Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

An Advanced Scheduler for Intervals

Thomas Weibel, <weibelt@ethz.ch>

Advisors: Nicholas D. Matsakis and Prof. Thomas Gross

August 2010

Informatik
Computer Science

Swiss Federal Institute of Technology Zürich
Laboratory for Software Technology
Department of Computer Science
8092 Zürich, Switzerland

Abstract

Intervals are a new, higher-level primitive for parallel programming allowing programmers to directly construct the program schedule. They are under active development at ETH Zürich as part of the PhD research of Nicholas D. Matsakis.

The intervals implementation in Java uses a work-stealing scheduler where a worker running out of work tries to steal work from others. The scope of this thesis is to improve the performance of the intervals scheduler.

We implement and analyze an advanced scheduler for intervals. It is designed for locality-aware scheduling using locality hints provided by the programmer. Instead of employing work-stealing workers, our scheduler groups workers into *Work-Stealing Places*. 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. Locality-aware intervals are added to their preferred place once they are ready for scheduling.

Providing locality hints to intervals is optional and the performance of locality-ignorant programs executed with the new scheduler implementation is comparable to that of the original scheduler.

•

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 [4], 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.

Acknowledgement

I would like to thank my thesis advisor, Nicholas D. Matsakis, for his support and guidance throughout this work. Thank you for numerous hours of discussions, valuable inputs, and advice.

My thanks also go to Zoltán Majó who provided me with helpful insight into the memory hierarchy of the NUMA architecture and supported me when I had questions with the testing machine.

Further, I want to thank Prof. Dr. Thomas R. Gross for giving me the opportunity to carry out my thesis in his group.

Finally, I thank all my family and friends. It is because of the constant support of my loved ones that I was able to complete the work successfully.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Intervals [48] are a new, higher-level primitive for multi-threaded programming allowing programmers to directly construct the program schedule. They are under active development at ETH Zürich as part of the PhD research of Nicholas D. Matsakis [46].

The intervals implementation in Java uses a work-stealing scheduler where a worker running out of work tries to steal work from others. The scope of this thesis is to improve the performance of the intervals scheduler.

1.1. Intervals

Traditional primitives for synchronizing multi-threaded programs, such as semaphores and barriers, are low-level and dangerous to use. They are prone to errors, especially deadlocks and race conditions, and require careful attention to implementation details to achieve good performance.

Intervals are a higher-level alternative that make parallel programming more secure while maintaining the flexibility and efficiency of threads. When using intervals, programmers create lightweight tasks and order them using *happens before* relations [30]. Programmers need not specify when tasks should block or acquire a lock. Instead they define when a task should execute in relation to other tasks, and what locks it should hold during execution. It is the duty of the runtime system to make this schedule pass.

The intervals API supports arbitrary *happens before* relations making the model very flexible. Intervals can be used to emulate existing thread primitives [48], but they can also be used to create program schedules for which no standard primitives exist, for example peer-to-peer synchronization.

Intervals can be extended to support both runtime and static checks for errors like data race protection [49] and deadlocks [45]. An error in one task prevents other, dependent tasks from executing [47].

1.1.1. Model

Intervals are first-class objects in the programming language which stand for the slice of program time used to execute a parallel task. They are structured in a tree with the root representing the entire program execution.

The conceptual model for intervals consists of points in time ordered by a *happens before* relation. In the model, an interval `i` consists of a pair of points called the start and end point: `i.start` and `i.end`. Programmers may define ordering constraints by adding *happens before* edges in between the start or end points of different intervals. An edge `p1` $\rightarrow$ `p2` indicates that the point `p1` must occur before the point `p2`. It also guarantees that any memory writes which *happen before* `p1` must be visible to `p2`. This is in line with the Java Memory Model [41], and the semantics of volatile fields and synchronized sections.

