

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

- 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 $1.15\times$ speedup
 - Cache hits increase by $1.5\times$ and cache misses decrease by $3.1\times$

- We implement and analyze an advanced scheduler for intervals which is designed for locality-aware scheduling using locality hints provided by the programmer.
- Providing locality hints is optional and the performance of locality-ignorant programs executed with the new scheduler implementation is comparable to that of the original scheduler.
- We studied the performance of our new scheduler implementation with benchmarks using data sharing intervals.

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

- The implementation of intervals makes use of a work-stealing scheduler similar to those found in Cilk or Java 7. The intervals scheduler employs a fixed number of threads called workers.
- Each worker has a local deque to maintain its own pool of ready intervals:
 - When assigning a new interval to a worker, the worker puts it onto the tail of its deque. And when obtaining work, it takes a ready interval from the tail of its deque to execute.
 - When a worker's deque is empty, it 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 LASSI¹, a locality-aware scheduler for intervals

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

- As modern chip multiprocessor systems feature a heterogeneous memory hierarchy where access times depend on which processor an interval is running, it may be more efficient to schedule an interval on one processor than another.
- Locality-aware intervals can lead to improved performance:
 - By scheduling data sharing intervals on the same processor they perform prefetching of shared regions for one another.
 - Scheduling non-communicating intervals with high memory footprints on different processors helps to reduce cache contention and potential cache capacity problems.
- The current implementation of the intervals library uses a locality-ignorant work-stealing scheduler to schedule ready-to-run intervals.
- Thus, we implement and analyze LASSI, a locality-aware scheduler for intervals.

Outline

1 Approach

2 Implementation

- Locality-Aware Intervals
- Work-Stealing Places

3 Performance Evaluation

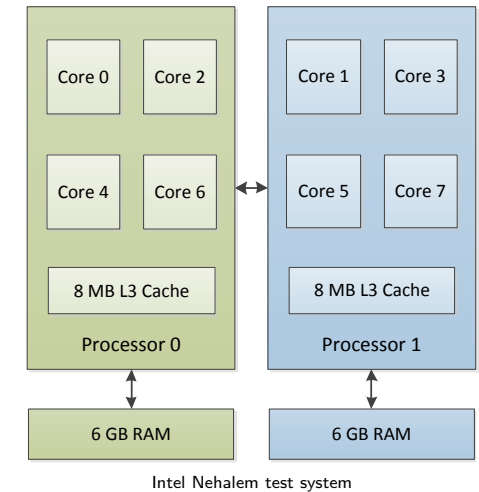
- Non-Locality Benchmarks
- Locality Benchmarks

4 Conclusions and Future Work

- Before starting with the implementation of the new scheduler, we implement a synthetic multi-threaded locality-aware benchmark called *Cache Stress Test*.
- This benchmark serves as a proof of concept for our plan to introduce locality-aware intervals.

