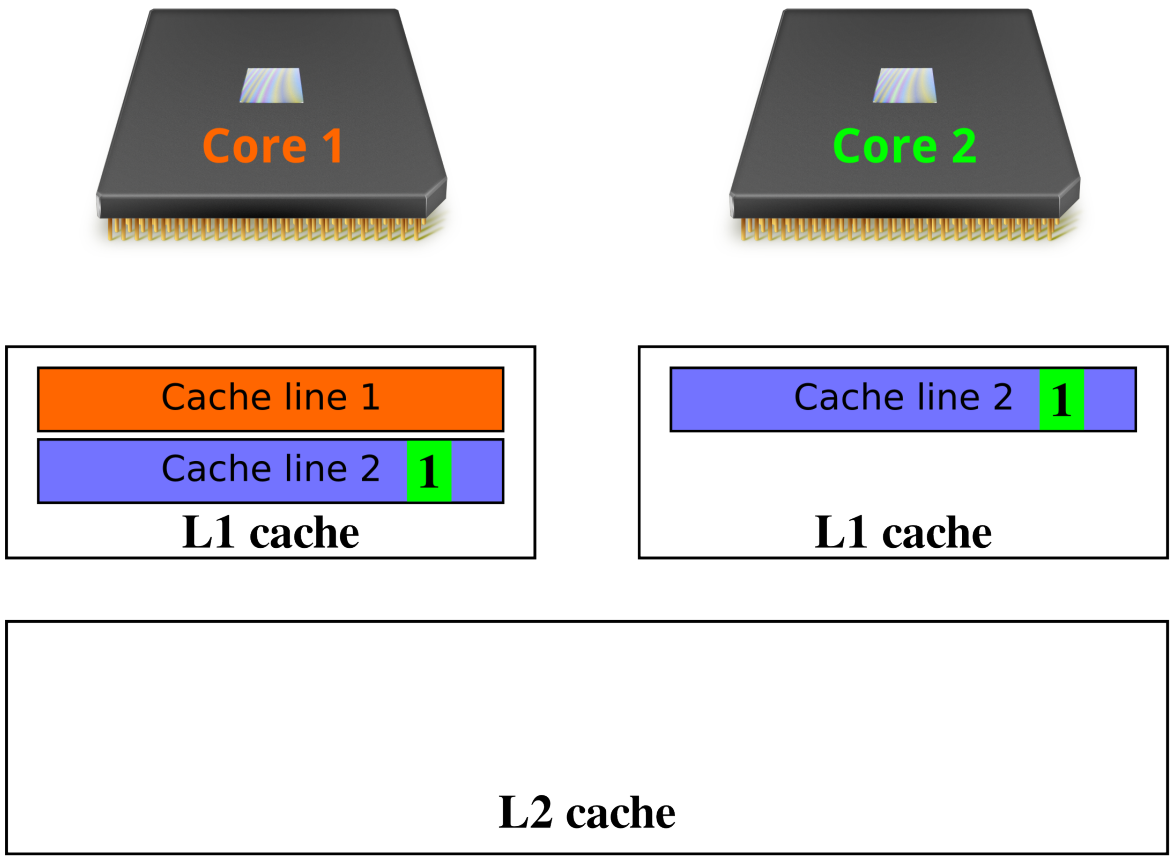
$\Rightarrow$ Prefetch can create false sharing



False sharing due to prefetch

- Prefetch consists in **fetching** data before they are needed
- read + disjoint write access in same cache line = false sharing

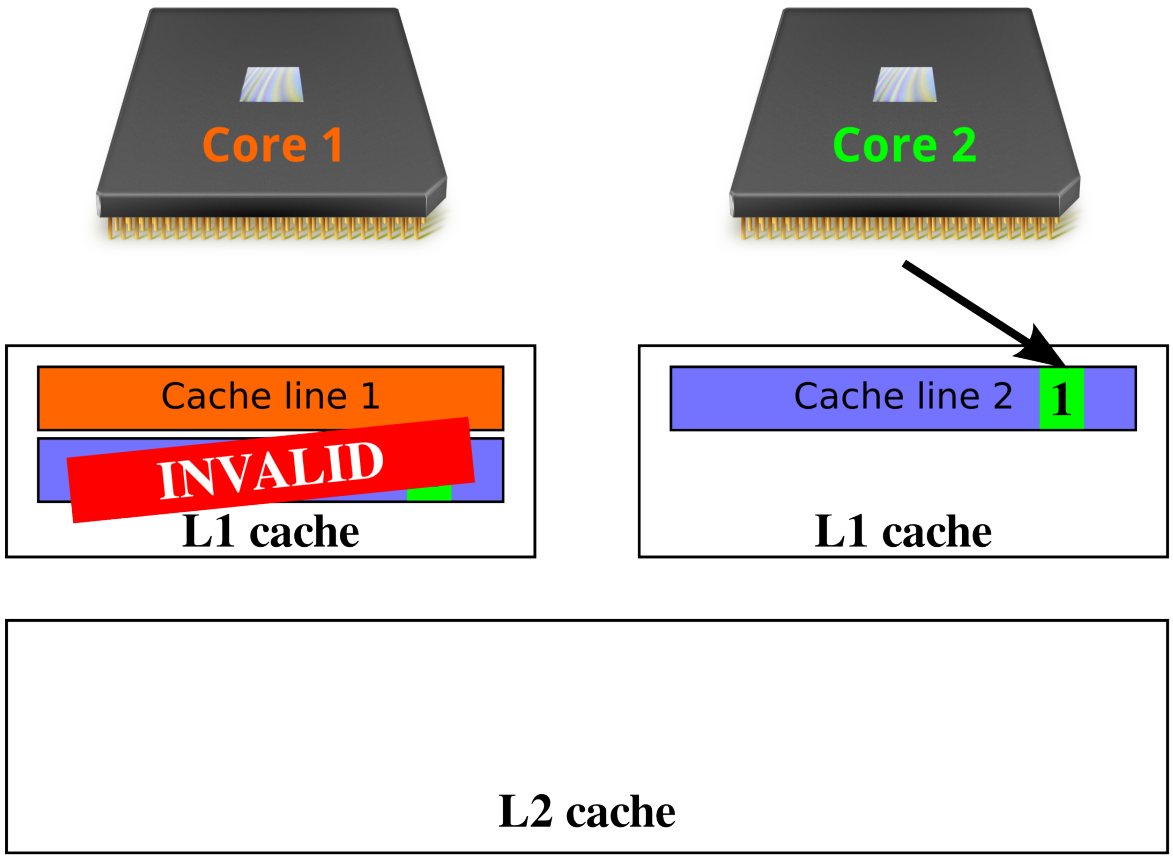
$\Rightarrow$ Prefetch can create false sharing



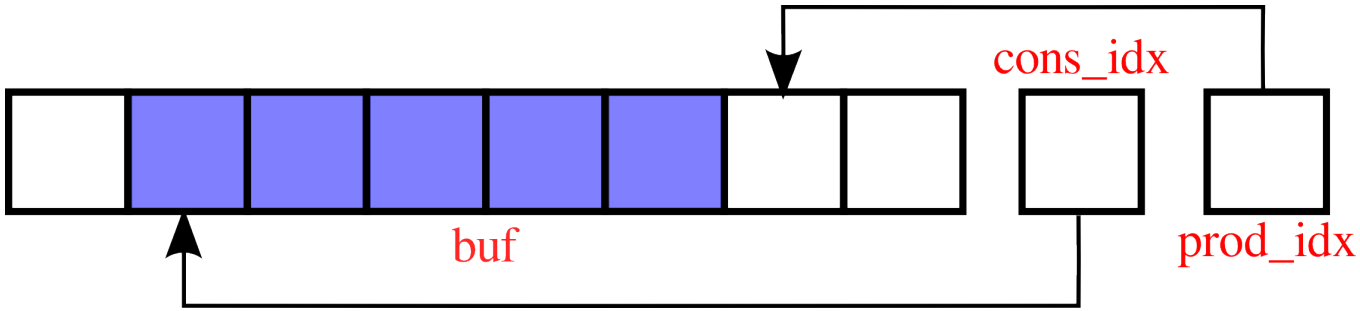
False sharing due to prefetch

- Prefetch consists in **fetching** data before they are needed
- read + disjoint write access in same cache line = false sharing