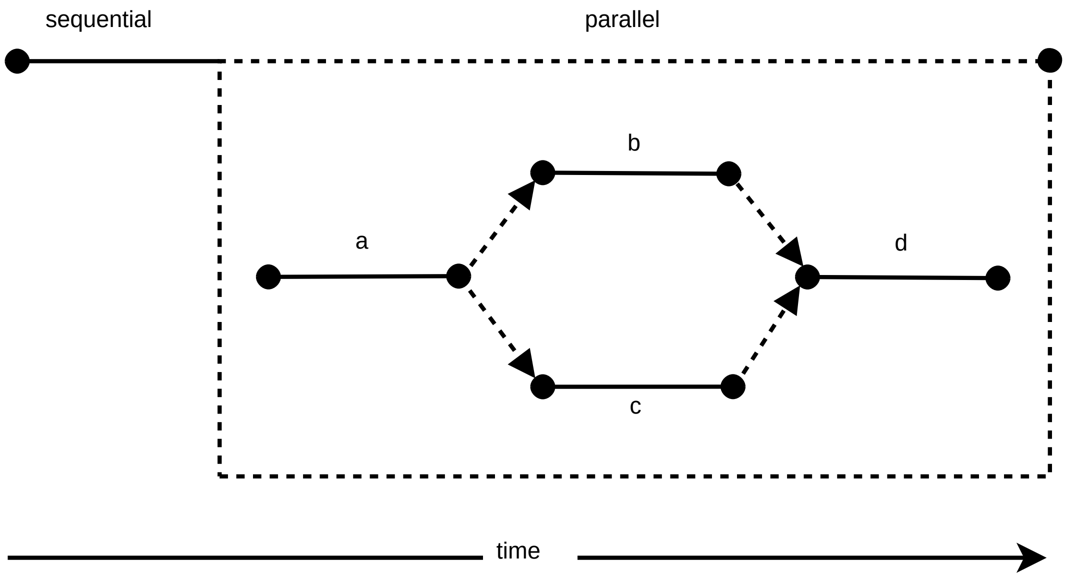


Figure 1.1.: Example interval graph: Showing an interval and its subintervals `a`, `b`, `c` and `d`.

An interval can have one or more locks. The intervals runtime automatically acquires those locks before the interval's start point occurs. When the task is completed and the end point of the interval has occurred, the acquired locks are released again.

When an interval executes, it begins by calling the sequential `run()` method. The `run()` method can either execute the task directly or create a number of subintervals to solve the task in parallel.

The intervals model can be depicted as a graph, as shown in Figure 1.1. The graph pictures a single interval with four subintervals, `a`, `b`, `c` and `d`. The start and end points of each interval are drawn as opaque circles. The subintervals of an interval are enclosed in a dashed box which is left out for leaf intervals.

The dashed edges connecting different points indicate *happens before* relations added by the programmer. For our example program, the end of `a` *happens before* the start of `b` and `c`, and the ends of `b` and `c` both *happen before* the start of `d`.

1.1.2. Java API

To create an interval, programmers have to subclass the abstract class `Interval` (Listing 1.1) and redefine its `run()` method. Besides the abstract `run()` method, `Interval` features final fields to access the start and end point as well as the parent interval.

```java
public abstract class Interval {
  public final Interval parent;
  public final Point start;
  public final Point end;

  protected abstract void run();
}
```

Listing 1.1: `Interval`: Base class for all intervals

Listing 1.2 shows the code used to construct the graph presented in Figure 1.1.

```java
public class ExampleInterval extends Interval {
  public ExampleInterval(Dependency dep, String name) {
    super(dep, name);
  }

  protected void run() {
    // Task
  }

  public static void main(String[] args) {
    Intervals.inline(new VoidInlineTask() {
      public void run(Interval start) {
        Interval a = new ExampleInterval(start, "a");
        Interval b = new ExampleInterval(start, "b");
        Interval c = new ExampleInterval(start, "c");
        Interval d = new ExampleInterval(start, "d");

        Intervals.addHb(a, b);
        Intervals.addHb(a, c);
        Intervals.addHb(b, d);
        Intervals.addHb(c, d);
        Intervals.schedule();
      }
    });
  }
}
```

Listing 1.2: Code to construct the sample interval graph shown in Figure 1.1

Creating Intervals

To start program execution, the programmer has to create a new child of the root interval, for example by using an inline interval. Inline intervals execute a task during the current interval and do not return until the task has completed.

