An Advanced Scheduler for Intervals

Master's Thesis

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

September 7, 2010

Executive Summary

- Advanced work-stealing scheduler for intervals
  - → Locality-aware scheduling using locality hints provided by the programmer
- Providing locality hints to intervals is optional
  - → Performance of locality-ignorant programs executed with new scheduler implementation comparable to original scheduler
- Locality hints improve runtime and cache hit and miss rates
  - → *Best locality* placement achieves up to 1.15× speedup
  - → Cache hits increase by 1.5× and cache misses decrease by 3.1×

Work-Stealing Intervals Scheduler

- Employs a fixed number of worker threads
- Each worker has local deque to maintain its own pool of ready intervals:
  - Puts and takes intervals to execute at the tail of its deque
  - When its deque is empty, tries to steal an interval from the deque's head of a victim worker chosen at random

Locality-Aware Intervals Scheduling

- Modern CMPs feature heterogeneous memory hierarchies:
  - Access times depend on which processor interval is running
  - May be better to run interval on one processor than another
- Locality-aware intervals can lead to improved performance:
  - Data sharing intervals running on the same processor perform prefetching of shared regions for one another
  - Running non-communicating intervals with high memory footprints on different processors reduces cache contention
- Current work-stealing intervals scheduler is locality-ignorant

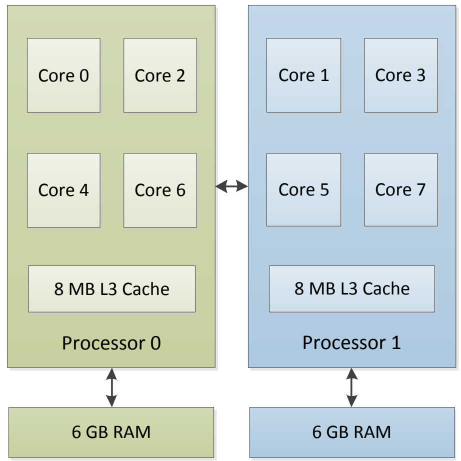
⇒ Introduce LASSI[1], a locality-aware scheduler for intervals

[1] The correct acronym would be LASI but we chose LASSI instead as we really enjoy drinking refreshing masala lassi ☺

Outline

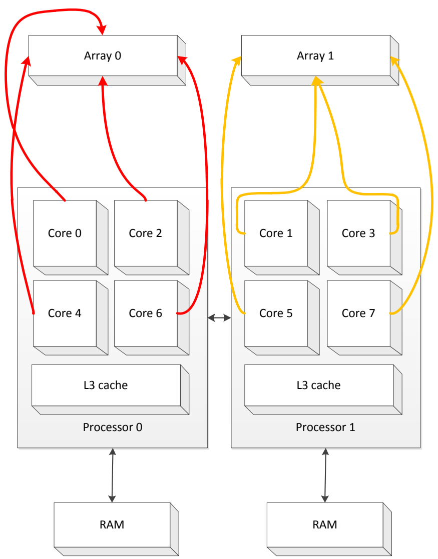
Cache Stress Test

- Multi-threaded locality-aware benchmark
- Randomly initializes two integer arrays of size 8 MB
- Binds 8 *Cache Stress* threads to each core
- Half of the threads work with array 0, the other half with array 1
- Each thread adds and multiplies all the elements of its array 100 times