$\Rightarrow$ Prefetch can create false sharing



Lamport: cache friendliness



All entries read and written sequentially

Lockless algorithm tailored to **single core** systems

1. High reliance on memory consistency
   - synchronization for each production and consumption
   - 2 variables needed for synchronization
2. False sharing
   - producer and consumer often work on nearby entries
3. Undesirable prefetch
   - prefetch may create false sharing on distant entries

State-of-the-art algorithms on multi-cores

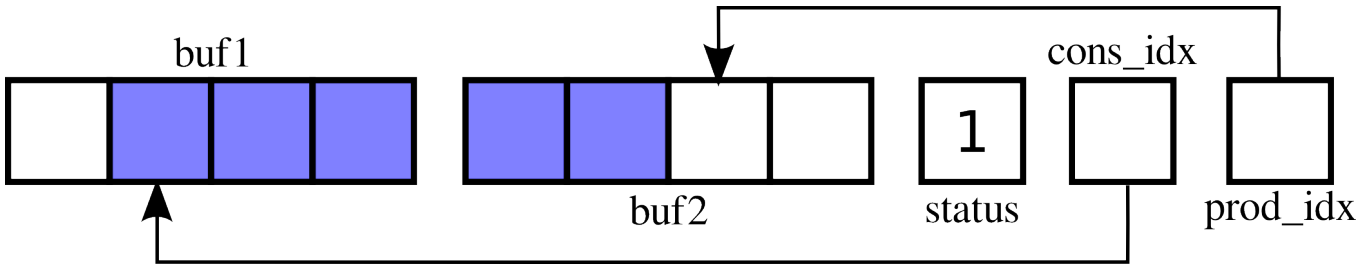
	Quantity of sharing	False sharing	Wrong prefetch
Lamport [Lam83]	All variables shared	KO	KO
FastForward [GMV08]	Only buffer	KO	KO
CSQ [ZOYB09]	N global variables	OK	KO
MCRingBuffer [LBC10]	2 global variables	OK	KO

Objectives

3 problems to solve:

1. Problem 1: excessive synchronization
2. Problem 2: false sharing of data
3. Problem 3: undesirable prefetch

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

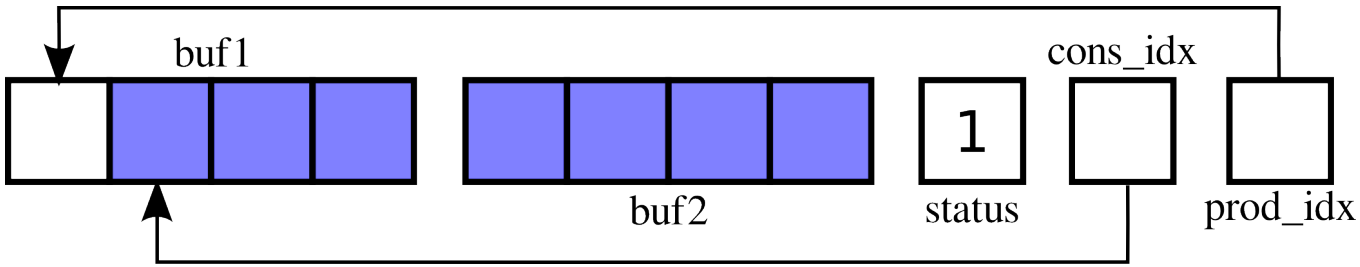
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

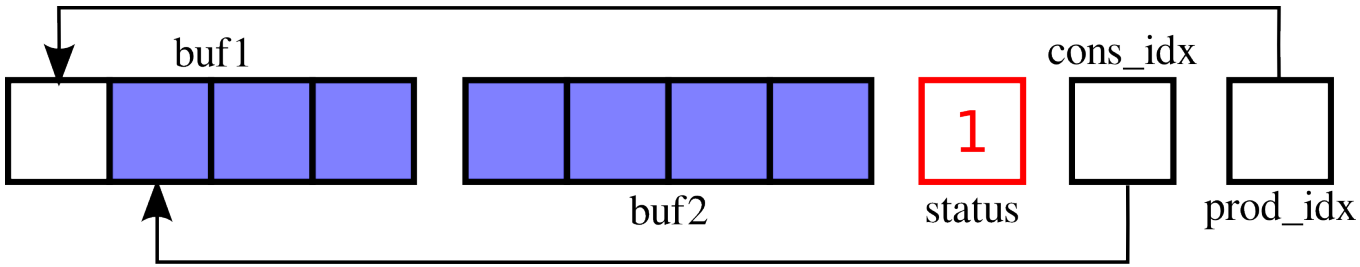
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

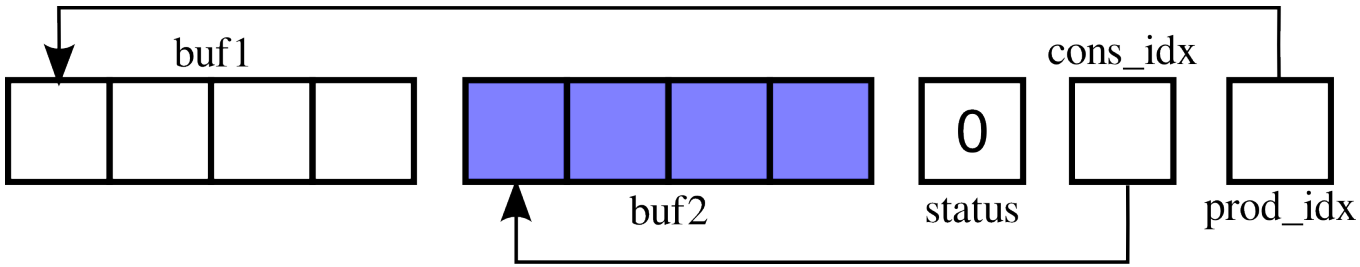
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

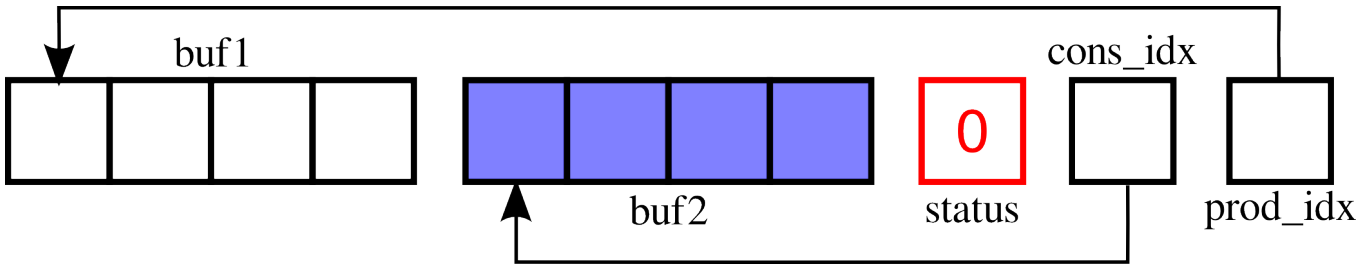
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

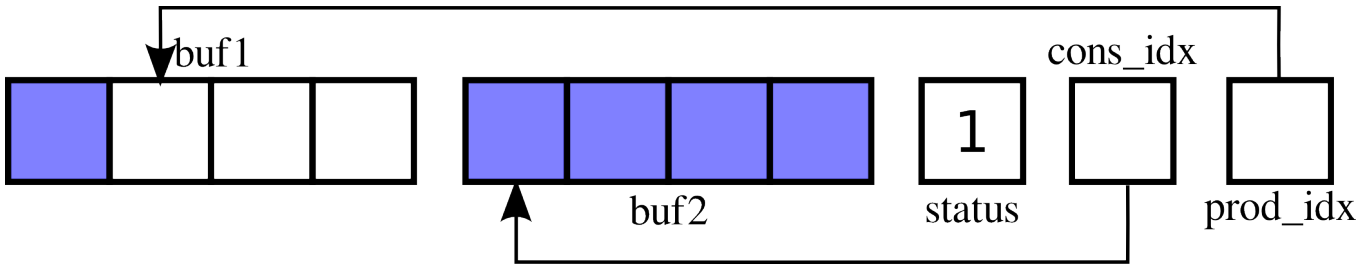
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: principle