Lines 11 – 24 create an inline interval by providing an anonymous task class with the redefined `run()` method. The interval has four subintervals, `a`, `b`, `c` and `d`. They are created on Lines 13 – 16 and are normal, non-blocking intervals.

Scheduling Intervals

Newly constructed intervals become eligible for execution once the `schedule()` method is invoked, as shown on Line 22 in Listing 1.2. This gives the user the opportunity to construct any required dependencies or perform other initialization. For example, adding the edge `a` $\rightarrow$ `b` on Line 18 would be unsafe if `b` could begin immediately, as it would be possible that `b.start` had already occurred before the call to `addHb()` could add the new dependency.

The runtime automatically invokes `schedule()` when the `run()` method of an interval returns.

1.2. Work-Stealing Scheduler

The implementation of intervals for Java makes use of a work-stealing scheduler similar to those found in Cilk [7, 18], Java 7 [31, 33, 34, 32], Intel Threading Building Blocks [59, 11], or Microsoft Task Parallel Library [35] but extended to support locks and happens before edges.

A work-stealing scheduler employs a fixed number of threads called workers. Each worker has a local double-ended queue, or deque, to maintain its own pool of ready tasks from which it obtains work. When a worker finds that its pool is empty, it steals a task from the pool of a victim worker chosen at random.

To obtain work, a worker takes the ready task from the tail of its deque and executes it. If the task terminates, the worker goes back to the tail of its deque to take another task upon which it can work. When assigning a new task to a worker, the worker puts the newly ready task onto the tail of its deque. Thus, as long as a worker's deque is not empty, the worker manipulates its deque in a LIFO (stack-like) manner.

When a worker tries to obtain work by taking a task from the tail of its deque and it finds that it is empty, the worker becomes a thief. It picks a victim worker at random and attempts to obtain work by removing the task at the head of the victim's deque. If the victim's deque is empty, then the thief picks another victim worker and tries again until it finds a victim whose deque it not empty. At which point the thief continues to work on the stolen task as described above. Since steals take place at the head of the victim's deque, stealing operates in a FIFO manner.

Accessing the run queues at different ends offers several advantages [18]:

- It reduces contention by having owner and thieves working on opposite sides of the deque.
- Recursive divide-and-conquer algorithms generate “large” tasks early. Thus, the older stolen task is likely to further provide more work to the stealing worker.
- Stealing a task also migrates its future workload, which helps to increase locality.

The assignment of tasks to workers for execution is done in a provably efficient manner [7, 6].

1.3. Overview

In Part I of the thesis we implement and analyze an advanced scheduler for intervals. It is designed for locality-aware scheduling using locality hints provided by the programmer. Instead of using work-stealing workers, our scheduler groups workers into *Work-Stealing Places*.

In the non-blocking work-stealing algorithm, the deques are implemented with non-blocking synchronization [4]. The current deque implementation of intervals however uses mutual exclusion when trying to steal. As a separate effort, we design and explore alternative non-blocking queue implementations with the aim to improve work-stealing performance in Part II.

Part I.

Locality-Aware Work-Stealing

Chapter 2

Introduction

The current implementation of the intervals library uses a locality-ignorant work-stealing scheduler to schedule ready-to-run tasks. In this thesis we introduce LASSI, a locality-aware scheduler for intervals.[1]

2.1. Locality-Aware Intervals Scheduling

In chip multiprocessor systems it may be more efficient to schedule a task on one processor than another. As modern CMPs feature a heterogeneous memory hierarchy where access times depend on which processor an interval is running, 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.

