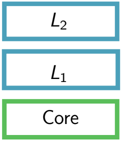
A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores

Marc Tchiboukdjian Vincent Danjean Thierry Gautier
Fabien Le Mentec Bruno Raffin

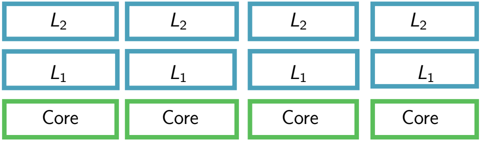
Shared Cache of Multicore Processors

1. One core with 2 cache levels



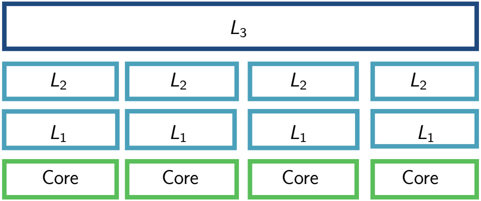
Shared Cache of Multicore Processors

1. One core with 2 cache levels
2. Multiple cores with private caches



Shared Cache of Multicore Processors

1. One core with 2 cache levels
2. Multiple cores with private caches
3. Multiple cores with private caches and **one shared cache**



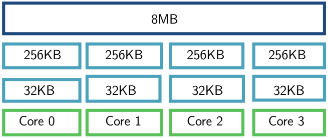
The sequential has an unfair advantage.

Can we still get linear speedup?

Scheduling for Efficient Shared Cache Usage

Schedule the computation so that
shared cache misses do not increase.

- with a **work stealing** scheduler allowing efficient dynamic load balancing
- for **parallel loops**: no dependencies between tasks



Xeon Nehalem E5530 (2.4Ghz)

Overview

Cache Efficient Work Stealing Scheduling for Parallel Loops

1. Standard Schedulers for Parallel Loops
2. New Scheduler Optimized for Shared Cache
3. Efficient Implementation of the Scheduler
4. Experiments

Parallel Loops

Examples of parallel loops

- OpenMP `#pragma omp parallel for`
- TBB `parallel_for`
- Cilk `parallel_for`
- Parallel STL `for_each` or `transform`

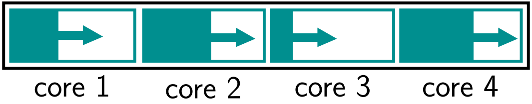
Problem characteristics

- Schedule n iterations on p cores
- Iterations can be processed independently
- Time to process one iteration can vary

Static Scheduling of Parallel Loops

Static Scheduling

- Allocate $\frac{n}{p}$ iterations to each core
- ex: OpenMP static scheduling



Characteristics

- **low overhead** mechanism
- **bad load balancing** if workload is irregular

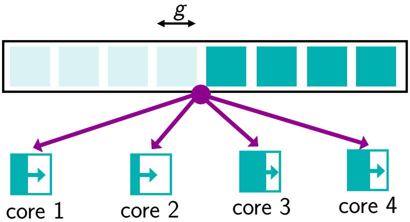
Dynamic Scheduling of Parallel Loops

OpenMP dynamic scheduling

- Allocate iterations in chunks of size g
- All chunks are stored in a centralized list
- Each thread remove a chunk from the list and process it

Characteristics

- **Good load balancing**
- **Contention** on the list
- **Chunk creation overhead**



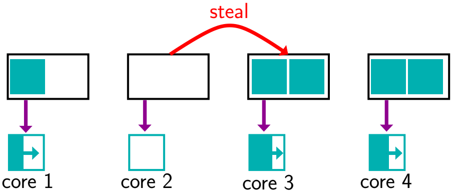
Scheduling Parallel Loops with Work Stealing

Work Stealing

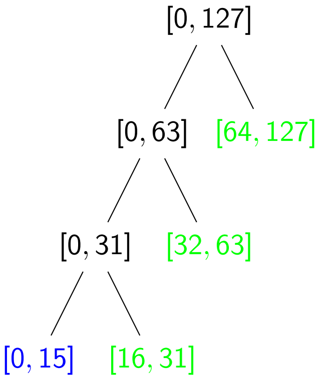
- Each thread has its own list of tasks (= chunks)
- If list is empty, steal tasks in a randomly selected list
- Binary tree of tasks to minimize number of steals:
  one steal ⇔ half of the iterations