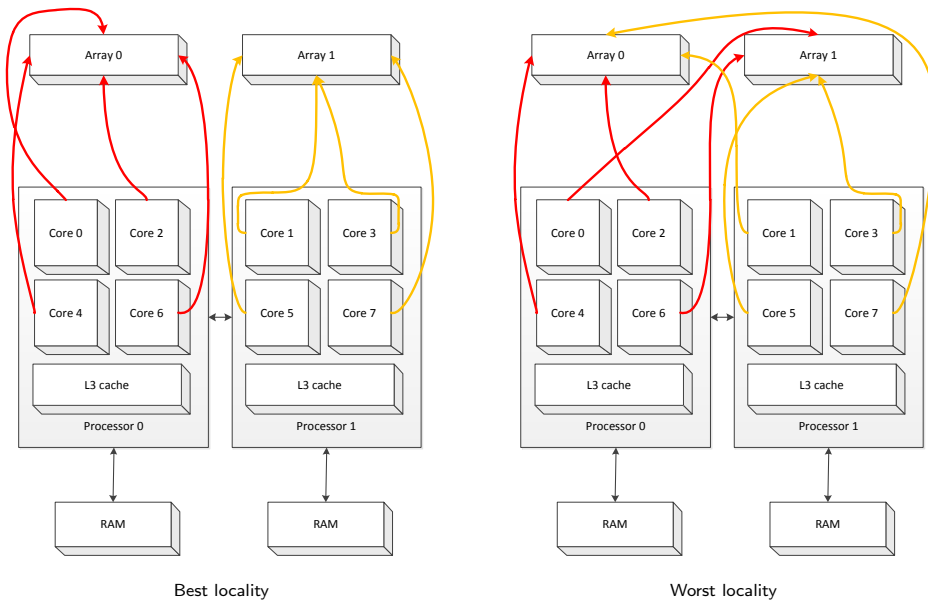
Cache Stress Test

- 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



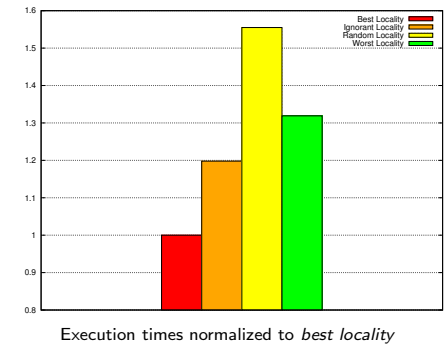
- We wrote our benchmark for “mafushi”, our Intel Nehalem system. “mafushi” has 2 processors with 4 cores each. The two 8 MB L3 caches are shared between all cores of the same processor.
- *Cache Stress Test* first randomly initializes two integer arrays of size 8 MB.
- Then the benchmark creates 8 *Cache Stress* threads per core with their affinity set to this specific core.
- One half of the threads operate on the elements of the first array and the other half operate on the elements of the second array.
- Each thread adds and multiplies all the elements of its respective array 100 times.

Best and 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 $1.55\times$



We implement several different variants of the *Cache Stress Test*, each having different locality properties.

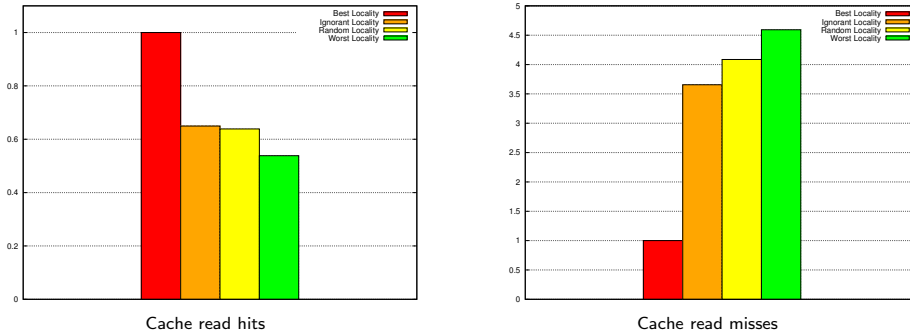
- In the *Best Locality* implementation all the threads working on the first array have affinity for a core on the first processor and all threads working on the second array have affinity for a core on the second processor.
- When using *Worst Locality*, half the threads with affinity for a core on the first processor work on the first array, and the other half work on the second array and vice versa.

Besides *Best* and *Worst Locality* we also implement variants with *Ignorant* and *Random Locality*.

- When we are using the *best locality* variant, we move all sharing threads onto the same processor which will perform prefetching of the array elements for each other.
- The exact opposite happens in the other variants: Threads compete for the L3 caches.
- The *best locality* implementation shows a significant speedup over the other locality benchmarks of 20 to 55 percent.

Cache Read Hits and Misses

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



Outline

1 Approach

2 Implementation

- Locality-Aware Intervals
- Work-Stealing Places

3 Performance Evaluation

- Non-Locality Benchmarks
- Locality Benchmarks

4 Conclusions and Future Work

- Compared to the other benchmarks, the *best locality* benchmark has between 50 and 80 percent more L3 cache read hits, and between 360 and 450 percent fewer cache read misses.
- The experiments confirmed our hypothesis and we decided to rewrite the intervals scheduler to support locality hints.