When using locality-ignorant work-stealing we cannot fully exploit the heterogeneous memory hierarchy of CMPs for our benefit. Thus, we implement and analyze LASSI, a locality-aware scheduler for intervals. LASSI is designed for locality-aware scheduling using locality hints provided by the programmer. Instead of employing work-stealing workers, it groups workers into *Work-Stealing Places*.

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. Locality-aware intervals are added to their preferred place once they are ready for scheduling.

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

Providing locality hints to intervals is optional and the performance of locality-ignorant programs executed with the new scheduler implementation is comparable to that of the original scheduler.

We study the performance of LASSI with benchmarks using data sharing intervals. Our experimental results show that *best locality* placement of intervals can achieve up to 1.15× speedup over *worst* or *ignorant locality* placement. Cache hits can be increased by up to 1.5× and cache misses can be reduced by up to 3.1× for the benchmarks and platform studied in this thesis.

2.2. Overview

Chapter 3 describes our approach in evaluating locality-aware scheduling. Chapter 4 explains the implementation of LASSI. The chapter presents the locality-aware intervals API, introduces *Work-Stealing Places*, and shows how worker threads are bound to specific processing units, e.g. cores. In Chapter 5 we describe the locality benchmarks and analyze their results. Chapter 6 puts our research in the context of related work. In Chapter 7 we conclude and summarize our research, and give some ideas for future work.

Chapter 3

Approach

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: If a multi-threaded benchmark with best possible locality has better performance and fewer last-level cache misses than the same benchmark with another or no specific locality, we should be able to see the same effect when porting the benchmark to a locality-aware implementation of intervals. Hence, we know whether it makes sense to design a locality-aware intervals scheduler.

We wrote our benchmark for the Intel Nehalem system described in Appendix A.2. The system has 2 processors with 4 cores each. Every core has its separate level 1 and 2 caches, but the per-processor 8 MB level 3 cache is shared between all cores of the same processor. The methodology we use to run the benchmark and measure its results is presented in Section 5.1.

Cache Stress Test first randomly initializes two integer arrays of size 2 097 144, i.e. the size of each array is about 8 MB. This is equal to the size of the last level cache per processor. Then the benchmark creates 8 *Cache Stress* threads per core with their affinity set to this specific core. Overall, we create 64 threads, bound to the 8 cores in groups of 8 threads. To bind threads to a specific core, we use the affinity library introduced in Section 4.3.

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.

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

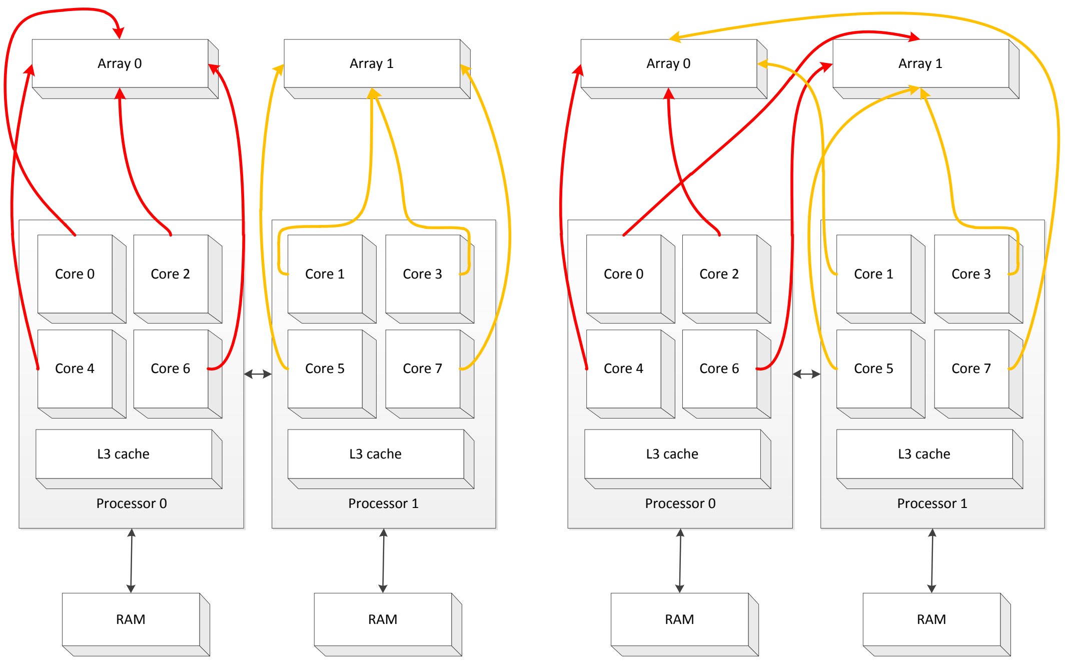
Best Locality: All the threads working on the first array have affinity for a core on the first processor and all threads working on the second array have affinity for a core on the second processor.