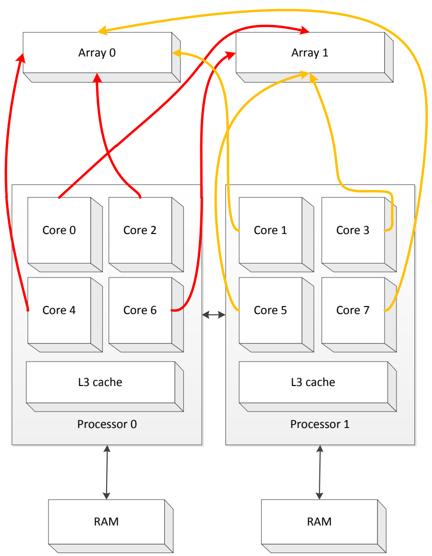


Intel Nehalem test system

Best and Worst Locality



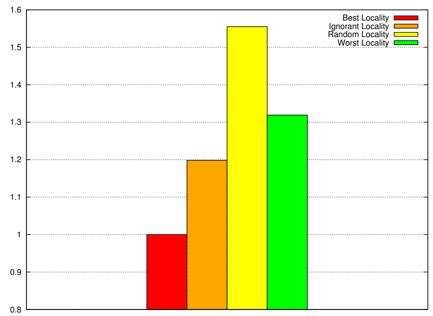
Best locality



Worst locality

Execution Times

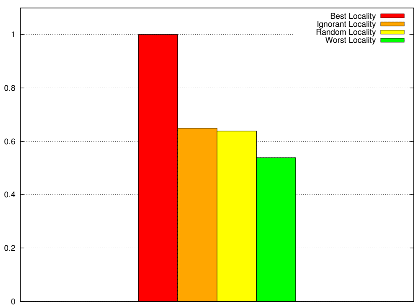
- *Best locality* variant: Sharing threads run on same processor → perform prefetching of array elements for each other
- Other variants: Threads compete for L3 caches
- *Best locality* has significant speedup of up to 1.55×



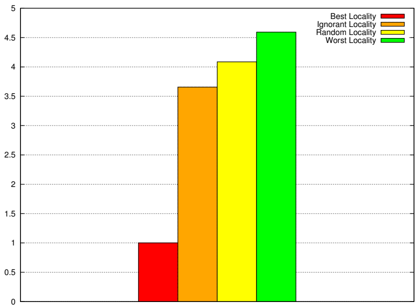
Execution times normalized to *best locality*

Cache Read Hits and Misses

- *Best locality* benchmark has between 1.5× and 1.8× more L3 cache read hits, and between 3.6× and 4.5× fewer read misses
- Cache read hits and misses normalized to the *best locality* implementation:



Cache read hits



Cache read misses

Outline

Locality-Aware Intervals API

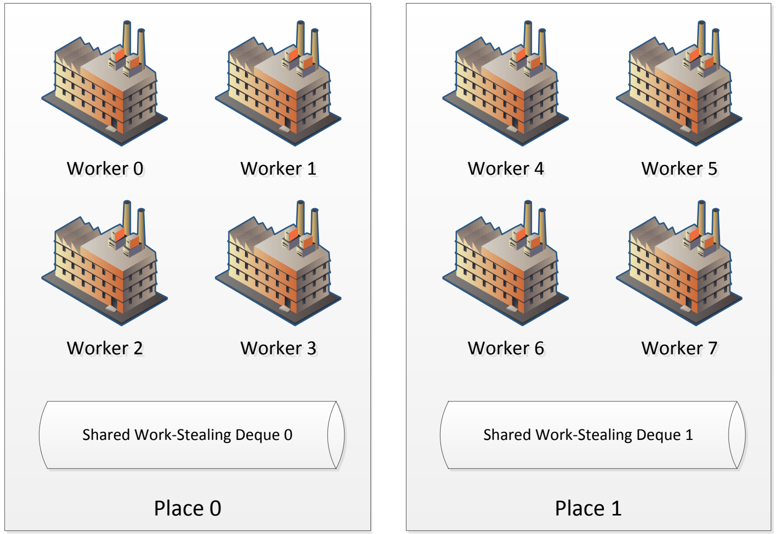
- Intervals are subtypes of abstract class `Interval`
- Specify the interval's locality when creating it
- Locality hints provided in the form of `PlaceID` objects
  → Assign the interval to the specified place
- If `PlaceID` is null, the interval is ignorant of its place
  → Assign the interval to a place in a round-robin fashion

```
public abstract class Interval extends WorkItem {
  public final PlaceID place;

  public Interval(Dependency dep, String name, PlaceID place) {
    this.place = place;
    // ...
  }

  // ...
}
```


Work-Stealing Places

- Traditional work-stealing scheduler designs: Every worker has local deque to maintain own pool of ready tasks
- LASSI uses *Work-Stealing Places* instead:
  - Each place has a fixed number of workers and a local deque
  - Workers of a place share its local deque
  - When the pool of a place is empty, its workers tries to steal a task from the pool of a victim place chosen at random

Intel Nehalem in Two-Processor Configuration



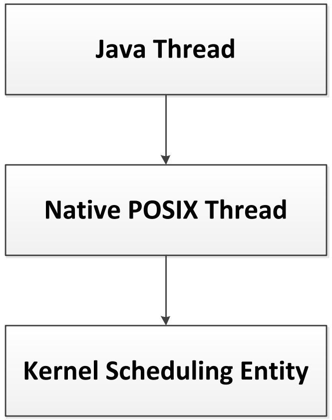
Work-Stealing Places used in our Intel Nehalem testing machine