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;
        // ...
    }

    // ...
}
```

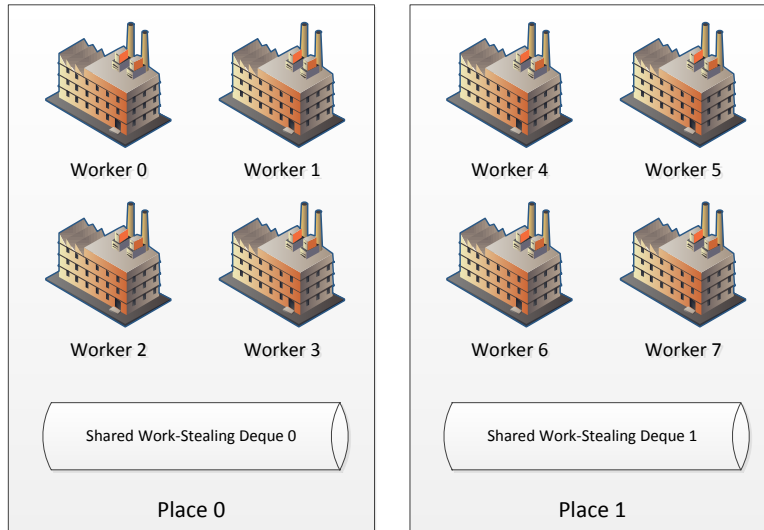
- Intervals are represented as subtypes of the abstract class `Interval`. We extend it to support locality hints.
- To make an interval locality-aware, the programmer has to specify the interval's locality when creating it.
- Locality hints are provided in the form of `PlaceID` objects. They specify which place the interval should be executed on.
- If the interval should be ignorant of its place, the programmer can provide `null` when creating it. This makes the scheduler to assign the interval to a place in a round-robin fashion.

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

- In traditional work-stealing scheduler designs, every worker has a local deque, to maintain its own pool of ready tasks from which it obtains work.
- LASSI uses *Work-Stealing Places* instead:
 - Each work-stealing place has a fixed number of workers and a local deque to maintain ready tasks.
 - The workers of a place share its local deque from which they obtain work.
 - When a worker finds that the pool of its place is empty, it 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

- The figure shows the work-stealing places used on our Intel Nehalem testing machine.
- Places are virtual: The mapping of physical units to places is performed by a concrete implementation of the abstract Places class.
- For our test machine we define a place for each processor.

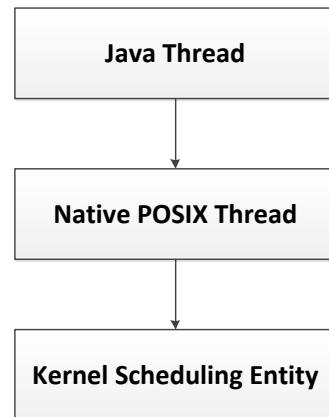
Alternative Designs

- 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

- Other locality-aware work-stealing schedulers provide each worker with a mailbox in addition to the work-stealing deque:
 - When creating a work item, a worker will push it onto both its deque and also into the tail of the mailbox of the worker that the interval has affinity for.
 - A worker will first try to obtain work from its mailbox before attempting to steal.
 - Because work items can appear twice, once in a mailbox and once in a deque, they have to be idempotent.
- We have decided to simplify our scheduler implementation by using a shared deque per *Work-Stealing Place*.
- This will not impact scalability as long as the places are not too large. We could show that up to 8 workers there is no significant difference between using a separate deque for each worker or a 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

- In recent Java Virtual Machines, threads are implemented with native threads. This means there is a 1-to-1 correspondence between Java and native threads.
- Unfortunately the Java Threads API does not expose the ability to set the CPU or core affinity despite numerous use cases where it would be beneficial such as improving cache and network performance or real-time applications.