Ignorant Locality: The threads are not bound to any specific core, i.e. they are *ignorant* of their locality.

Random Locality: The affinity of the threads is set to a *random* core.

Worst Locality: Half the threads with affinity for a core on the first processor work on the first array, and the other half works on the second array and vice versa.

Figure 3.1 illustrates the core affinities of the threads for the *best* and *worst locality*.



(a) Best Locality **(b)** Worst Locality

Figure 3.1.: Multi-threaded *Cache Stress Test* with *best* and *worst locality*

As the name already gives away, the main idea behind the *Cache Stress Test* benchmark is to stress the cache and provoke cache prefetching and contention.

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. That is, they help to obtain and maintain the frequently used array elements in the local L3 cache.

The exact opposite happens in the other variants: Threads compete for the L3 caches and overwrite each other's entries. Instead of reading from the processor's local cache, threads either have to read from the other processor's cache or the memory subsystem. Reading from those places is much more expensive than reading from the local L3 cache. Table 3.1 cites Levinthal [37] and gives rough approximations for the memory access times on our test system. The latencies depend on the core and uncore frequencies, memory speeds, BIOS settings, number of DIMMs, and so forth.

Data Source	Latency
L3 cache hit, line unshared	$\sim$ 40 cycles
L3 cache hit, shared line in another core	$\sim$ 65 cycles
L3 cache hit, modified in another core	$\sim$ 75 cycles
Remote L3 cache	$\sim$ 100 – 300 cycles
Local DRAM	$\sim$ 60 ns
Remote DRAM	$\sim$ 100 ns

Table 3.1.: Memory access times on the Intel Nehalem processor

Table 3.2 shows the execution times and the speedups over the sequential algorithm for the different locality variants. As expected, the implementation with *best locality* is the fastest and provides the largest speedup.

	Runtime (in seconds)	Speedup (over sequential)
Best Locality	3,327	7,69
Ignorant Locality	3,985	6,42
Random Locality	5,175	5,83
Worst Locality	4,389	4,94
Sequential Implementation	25,571	1

Table 3.2.: Multi-threaded *Cache Stress Test* execution times and speedups over the sequential implementation

In Figure 3.2 we illustrate the execution times normalized to that of the *best locality* implementation. The *best locality* implementation shows a significant speedup over the other locality benchmarks of 1.2× – 1.55×.

Table 3.3 lists the number of L3 cache read hits and misses. In Figure 3.3 they are shown normalized to the measurements of the *best locality* implementation.

Compared to the other benchmarks, the *best locality* benchmark has between 1.5× and 1.8× more L3 cache read hits, and between 3.6× and 4.5× fewer L3 cache read misses.