Characteristics

- Good load balancing
- Contention is reduced
- **Task creation overhead**

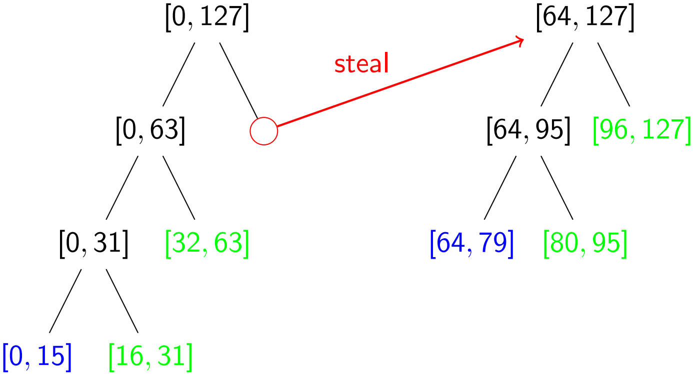


Scheduling Parallel Loops with Work Stealing



- terminated tasks
- ready tasks
- running tasks

Scheduling Parallel Loops with Work Stealing



- terminated tasks
- ready tasks
- running tasks

Parallel Loops with XKAAPI

XKAAPI

- Work stealing library
- **Tasks are created on a steal:** reduce task creation overhead
- Cooperative stealing: The victim stops working to answer work requests
- The victim can answer to multiple requests at a time

Characteristics

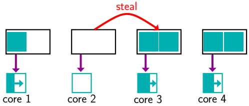
- Good load balancing
- Low overhead mechanism

```
typedef struct {
  InputIterator  ibeg;
  InputIterator  iend;
} Work_t ; // Task

void parallel_for (...) {
  while (iend != ibeg)
    do_work ( ibeg++ ) ;
} // no more work -> become a thief

void splitter ( num_req ) {
  i = 0 ;
  size = victim.iend - victim.ibeg ;
  bloc = size / ( num_req + 1 ) ;
  local_end = victim.iend ;
  while ( num_req > 0 ) {
    thief->iend = local_end ;
    thief->ibeg = local_end - bloc ;
    local_end -= bloc ;
    --num_req ;
  }
} // victim + thieves -> parallel_for
```

Overview

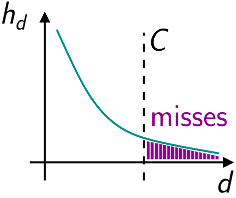
Cache Efficient Work Stealing Scheduling for Parallel Loops

1. Standard Schedulers for Parallel Loops



2. **New Scheduler Optimized for Shared Cache**
3. Efficient Implementation of the Scheduler
4. Experiments

Reuse Distances [Beyls and D'Hollander 01]



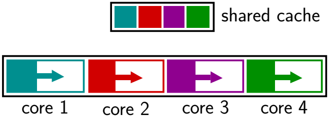
- Number of distinct elements accessed between two accesses to the same element.
- If first access, reuse distance is infinity.
- On a fully associative LRU cache of size C:
  reuse distance $\leq C \Rightarrow$ hit
  reuse distance $> C \Rightarrow$ miss
- h_d: number of accesses with a reuse distance d
- number of cache misses $M(C) = \sum_{d=C+1}^{\infty} h_d$

element access	A	B	A	B	B	C	B	A	C
reuse distance	∞	∞	2	2	1	∞	2	3	3
cache content	$\emptyset$	A	A,B	A,B	A,B	A,B	B,C	B,C	A,B

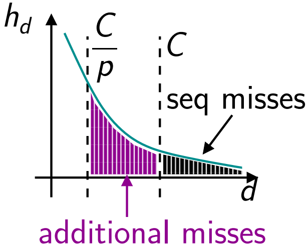
Shared Cache Misses with the Classic Schedule

- Cores work on elements far away
- Good temporal locality of the sequential algorithm

⇒ Cores work on distinct data



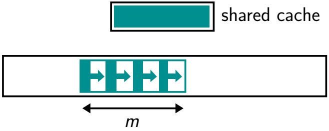
- To 1 access by a core corresponds $p-1$ accesses to distinct data by the other cores
- Reuse distance is multiplied by p: $h_d^{seq} = h_{p\cdot d}^{par}$
- Number of cache misses



$$M_{par}(C) = \sum_{d=C+1}^{\infty} h_d^{par} = \sum_{d=C+1}^{\infty} h_{d/p}^{seq} = \sum_{d=C/p+1}^{\infty} h_d^{seq} = M_{seq}(C/p)$$