- So to bind the workers to a specific core, we wrote a small JNI library.

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`

- By setting the core affinity of threads, we only control the locality of the work but we do not have control over data locality.
- In the Java HotSpot VM, the NUMA-aware allocator has been implemented to provide automatic memory placement optimizations for Java applications.
- To enable the NUMA-aware allocator, we invoke the JVM with the option `UseNUMA` when using LASSI.

Outline

1 Approach

2 Implementation

- Locality-Aware Intervals
- Work-Stealing Places

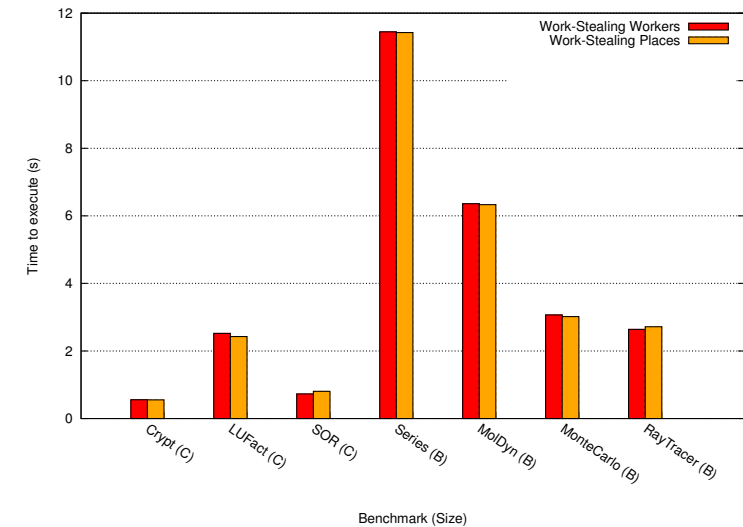
3 Performance Evaluation

- Non-Locality Benchmarks
- Locality Benchmarks

4 Conclusions and Future Work

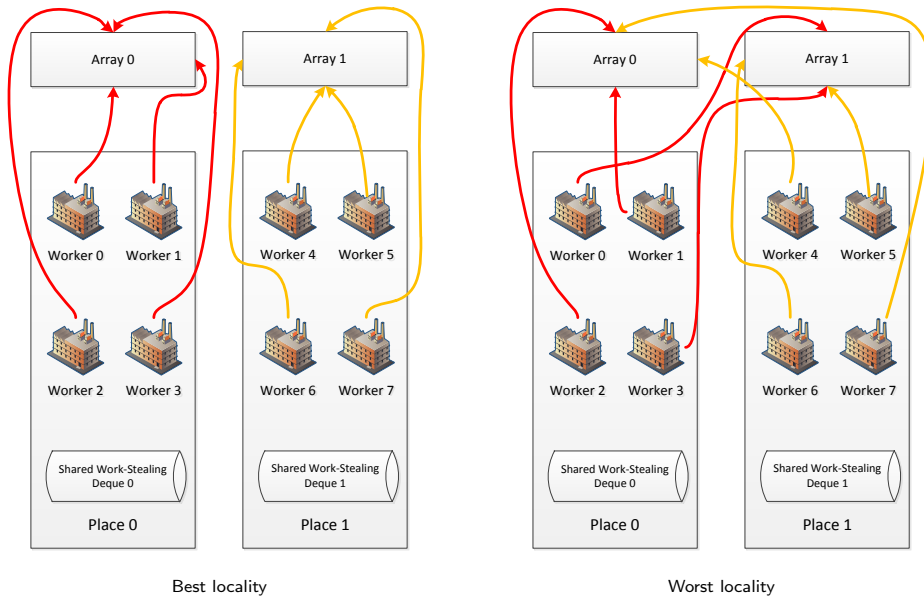
Java Grande Forum Benchmarks

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



- It is important that our new scheduler implementation does not affect the performance of existing locality-ignorant intervals applications. Thus, we run the locality-ignorant JGF benchmarks with our new scheduler implementation.
- As the figure shows, the performance of the locality-ignorant JGF benchmarks executed on LASSI is comparable to the original implementation.

Cache Stress Test: Best and Worst Locality



Best locality

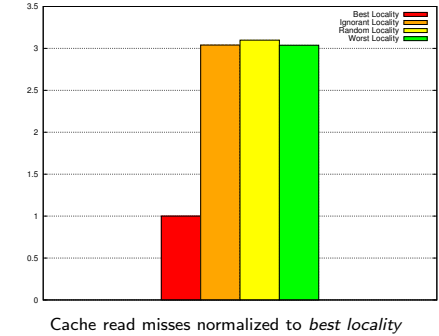
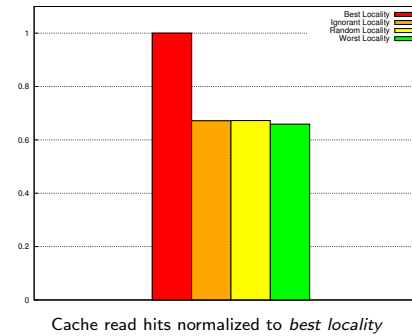
Worst locality

The *Cache Stress Test* benchmark was ported over from the threaded implementation.

- In the *Best Locality* all the intervals working on the first array have their locality set to *place 0* and all intervals working on the second array have their locality set to *place 1*.
- When using *Worst Locality* half the intervals with locality for *place 0* work on the first array, and the other half work on the second array and vice versa for the intervals with locality for *place 1*.

Cache Stress Test: Performance

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



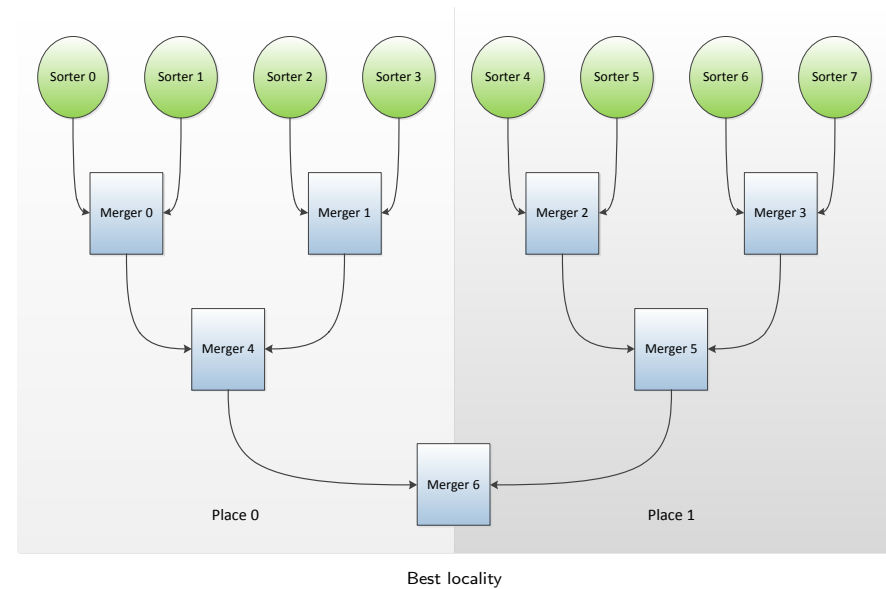
- When running the intervals implementations of the *Cache Stress Test* benchmarks, we observe similar behavior to the threaded versions. The *best locality* implementation is about 12 percent faster compared to the other locality benchmarks.
- The *best locality* benchmark has up to 50 percent more L3 cache read hits and 310 percent fewer cache read misses than the other benchmarks.

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

- The *Merge Sort* benchmark uses divide-and-conquer to recursively sort randomly initialized integer values.
- The integer values need about 16 MB of memory which is equal to the size of the last level caches of our test machine.
- *Merge Sort* first creates 8 192 sorter intervals per worker.
- Each sorter randomly initializes an array of the size mentioned on the slide and sorts it sequentially