Communication through 2 semi-buffers:
production in one semi-buffer, consumption in the other

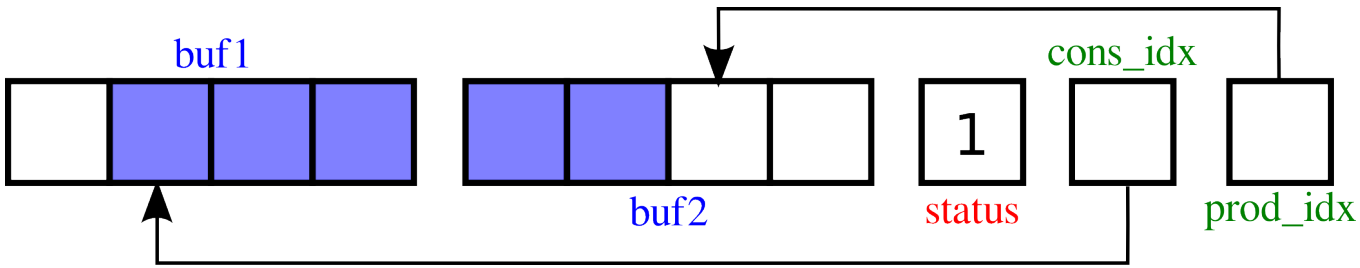
When one semi-buffer is fully filled/emptied:

- producer: switch *status* to 1 if equal to 0
- consumer: switch *status* to 0 if equal to 1

Synchronization invariant

status switched twice ⇒ semi-buffers can be exchanged

BatchQueue: cache friendliness (1)

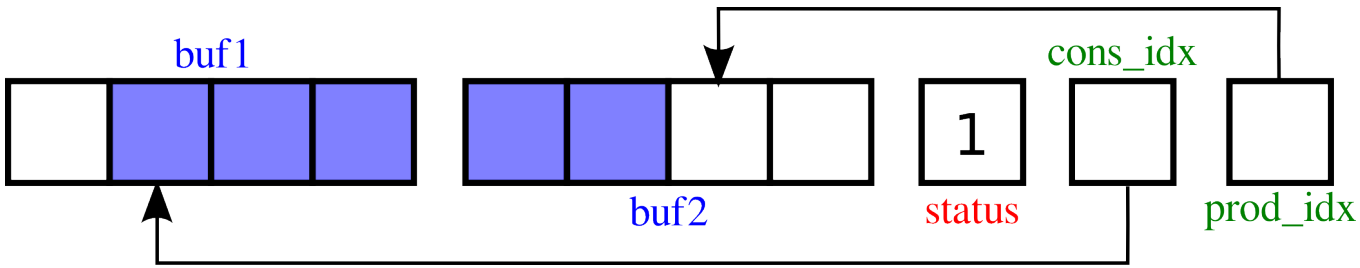


2 private variables: prod_idx and cons_idx
2 semi-private buffers: buf1 and buf2
1 shared variable: status

Problem 1: reduce the amount of synchronization

+ batch processing for fewer synchronization
+ synchronize on a single variable

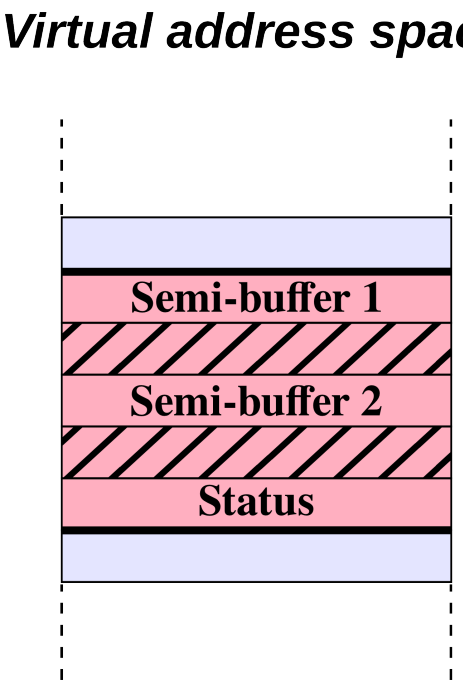
BatchQueue: cache friendliness (2)



Problem 2: avoid false sharing

+ producer and consumer work on separate buffers
+ alignment of buffers and variables on cache line boundaries

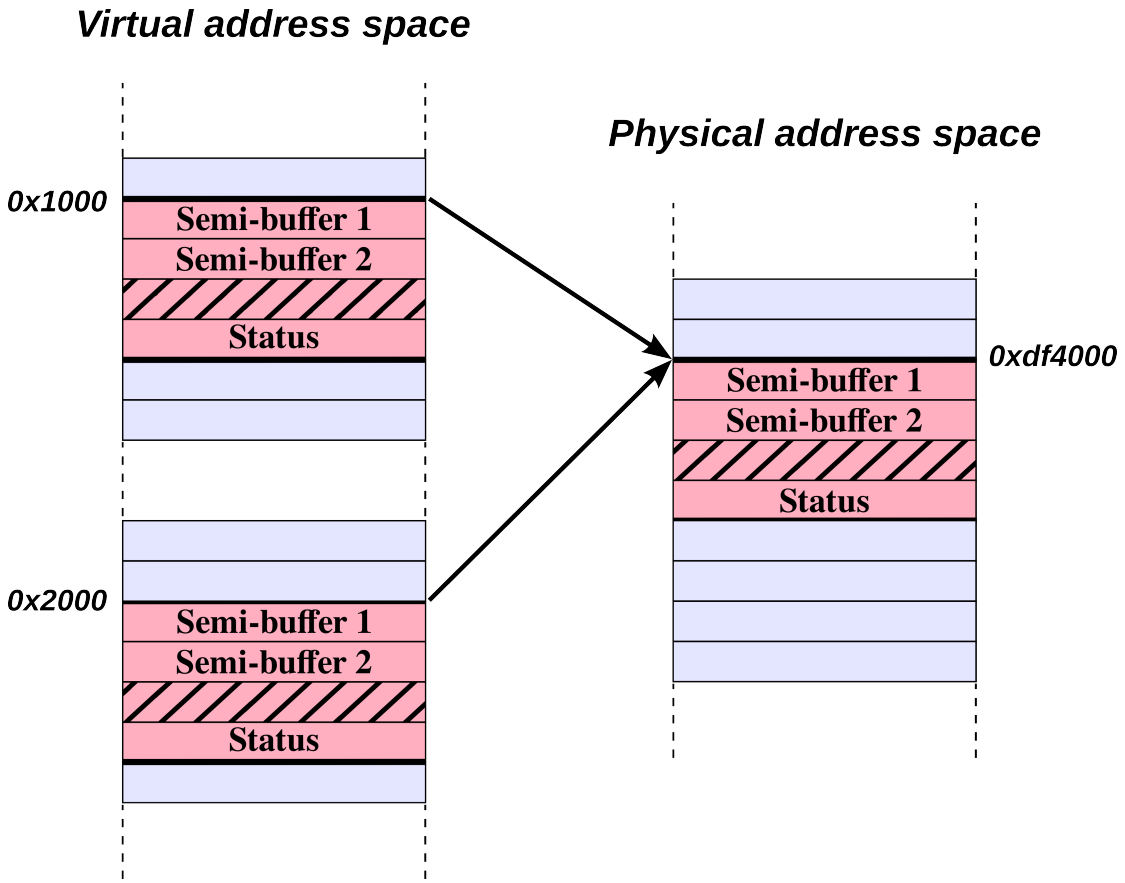
BatchQueue: cache friendliness (3)



Problem 3: prevent undesirable prefetch

+ padding between each component of the structure?
  ⇒ prevent optimizations possible with contiguous buffers

Avoiding false sharing due to prefetch



Problem 3: prevent undesirable prefetch

- \+ Add some padding between semi-buffers and *status* variable
- \+ Access each semi-buffer through a different memory mapping
  $\Rightarrow$ consistency of L1 caches based on virtual addresses