A Shared Cache Aware Schedule

- Cores work at distance at most m
- $r(m)$ accesses to distinct elements in a window of size m
- The reuse distance is increased by at most $r(m)$: $h_d^{seq} = h_{d+r(m)}^{win}$
- Number of cache misses



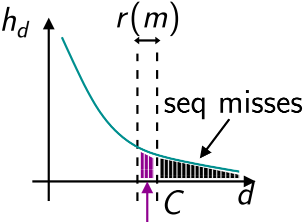
$$M_{win}(C) \leq \sum_{d=C+1-r(m)}^{\infty} h_d^{seq} = M_{seq}(C) + \sum_{d=C+1-r(m)}^{C} h_d^{seq}$$

Small Overhead over Sequential

- good sequential locality

$\Rightarrow$ $r(m)$ small

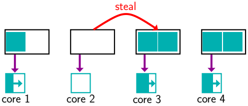
$\Rightarrow$ h_d small for large d



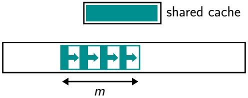
Overview

Cache Efficient Work Stealing Scheduling for Parallel Loops

1. Standard Schedulers for Parallel Loops



2. New Scheduler Optimized for Shared Cache



3. **Efficient Implementation of the Scheduler**
4. Experiments

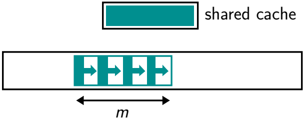
Implementing the Shared Cache Aware Schedule

Using a standard parallel for loop: StaticWindow

- Divide the iterations in n/m chunks of size m
- Each chunk is processed in parallel with a standard parallel for
- Two versions: pthread and XKAAPI

Pthread version

- each thread processes $1/p$ of a chunk
- wait on a barrier after each chunk



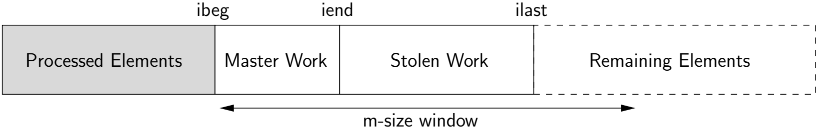
XKAAPI version

- each chunk is processed in parallel with a parallel for

Implementing the Shared Cache Aware Schedule

Optimized implementation using XKAAPI: SlidingWindow

- Processing iteration i enables iteration $i + m$
- Master thread is at the beginning of the sequence
- On a steal, the master can give work
    - In the interval $[ibeg, iend[$ like the other workers
    - In the interval $[ilast, ibeg + m[$ enabled since the last steal



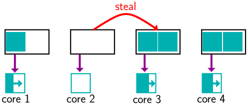
```
typedef struct {
  InputIterator  ibeg;
  InputIterator  iend;
} Work_t ; // Task
```

```
typedef struct {
  InputIterator  ibeg;
  InputIterator  iend;
  InputIterator  ilast;
} Master_Work_t ; // Master Task
```

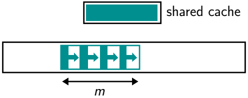
Overview

Cache Efficient Work Stealing Scheduling for Parallel Loops

1. Standard Schedulers for Parallel Loops



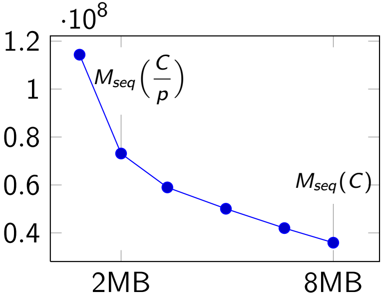
2. New Scheduler Optimized for Shared Cache



3. Efficient Implementation of the Scheduler

4. **Experiments**

Application: Isosurface Extraction

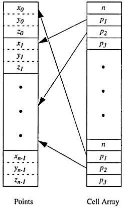


Isosurface extraction

- Common scientific visualization filter
- Memory bounded

Algorithm

- Iterate through all cells in the mesh
- Interpolate surface inside each cell
- **Cells close in the mesh share points**



Experiments

2 processors

- **Opteron**: 2 Dualcores with private L_1 and L_2 caches
- **Nehalem**: Quadcore with private L_1, L_2 caches and shared L_3