- Merger intervals merge two neighboring sorted arrays into one sorted array of doubled size until all subarrays are merged into a single array.

Merge Sort: Locality

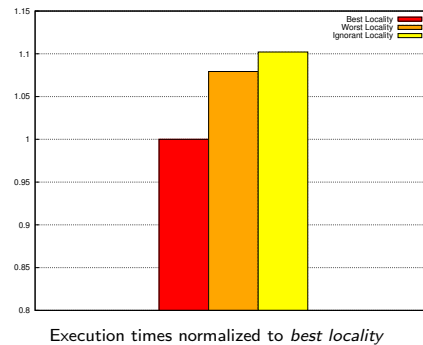


We implement several variants of the benchmark, each having different locality properties:

- The *best locality* implementation puts data sharing intervals onto the same *place*. Half the sorter intervals have locality for *place 0* and the other half have their locality set for *place 1*. Merger intervals are scheduled on the same place as their left predecessors in the tree hierarchy.
- When using *Worst Locality*, the sorter intervals have the same locality as in the *best locality* implementation. The merger intervals set their locality to the other place as their left predecessor in the tree hierarchy.
- In the *Ignorant Locality* variant the sorter and merger intervals have no locality set.

Merge Sort: Performance

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



- The figure illustrates the execution times normalized to that of the *best locality* implementation. The *best locality* implementation provides a speedup of up to 10 percent compared to the other locality benchmarks.
- The difference between the number of cache read hits and misses is small: The *best locality* variant has just about 2% more cache read hits and 7% less cache read misses than the *worst* and *ignorant locality* variants:
 - This is mainly because of the rather small benchmark size and limited level of data sharing between the intervals.

Block Matrix Multiplication

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

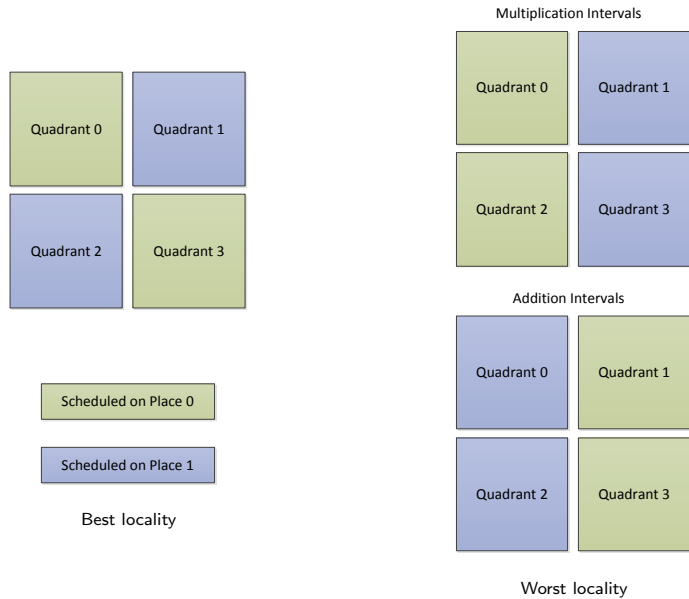
$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

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

- ⇒ 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

- The *Block Matrix Multiplication* benchmark multiplies two random 2048×2048 matrices. Each matrix needs about 16 MB of memory.
- By using the recursion, the matrix multiplication can be reduced to 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices.
- The multiplications can be calculated in parallel and when they are done, the additions can also be computed in parallel.

Block Matrix Multiplication: Locality

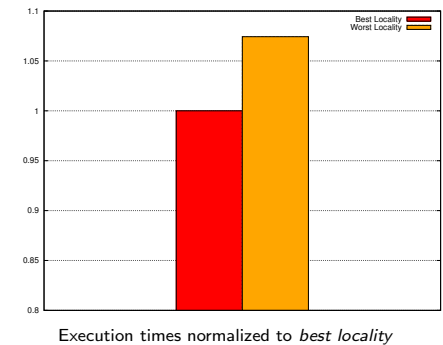


We implement two variants of the benchmark, *best locality* and *worst locality*:

- The *best locality* benchmark runs all addition and multiplication intervals of the submatrices of quadrants 0 and 3 in *place 0* and the ones of the quadrants 1 and 2 in *place 1*. This way the places are able to share their local L3 cache in an efficient way.
- The *worst locality* benchmark runs the multiplication and addition intervals in different places, destroying cache locality.

Block Matrix Multiplication: Performance

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