	L3 Cache Read Hits	L3 Cache Read Misses
Best Locality	2 577	219
Ignorant Locality	1 674	800
Random Locality	1 646	894
Worst Locality	1 387	1 005

Table 3.3.: Multi-threaded *Cache Stress Test* L3 cache read hits and misses (rounded to the nearest million)

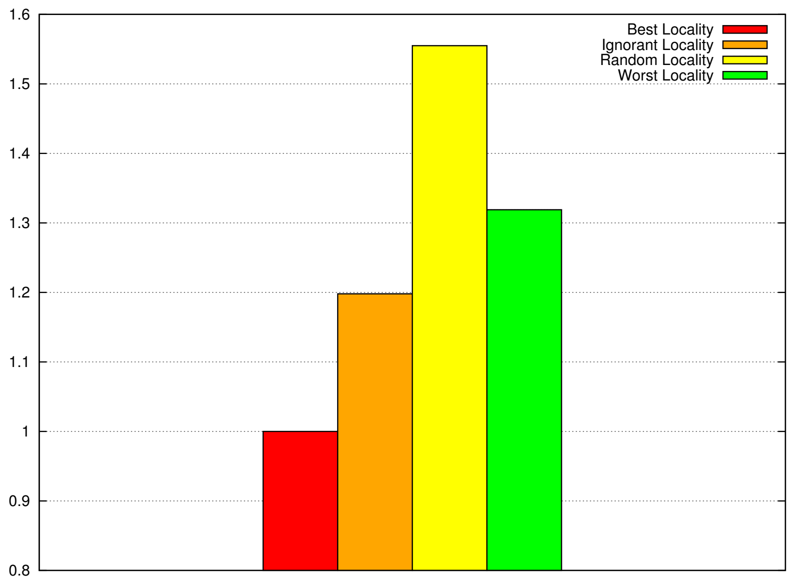
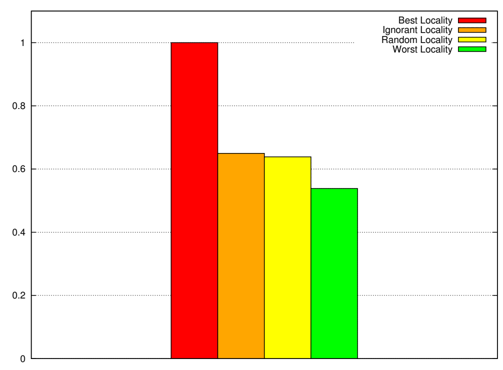
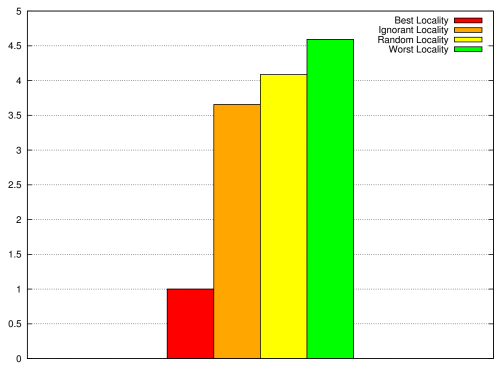


Figure 3.2.: Multi-threaded *Cache Stress Test* with execution times normalized to *best locality*



(a) L3 Cache Read Hits



(b) L3 Cache Read Misses

Figure 3.3.: Multi-threaded *Cache Stress Test* with L3 cache read hits and misses normalized to *best locality*

Our experiments confirmed the hypothesis that the locality of threads matters. Thus, we decided to rewrite the intervals scheduler to support locality hints provided by the programmer. Chapter 4 describes the implementation of the locality-aware scheduler for intervals, LASSI. In Chapter 5 we evaluate the performance of LASSI with a variety of benchmarks.