Alternative Designs

- Other designs provide each worker with mailbox in addition to work-stealing deque:
  - Worker pushes work item onto both its deque and into the mailbox of the worker the item has affinity for
  - Worker tries to get work from its mailbox before stealing
  - Work items must be idempotent as they can appear twice
- Simplify by using a shared deque per *Work-Stealing Place*
- Will not impact scalability as long as the places are small
  - → Up to 8 workers: No significant difference between using separate deque for each worker or shared deque per place

Setting Core Affinity of Worker Threads

- 1-to-1 correspondence between Java and native threads
- Java Threads API does not expose ability to set the CPU or core affinity
- JNI library to bind workers to a core:
  - `pthread_self()` gets the native thread ID
  - `pthread_setaffinity_np()` sets core affinity of worker thread



Linux 1-to-1 thread mapping

Data Locality

Setting core affinity of threads only controls locality of work

→ No control over data locality

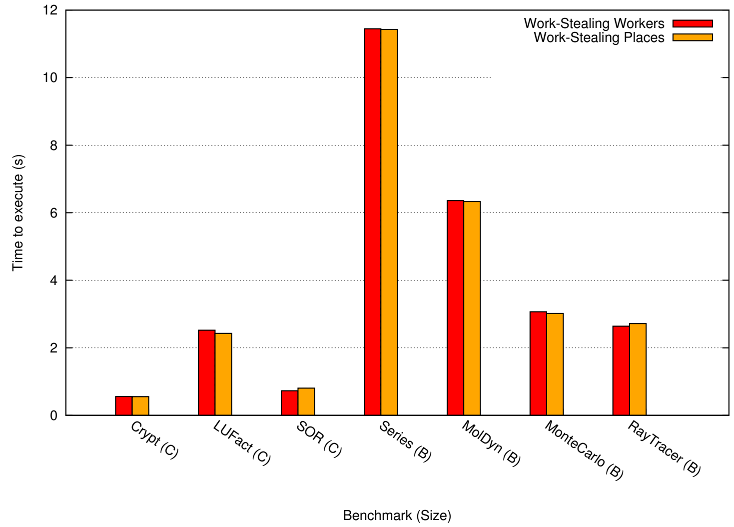
Java HotSpot VM: NUMA-aware allocator

- Provides automatic memory placement optimizations
- Relies on a hypothesis that thread allocating an object will be the most likely to use it
  - → Places it in the region local to the allocating thread
- Enabled by invoking the JVM with `-XX:+UseNUMA`

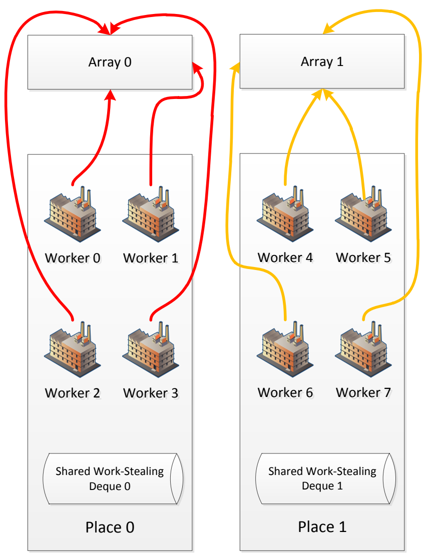
Outline

Java Grande Forum Benchmarks

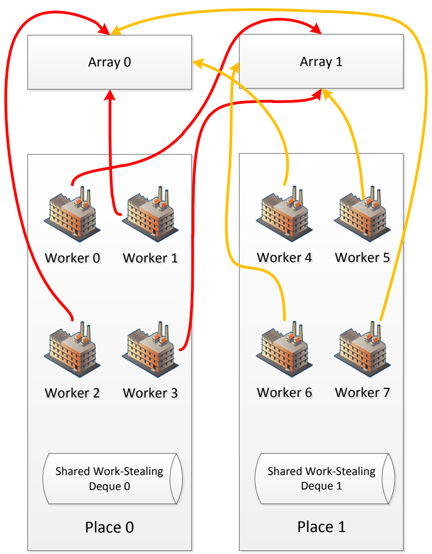
New scheduler implementation does not affect performance of existing locality-ignorant intervals applications:



Cache Stress Test: Best and Worst Locality



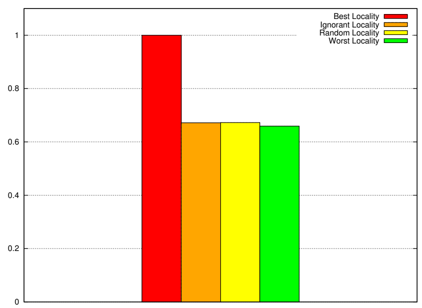
Best locality



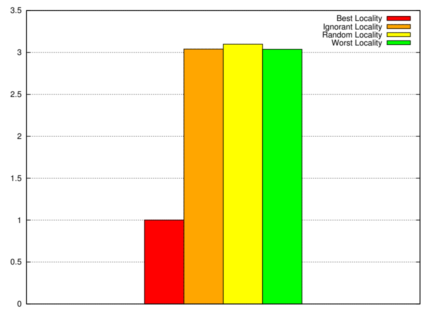
Worst locality

Cache Stress Test: Performance

- *Best locality* has speedup of up to 1.12×
- *Best locality* benchmark has up to 1.5× more L3 cache read hits and 3.1× fewer read misses:



Cache read hits normalized to *best locality*

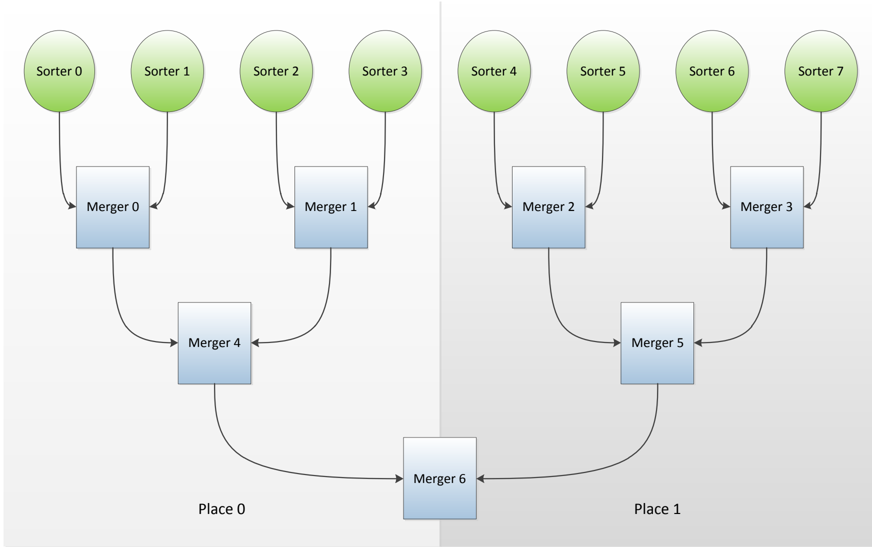


Cache read misses normalized to *best locality*

Merge Sort

- Uses divide-and-conquer to recursively sort 4 194 304 randomly initialized integer values
  - → Needs about 16 MB of memory
- Creates 8 192 sorter intervals per worker
- Each sorter randomly initializes array of size $4\,194\,304/(8 \times 8\,192)$ and sorts it sequentially
- Mergers merge two neighboring sorted arrays into one sorted array until all subarrays are merged into a single array

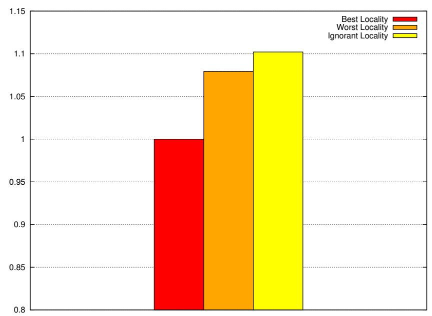
Merge Sort: Locality



Best locality

Merge Sort: Performance

- *Best locality* has speedup of up to 1.1×
- *Best locality* benchmark has up to 1.02× more L3 cache read hits and 1.07× fewer read misses:
  - → Rather small benchmark size and limited level of data sharing



Execution times normalized to *best locality*

Block Matrix Multiplication

Multiplies two random 2048×2048 matrices A and B using the recursion:

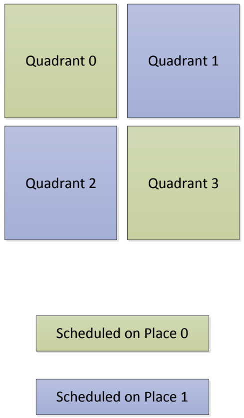
$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{pmatrix}$$

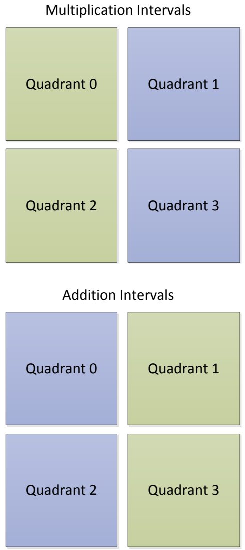
⇒ Matrix multiplication can be reduced to 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices

⇒ 8 multiplications can be calculated in parallel and when done, 4 additions can also be computed in parallel