- The figure depicts the execution time of the *worst locality* implementation normalized to that of the *best locality* implementation. The *best locality* implementation shows a speedup of about 7%.
- The difference between the number of cache read hits and misses is little: The *best locality* variant has just about 2% more cache read hits and 6% less cache read misses than the *worst locality* variant.
 - The main reason for this is the rather small benchmark size and limited level of data sharing between the intervals.

Outline

1 Approach

2 Implementation

- Locality-Aware Intervals
- Work-Stealing Places

3 Performance Evaluation

- Non-Locality Benchmarks
- Locality Benchmarks

4 Conclusions and Future Work

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









- In this thesis, we introduced LASSI, a locality-aware scheduler for intervals using locality hints provided by the programmer.
- Instead of employing work-stealing workers, LASSI uses *Work-Stealing Places* to provide locality-awareness.
- The performance of existing locality-ignorant programs run with LASSI is comparable to the original scheduler implementation.
- By scheduling data sharing intervals on the same processor they perform prefetching of shared regions for one another which improves performance. Our experimental results show a speedup as well as an increase in L3 cache read hits and a decrease in cache read misses.
- Our benchmarks do not test how scheduling non-communicating intervals with high memory footprints on different processors helps to reduce cache contention and potential cache capacity problems.

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?

- A possible area of future work would be to improve the API.
- Places have to be manually configured for each system. This should be automated by making the hardware transparent to the user.
- LASSI depends on the heuristics of the NUMA-aware allocator implemented in Java HotSpot VM to provide automatic memory placement optimizations. Further research could be done in extending *Work-Stealing Places* to co-locate tasks and data.
- Load balancing across work-stealing places can lead to counter-productive stealing. Future work should avoid such steals.
- Worker threads share their assigned core with other processes in the system. Online contention detection could be used to dynamically reduce or increase the number of worker threads.

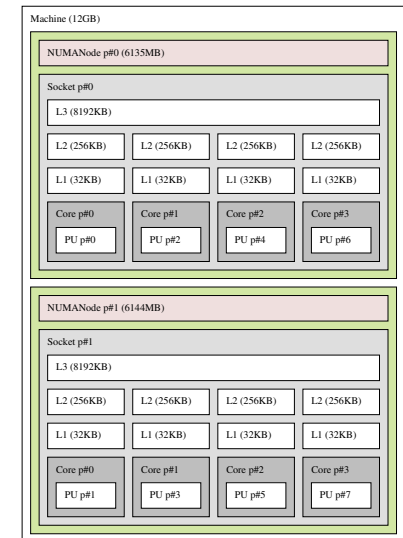
-  Umut A. Acar et al. “*The data locality of work stealing*”. 2000.
-  Robert D. Blumofe and Charles E. Leiserson. “*Scheduling multithreaded computations by work stealing*”. 1999.
-  Robert D. Blumofe et al. “*Cilk: an efficient multithreaded runtime system*”. 1995.
-  Yi Guo et al. “*SLAW: a scalable 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_LL_C_HITS.READ: Number of L3 cache read hits