Algorithms on multi-cores

	Quantity of sharing	False sharing	Wrong prefetch
Lamport [Lam83]	All variables shared	KO	KO
FastForward [GMV08]	Only buffer	KO	KO
CSQ [ZOYB09]	N boolean variables	OK	KO
MCRingBuffer [LBC10]	2 variables	OK	KO
BatchQueue [PSTF10]	1 boolean variable	OK	OK

BatchQueue: lockless algorithm tailored to cache coherency

1. synchronization reduced and simplified
2. no false sharing of data
3. sharing made explicit with different memory mappings

Microbench: test descriptions

Principle:

- Send data between the two cores
- Measure time to transfer all data

Two variants of the micro benchmark:

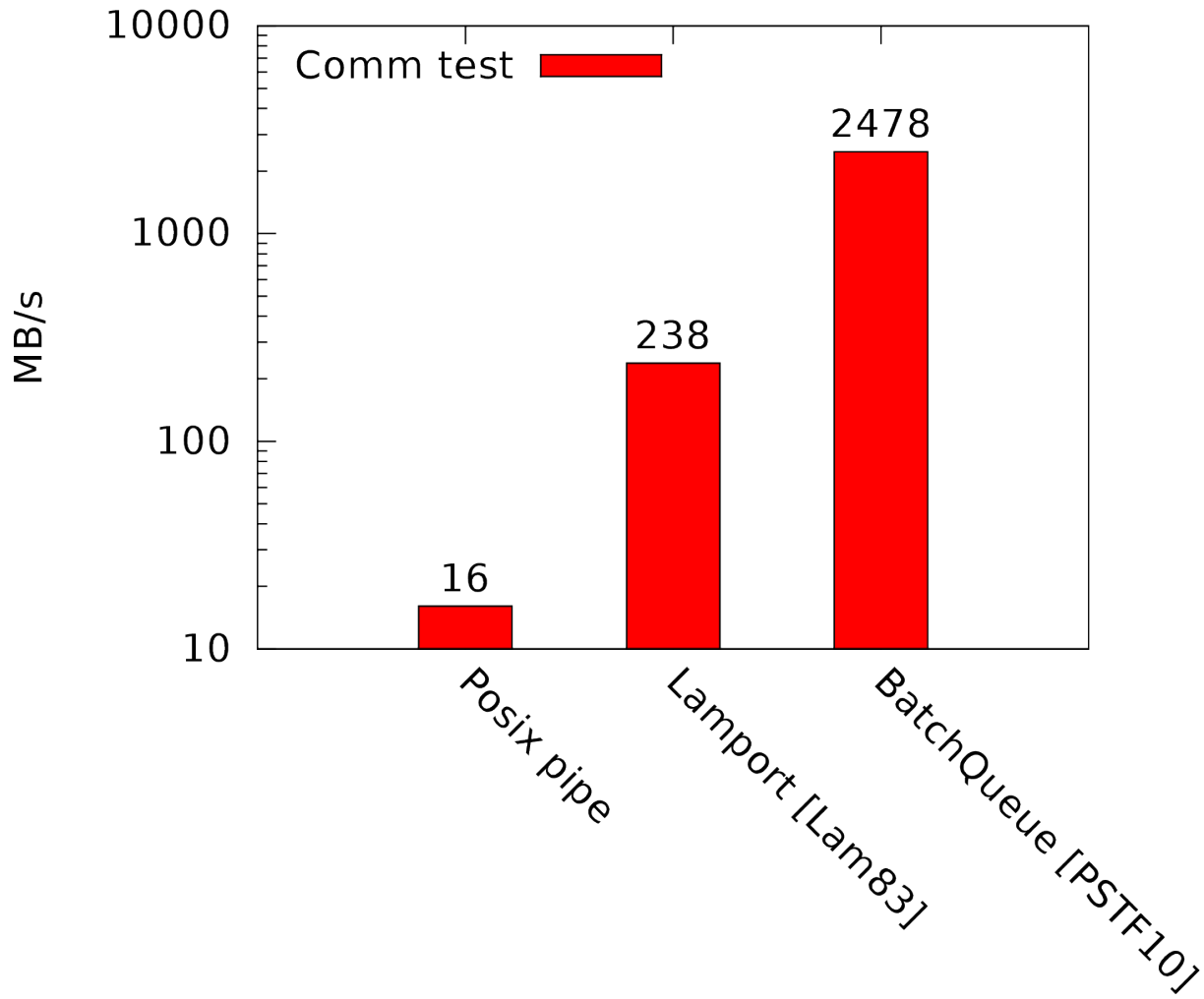
- "*comm*" test ⇒ measure maximum throughput
- "*matrix*" test ⇒ measure throughput when L1 under pressure

Machines:

- bossa (except NUMA)
  - Processors: Intel Xeon X5427 quad-core 3GHz,
  - Memory: 10 GiB RAM, 32 KiB L1, 6 MiB L2 shared by pair
  - System: Linux 3.2 (64 bits), gcc 4.6.3 (-03 + inline functions)
- amd48 (for NUMA only)
  - Processors: AMD Opteron 6172 hexa-core 2.1GHz
  - Memory: 32 GiB RAM, 64 KiB L1, 512 KiB L2, 5 MiB L3
  - System: Linux 3.0 (64 bits), gcc 4.6.3 (-03 + inline functions)

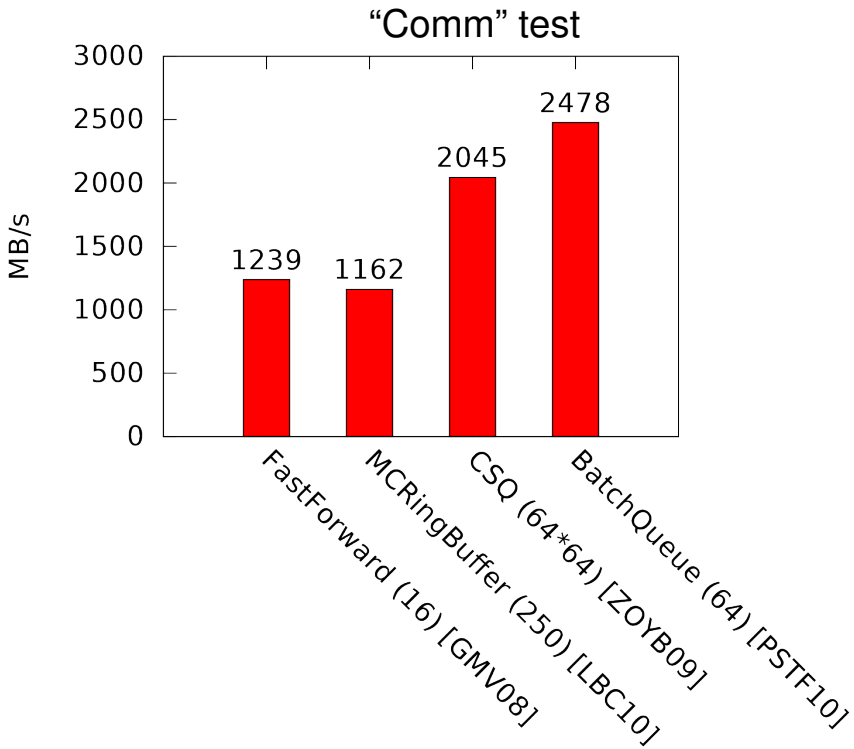
Microbench evaluation: order of magnitude