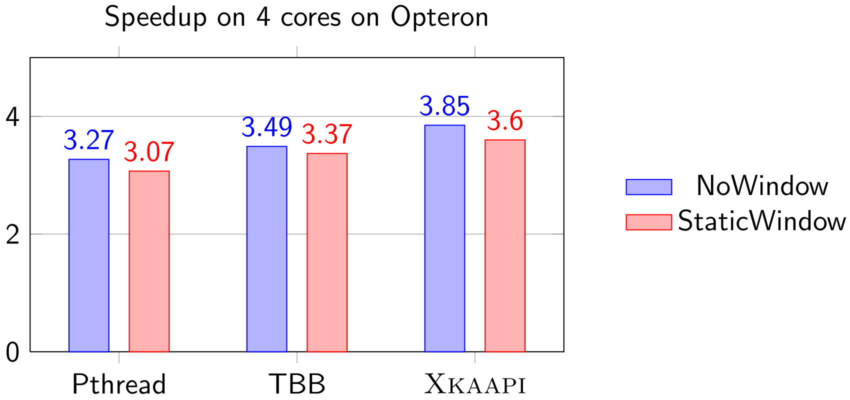
2 schedules

- **NoWindow**: classic schedule
- (Static or Sliding) **Window**: shared cache aware schedule

7 implementations

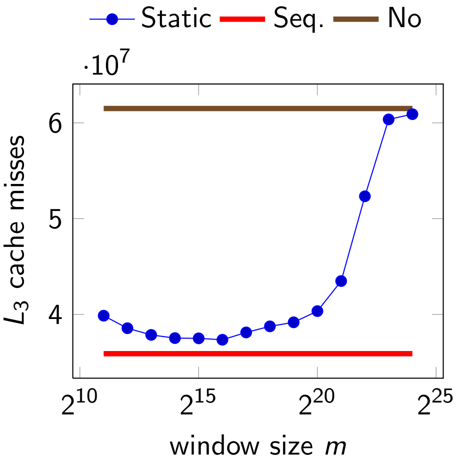
- **NoWindow**: Pthread, TBB and XKAAPI
- **StaticWindow**: Pthread, TBB and XKAAPI
- **SlidingWindow**: XKAAPI

Synchronization overhead



- Pthread < TBB < XKAAPI
- On Opteron: **no shared cache** ⇒ Window < NoWindow
  more synchronizations without gain in cache misses

Window Size m

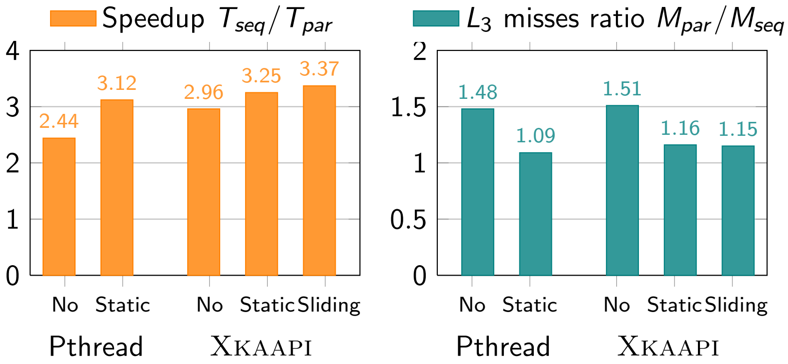


- On 4 cores of Nehalem
- Shared 8MB L_3 cache
- For small m: $M_{window} \approx M_{seq}(C)$
- $M_{no-window} \approx M_{seq}\left(\frac{C}{p}\right)$

$M_{no-window} = 6.15 \cdot 10^7$

$M_{seq}\left(\frac{C}{p}\right) = 7.13 \cdot 10^7$

Speedup and Cache Misses on Nehalem

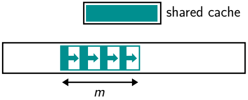


- 4 cores of Nehalem with 8MB of shared cache L_3
- Best performance: SlidingWindow

Conclusion

Shared Cache Aware Scheduler

- Shared cache aware scheduler for parallel loops
- Efficient implementation using work stealing
- *For application with good sequential locality*
  the window strategy is as good as if
  each core had its own copy of the L_3 cache



Future work

- Experiment with other applications
- Automatically find window size with reuse distance histogram
- Cache-oblivious version (cache size unknown)?