 - UNC_LL_C_MISS.READ: Number of L3 cache read misses



- The performance results are obtained on an Intel Nehalem system with 2 processors and 8 cores, running Ubuntu with kernel patched to support perfmon2.
- The JVM used is Sun Hotspot which is invoked with the listed parameters.
- Besides the runtime of the benchmarks, we are mostly interested in the count of cache hit and miss events. Since the last level cache is part of the uncore, we have to use the uncore PMU to count L3 cache events.
- The execution times and cache event counts reported are the average of the 3 best benchmark iterations from 10 separate invocations.

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

- The performance of work-stealing schedulers is in a large part determined by the efficiency of their work queue implementations.
- In the non-blocking work-stealing scheduler, the queues are implemented with non-blocking synchronization. That is, instead of using mutual exclusion, it uses atomic synchronization primitives such as Compare-and-Swap.
- The current work-stealing queue of intervals however uses mutual exclusion when trying to steal.
- Thus, as a separate effort, we design and explore alternative non-blocking queue implementations with the aim to improve work-stealing performance.

- Our hypothesis was that we could improve the performance of the intervals scheduler with non-blocking work-stealing queues.
- We evaluated the performance of our alternative queues by using the intervals implementations of various Java Grande Forum benchmarks.
- None of the work-stealing queues we developed significantly improves work-stealing performance on the machines we had to test them with.
- Apart from the more complex implementation in comparison to the original *Work-Stealing Deque*, a possible reason for this could be the rather small number of cores of our benchmark machines. We could not find any significant difference between the runtimes of the Java Grande Forum benchmarks when using the different implementations – whether we use work-stealing or a single shared work queue, we get almost the same results.

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

- Given the results of the performance evaluation, a direction for future research would be to explore the performance of the different work-stealing queue implementations on machines featuring more than 8 cores.
- It may also be interesting to see how our work-stealing queues would benefit from using the steal-half algorithm of Hendler and Shavit.
- Another optimization would be working with a shared pool of arrays. With the shared pool implementation, whenever the queue needs a larger array, it allocates one of the appropriate size from the pool, and whenever it shrinks to a smaller array and does not need the larger array anymore, it can return it to the pool.