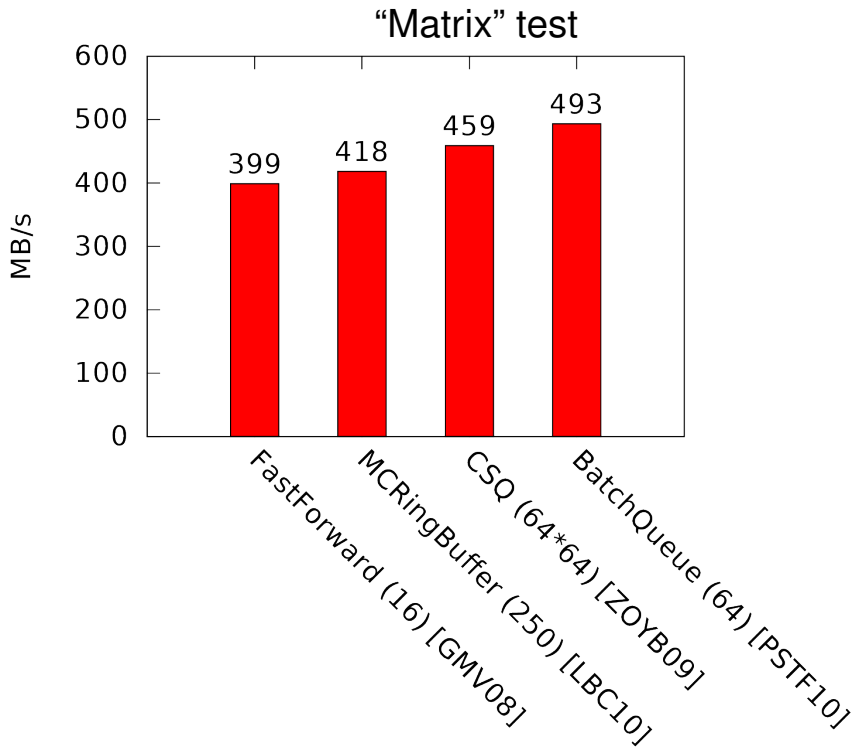
Order of magnitude in speed of communication algorithms



Microbench evaluation: default configuration

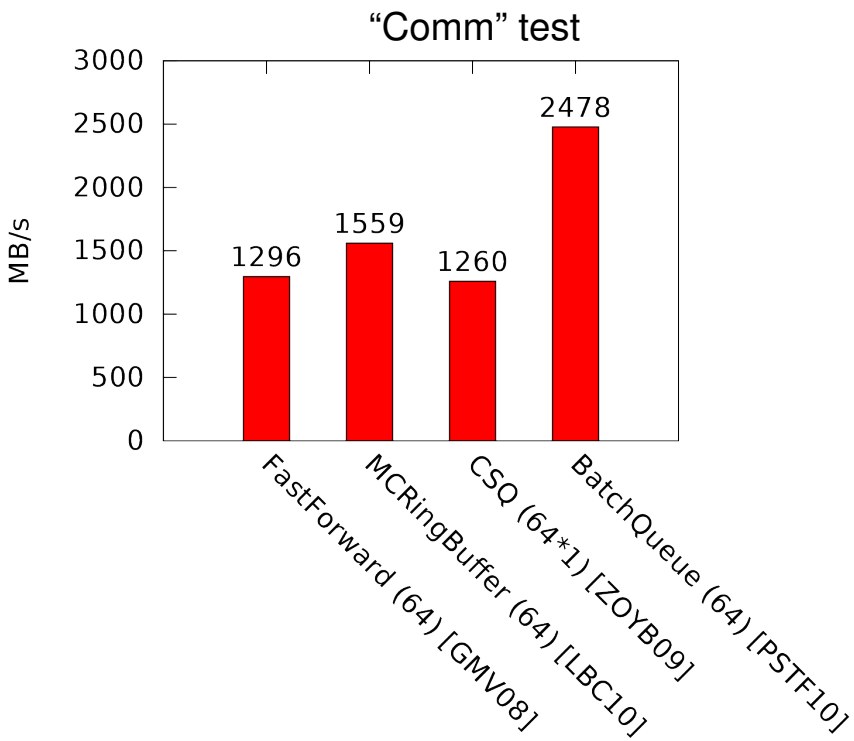
Comparison of communication algorithms with default configuration

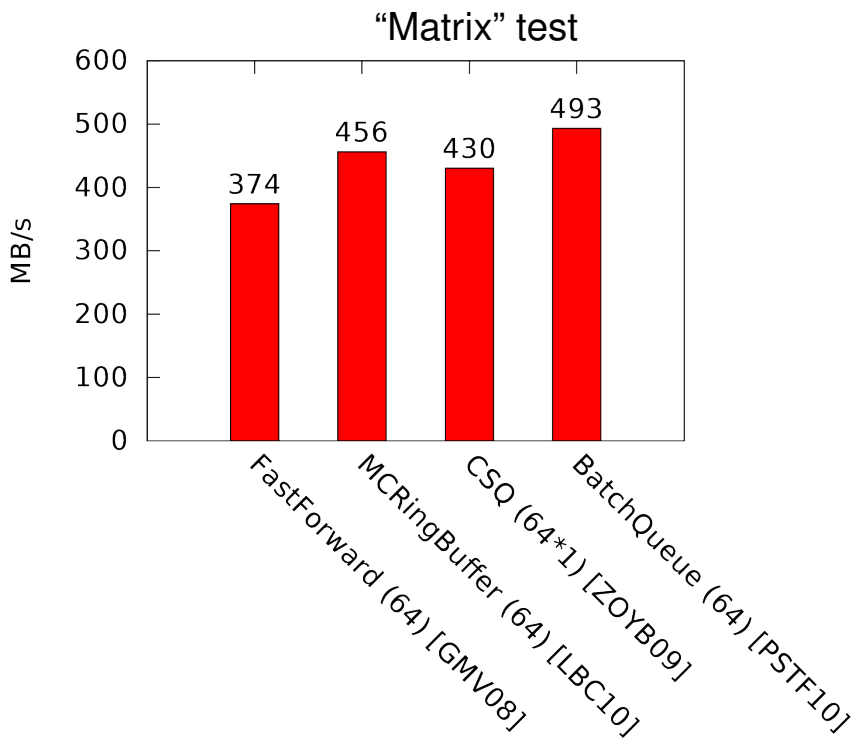




Microbench evaluation: fixed buffer size

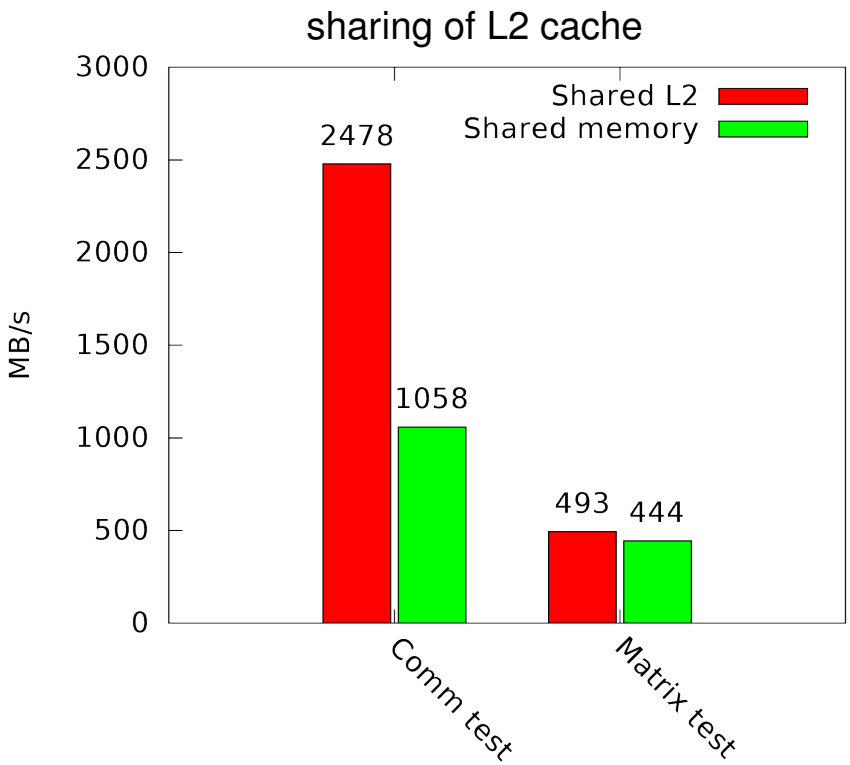
Comparison of communication algorithms with same buffer size