Block Matrix Multiplication: Locality



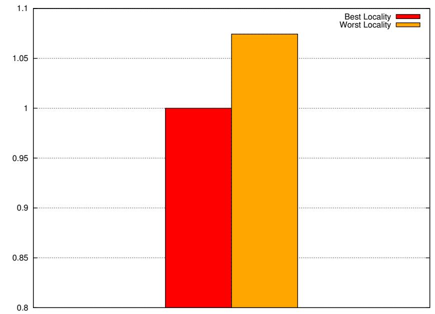
Best locality



Worst locality

Block Matrix Multiplication: Performance

- *Best locality* has speedup of up to 1.07×
- *Best locality* benchmark has up to 1.02× more L3 cache read hits and 1.06× fewer read misses:
  - → Rather small benchmark size and limited level of data sharing



Execution times normalized to *best locality*

Outline

Conclusions

- Introduced LASSI, a locality-aware scheduler for intervals
- *Work-Stealing Places* to support locality-awareness
- Performance of existing locality-ignorant programs comparable to the original scheduler implementation
- Scheduling data sharing intervals on the same processor:
  - → Prefetching of shared regions for one another
- Benchmarks do not test scheduling non-communicating intervals with high memory footprints on different processors

Future Work

- Improve API of *Work-Stealing Places* and locality-aware intervals
- Make underlying machine transparent to the user
- Extend *Work-Stealing Places* to co-locate tasks and data
- Avoid counter-productive steals
- Online contention detection to dynamically reduce or increase number of worker threads depending on system load

Questions?

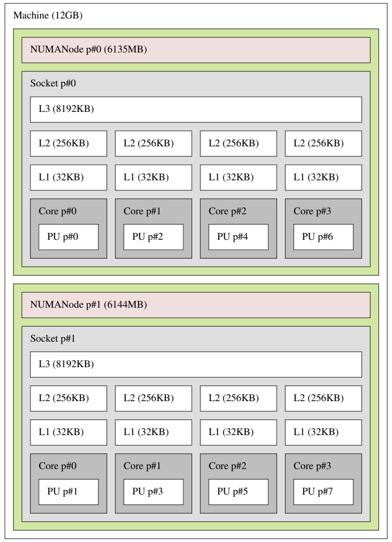
- Umut A. Acar et al. *"The data locality of work stealing"*. 2000.
- Robert D. Blumofe and Charles E. Leiserson. *"Scheduling multithreaded computations by work stealing"*. 1999.
- Robert D. Blumofe et al. *"Cilk: an efficient multithreaded runtime system"*. 1995.
- Yi Guo et al. *"SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems"*. 2010.
- Doug Lea. *"A Java fork/join framework"*. 2000.
- Nicholas D. Matsakis and Thomas R. Gross. *"Programming with Intervals"*. 2009.
- M. S. Squillante and E. D. Lazowska. *"Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling "*. 1993.
- Bratin Saha et al. *"Enabling scalability and performance in a large scale CMP environment"*. 2007.

Intel Nehalem Test Machine

- Intel Nehalem with 2 processors and 4 cores each
- Ubuntu 9.04 with kernel 2.6.29 patched to support perfmon2
- Sun Hotspot JDK 1.6.0_20 invoked with:

```
-server -Xmx4096M -Xms4096M
-Xss8M -XX:+UseNUMA
```

- perfmon2 tracks:
  - `UNC_LLC_HITS.READ`: Number of L3 cache read hits
  - `UNC_LLC_MISS.READ`: Number of L3 cache read misses



Alternative Work-Stealing Queues

- Performance of work-stealing schedulers in large part determined by the efficiency of the work queue
- Non-blocking queues employ atomic synchronization primitives such as Compare-and-Swap instead of mutual exclusion
- Current work-stealing queue of intervals uses mutual exclusion when trying to steal

⇒ Design and explore alternative non-blocking queues with the aim of improving work-stealing performance

Results

- Evaluate the performance of our queues with intervals implementations of various Java Grande Forum benchmarks
- None of the alternative work-stealing queues significantly improves performance on our test machines

Possible Reason

There is no noticeable difference between the speedup of work-stealing and a global shared work queue when not using more than 8 cores

Future Work

- Explore the performance of the different work-stealing queues on machines with more than 8 cores
- See how our work-stealing queues would benefit from using the steal-half algorithm of Hendler and Shavit
- Using a shared pool of arrays:
  - When the queue needs a larger array, allocate one of the appropriate size from the pool
  - Whenever it shrinks to a smaller array and does not need the larger array anymore, return it to the pool