Chapter 4

Implementation

In this chapter we describe LASSI, our new implementation of the intervals scheduler. It is designed for locality-aware scheduling using locality hints provided by the programmer. Instead of using work-stealing workers, LASSI groups workers into *Work-Stealing Places.*

The API of locality-aware intervals is introduced in Section 4.1. Section 4.2 presents the idea and implementation of work-stealing places. It also describes the locality-aware scheduling policy. LASSI implements each worker as a separate Java thread. For locality-aware scheduling we need to bind every worker thread to a separate core. In Section 4.3 we show how we can set the core affinity of Java threads.

4.1. Locality-Aware Intervals API

Intervals are represented as subtypes of the abstract class `Interval`. To make an interval locality-aware, the programmer has to specify the interval's locality when creating it. We extend the abstract `Interval` class to support locality hints:

```java
public abstract class Interval extends WorkItem {
  public final PlaceID placeId;

  public Interval(Dependency dep, String name, PlaceID placeID) {
    this.placeID = placeID;

    // ...
  }

  public PlaceID getPlaceID() {
    return placeID;
  }

  // ...
}
```

Listing 4.1: Locality-aware `Interval` class

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 (Section 4.2).

The `PlaceID` class (Listing 4.2) encapsulates an integer ID and name. We are using a class for the place ID instead of an integer due to type safety.

The number of places is fixed at the time an intervals program is launched: There is no construct to create a new place and we do not want programmers to create their own place IDs. Thus, the `PlaceID` class is abstract and programmers have to use the `Config` class (Listing 4.3) to get a `PlaceID` object.

```
public abstract class PlaceID {
  public final int id;
  public final String name;

  public PlaceID(int id, String machine) {
    this.id = id;
    this.name = machine + "-place-" + id;
  }

  public String toString() {
    return name;
  }
}
```

Listing 4.2: `PlaceID` class

We introduce the `Config` class to simplify switching to another machine as the current scheduler implementation cannot discover places automatically. `Config` features an instance of the `Places` class which is described in Section 4.2. Programmers can use it to get place IDs, e.g. `Config.places.getPlaceID(0)`.

```
public class Config {
  // Creation method to simplify switching to another machine
  public static final Places places = new NehalemPlaces();
}
```

Listing 4.3: `Config` class

4.2. Work-Stealing Places

A work-stealing scheduler employs a fixed number of threads called workers. In traditional work-stealing scheduler designs, every worker has a local double-ended queue, or 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.

Figure 4.1 shows the work-stealing places used on our Intel Nehalem testing machine (Appendix A.2).