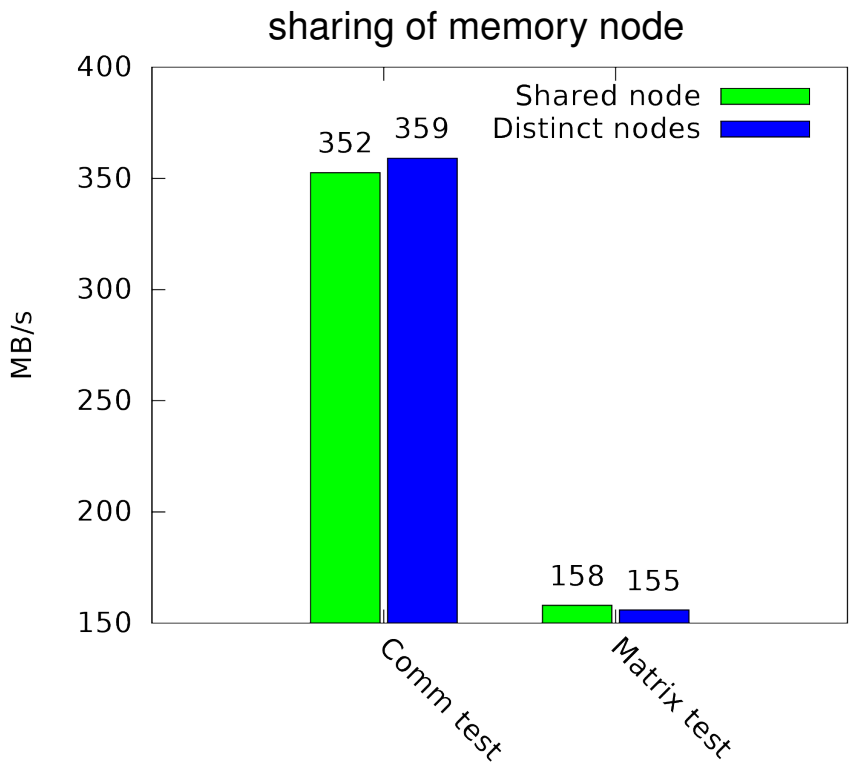




Microbench evaluation: cache sharing

Influence of memory hierarchy on BatchQueue's performance





Prefetch can only mitigate against small latencies

Contribution 2

Automated usage of BatchQueue for pipeline parallelism

Parallelization frameworks

Parallelizing a program requires a lot of commonplace code:

- thread management (creation, scheduling, termination)
- synchronization (mutex, barriers)
- communication

Some high level frameworks exist to hide these details:

- Data/task parallelism: OpenMP, Threading Building Blocks, Cilk Plus, ...
- Pipeline parallelism: StreamIt, **OpenMP stream-computing extension**

Improving these frameworks benefits all programs using them

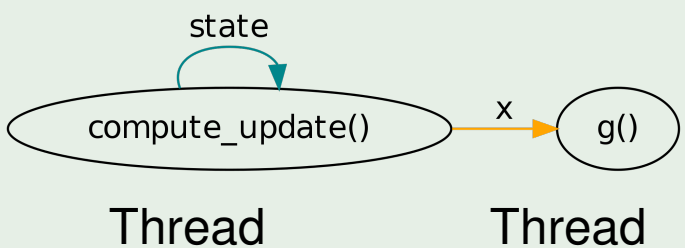
OpenMP stream-computing extension

OpenMP stream-computing extension offers a familiar syntax
⇒ more likely to be used by many programs

Example d'utilisation

```
#pragma omp parallel
#pragma omp single
for (i = 0; i < N; i++) {
   #pragma omp task input(state) output (x, state)
   x = compute_update(&state);

   #pragma omp task input (x)
   retval = g(x);
}
```

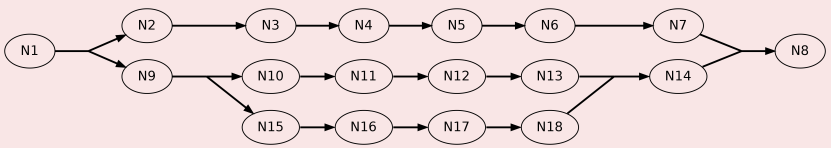


Improving OpenMP stream-computing extension

Problem

It uses MPMC (Multiple Producers Multiple Consumers) queues internally for communication. Yet:

1. MPMC incurs extra synchronization cost (among producers and among consumers)
2. Pipeline parallelism is mostly about linear streams

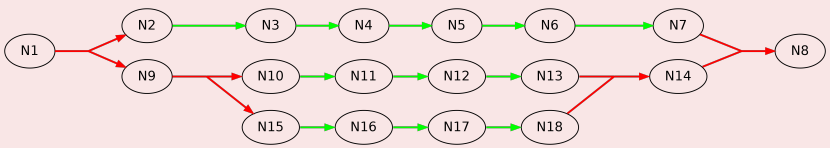


Improving OpenMP stream-computing extension

Problem

It uses MPMC (Multiple Producers Multiple Consumers) queues internally for communication. Yet:

1. MPMC incurs extra synchronization cost (among producers and among consumers)
2. Pipeline parallelism is mostly about linear streams

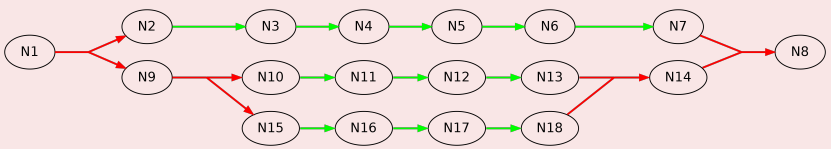


Improving OpenMP stream-computing extension

Problem

It uses MPMC (Multiple Producers Multiple Consumers) queues internally for communication. Yet:

1. MPMC incurs extra synchronization cost (among producers and among consumers)
2. Pipeline parallelism is mostly about linear streams



Solution

Automatic selection of BatchQueue for linear streams
$\Rightarrow$ compatibility retained

BatchQueue in OpenMP stream-computing extension

2 sets of modifications:

1. make communication algorithms interchangeable
2. allow transparent use of BatchQueue

BatchQueue in OpenMP stream-computing extension

2 sets of modifications:

1. make communication algorithms interchangeable
2. allow transparent use of BatchQueue

1st step: interchangeable communication algorithms

Adapt BatchQueue to OpenMP stream-computing extension API:

- **adopt similar function calling sequences:** return value of functions passed as parameter of subsequent function calls
- **adopt similar structure organisation:** different functions are passed in different structures
- **zero-copy communication:** production and consumption directly to and from the communication buffer

BatchQueue in OpenMP stream-computing extension

2 sets of modifications:

1. make communication algorithms interchangeable
2. allow transparent use of BatchQueue

2nd step: transparent use of BatchQueue

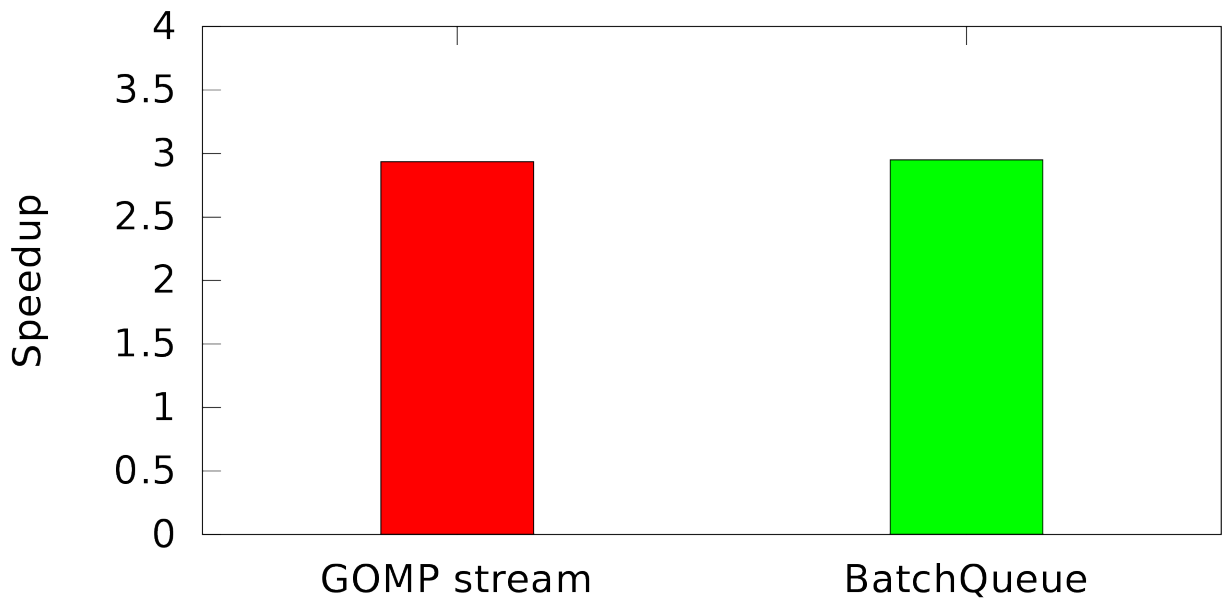
- **Automatic selection of BatchQueue for linear streams**
- **Buffer size proportional to the number of participants**
  $\Rightarrow$ keep memory footprint of both algorithms similar

FMradio

Function: FM demodulation via a serie of filters
Source: OpenMP stream-computing extension paper

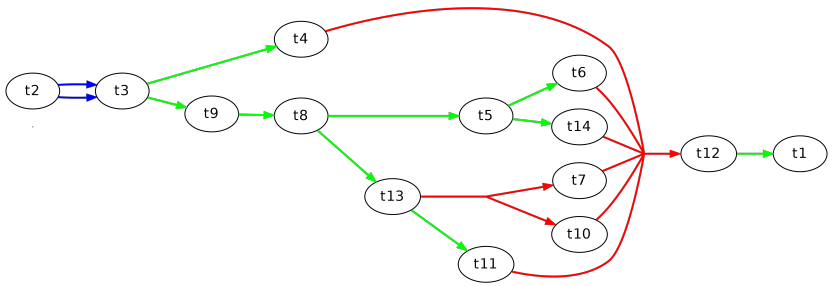
Machine quadhexa

- Processors: Intel Xeon X7460 hexa-core 2.6GHz,
- Memory: 126 GiB RAM, 32 KiB L1, 3 MiB L2 shared by pair
- System: Linux 3.6 (64 bits), gcc 4.6.0

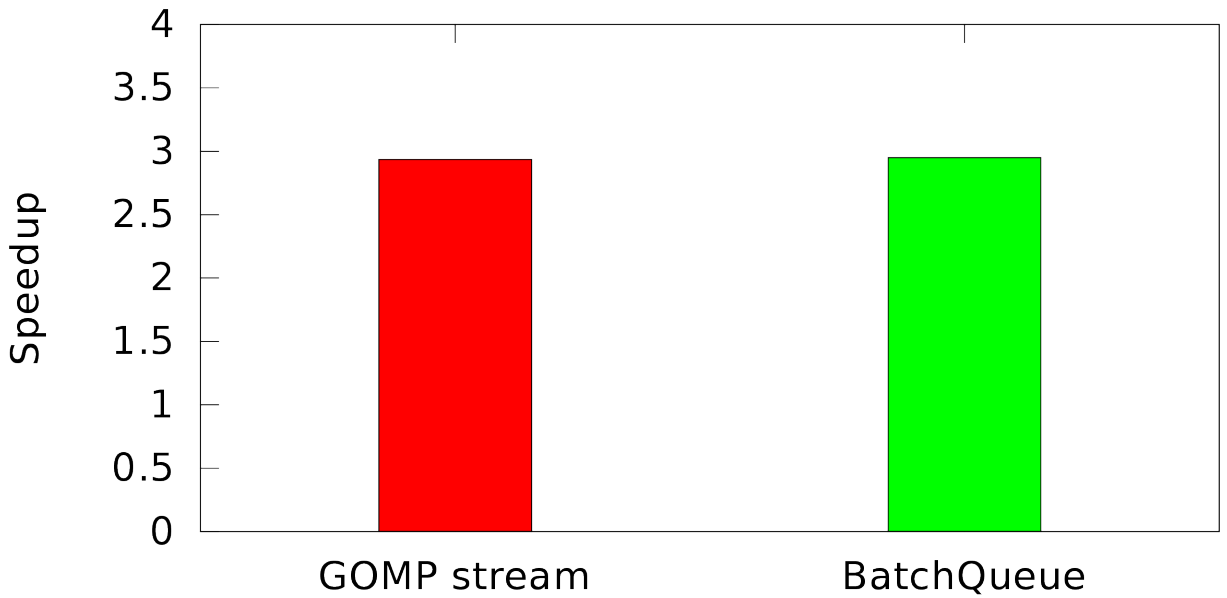


FMradio

Function: FM demodulation via a serie of filters
Source: OpenMP stream-computing extension paper
Particularity: non linear pipeline

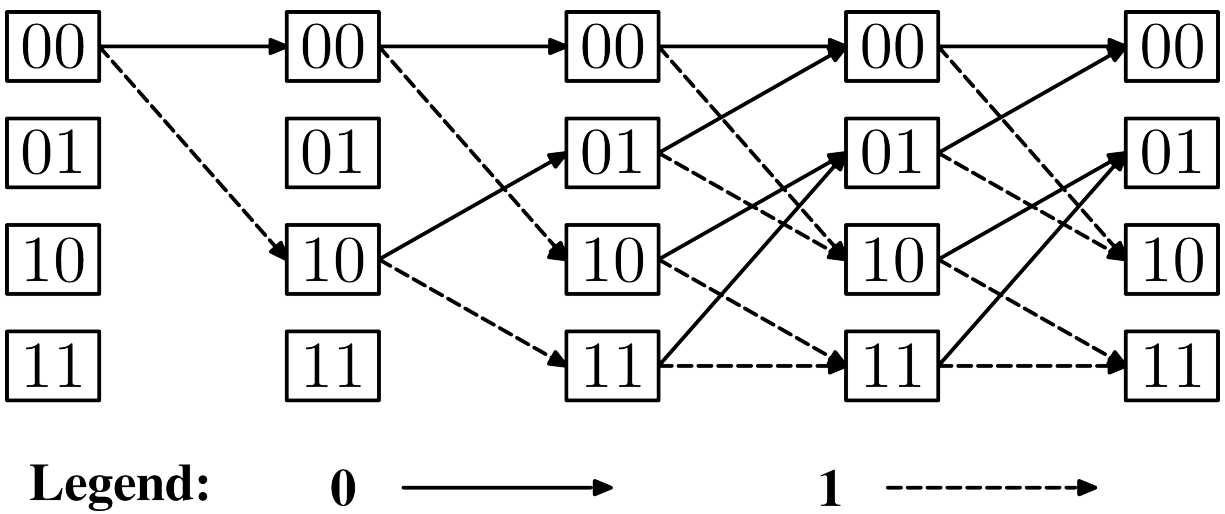


Structure analysis of the pipeline



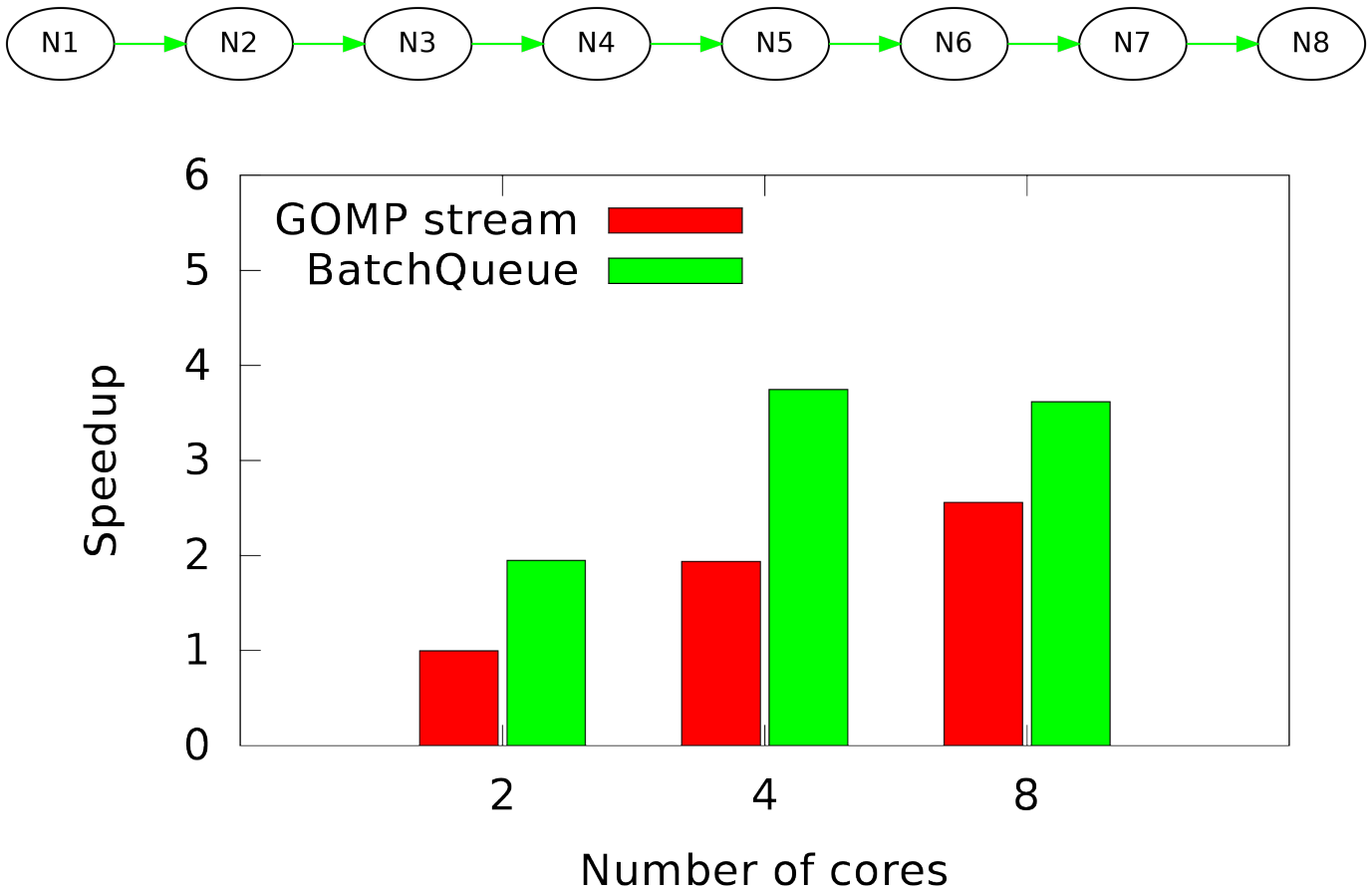
Trellis computation

Function: computation of the most likely CRC from a given analog signal
Source: Work from Alcatel-Lucent on AAC decoding
Particularity: fills a trellis with dependencies between columns



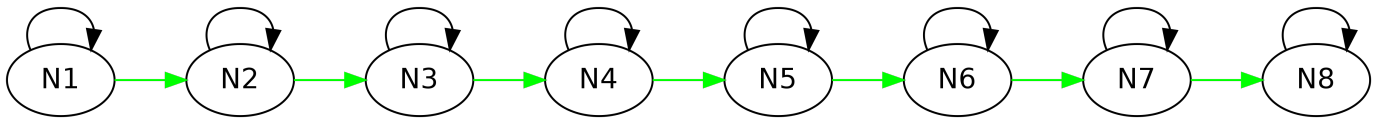
Trellis computation

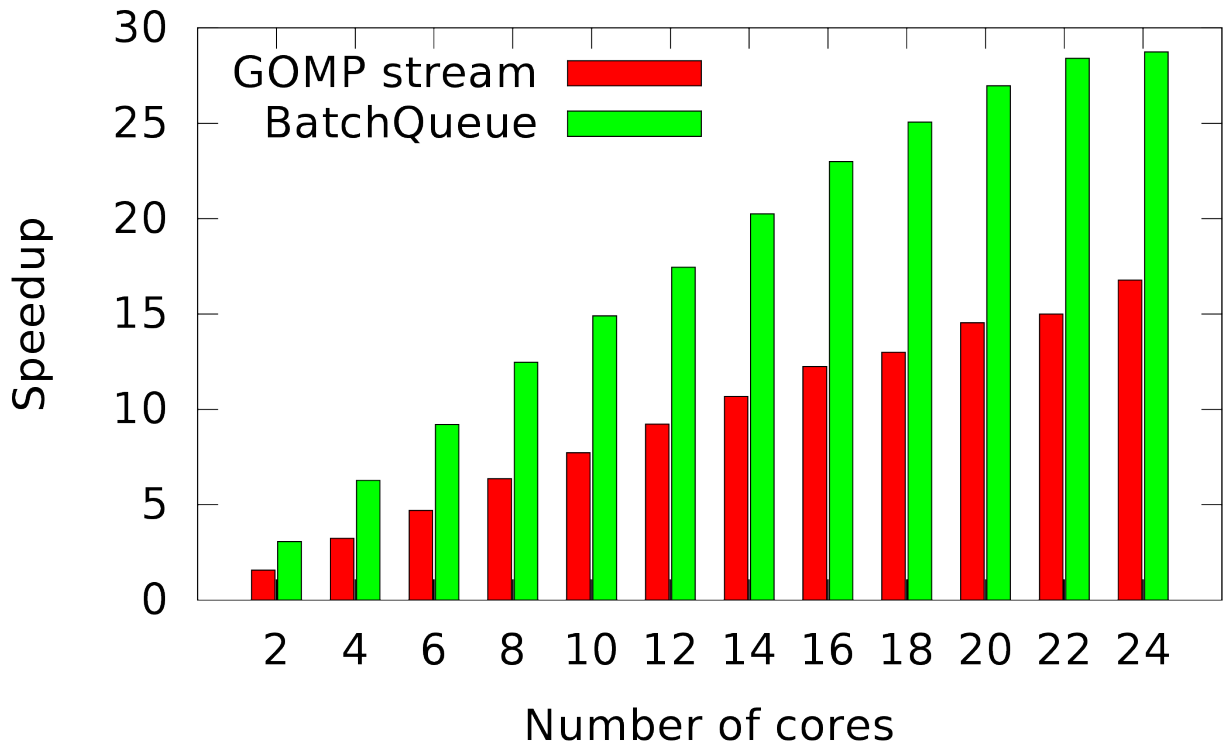
Function: computation of the most likely CRC from a given analog signal
Source: Work from Alcatel-Lucent on AAC decoding
Particularity: fills a trellis with dependencies between columns



Pipeline template

Function: template of code only parallelizable with pipeline parallelism
Particularity: backward dependencies between data units





Conclusion

- Optimized inter-core communication with BatchQueue:
  1. Tackle problem with memory consistency
     - \+ reduce the need for consistency
     - \+ avoid false sharing when accessing buffer
     - \+ prevent prefetch from creating false sharing
       ⇒ throughput improved up to a factor 2
  2. Minimize memory footprint
     - \+ low memory overhead
       ⇒ only one extra bit per queue to synchronize
- Automated usage of BatchQueue for pipeline parallelism:
  - \+ modifications transparent to applications using OpenMP
    ⇒ automatic selection of BatchQueue for linear streams
  - \+ speedup improved in applications up to a factor 2

Future work

Short term perspectives

- Improve interaction with scheduler to reduce spinning
- Fetch the status bit asynchronously using SMT + prefetch

Long term perspectives

- Support 1-to-N and N-to-1 communication
  $\Rightarrow$ create optimized algorithms for specialized cases
- Support N-to-N communication
  $\Rightarrow$ follow similar approach to make a cache friendly algorithm
- Use BatchQueue in other domains
  e.g.: offload some computation to a dedicated core
- Adapt dynamically communication algorithms in applications


J. Giacomoni, T. Mosely, and M. Vachharajani.
Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue.
In *Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008.


Leslie Lamport.
Specifying concurrent program modules.
ACM Trans. Program. Lang. Syst., 5(2):190–222, 1983.


P.P.C. Lee, T. Bu, and G. Chandranmenon.
A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring.
In *IPDPS '10: Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.


Thomas Preud'homme, Julien Sopena, Gaël Thomas, and Bertil Folliot.
Batchqueue: Fast and memory-thrifty core to core communication.
In *2010 22nd International Symposium on Computer Architecture and High Performance Computing*, pages 215–222. IEEE, 2010.


Y. Zhang, K. Ootsu, T. Yokota, and T. Baba.
Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs.
IAENG International Journal of Computer Science, 36, 2009.