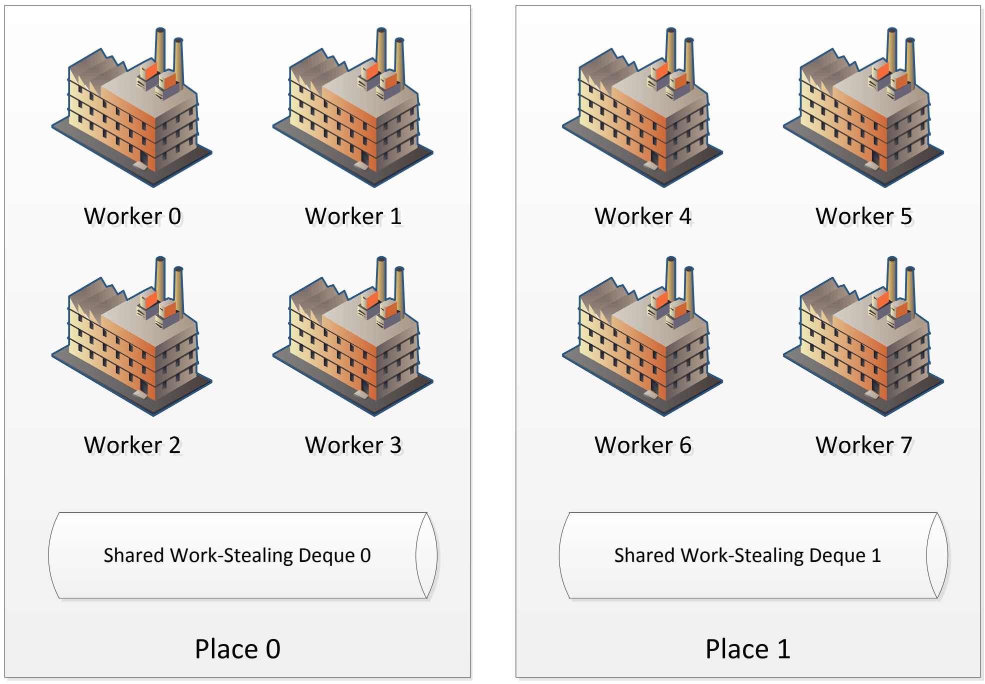


Figure 4.1.: *Work-Stealing Places* used in our Intel Nehalem testing machine (Appendix A.2)

Other locality-aware work-stealing schedulers use a more complex design. Acar, Blelloch, and Blumofe [1] for example provide each worker with a mailbox in addition to the work-stealing deque: “A mailbox is a FIFO queue of pointers to work items that have affinity for the worker. So when creating a work item, a worker will push it onto both the deque, as in normal work-stealing, and also onto the tail of the mailbox of the worker that the interval has affinity for. Now 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.” Idempotent intervals are introduced in Section 10.2.

However, we have decided to simplify our scheduler implementation by using a shared deque per *Work-Stealing Place*. We believe that this would not impact scalability as long as the places are not too large. We could show in Section 11.4 that up to 8 workers there is no significant difference between using a separate deque for each worker or a shared deque per place.

4.2.1. Places Implementation

Places are virtual: The mapping of physical units to places is performed by a concrete implementation of the abstract `Places` class shown in Listing 4.4.

```
 1 public abstract class Places {
 2   protected static class PlaceIDImpl extends PlaceID {
 3     public PlaceIDImpl(int id, String machine) {
 4       super(id, machine);
 5     }
 6   }
 7
 8   public final String name;
 9   public final int length;
10   public final int unitsLength;
11   protected final PlaceID[] placeIDs;
12   protected final int[][] places;
13   protected final int[] units;
14
15   public Places(String name, PlaceID[] placeIDs, int[][] places,
16       int[] units) {
17     this.name = name;
18     this.placeIDs = placeIDs;
19     this.places = places;
20     this.units = units;
21     this.length = places.length;
22     this.unitsLength = units.length;
23   }
24
25   public PlaceID getPlaceID(int id) {
26     return placeIDs[id % placeIDs.length];
27   }
28
29   public int[] get(int id) {
30     return places[id % length];
31   }
32
33   public int getUnit(int id) {
34     return units[id % unitsLength];
35   }
36 }
```

Listing 4.4: Abstract `Places` class

The abstract `Places` class provides a concrete implementation of the `PlaceID` class (Lines 2 – 6) to its subclasses.

Work-Stealing Places have a couple of public fields and methods. `name` is the name of the place, `length` contains the number of places and `unitsLength` saves the number of overall processing units the machine has. With `getPlaceID()` we can get a certain place ID. `get()` returns the IDs of the physical units belonging to a certain place. By calling `getUnit()` with the logical unit ID, we get the physical unit ID.

An example of a concrete place mapping is given in Listing 4.5. The class `NehalemPlaces` defines the places for our Intel Nehalem test machine according to its memory hierarchy. As Figure 4.2 depicts, this system includes two processors with four cores each. The machine has 12 GB RAM and both processors have a direct connection to half of the memory space. While every core has its separate level 1 and level 2 caches, the per-processor 8 MB level 3 cache is shared between all cores of the same processor.

```
 1  public class NehalemPlaces extends Places {
 2    private static String name = "nehalem";
 3    private static PlaceID[] placeIDs = { new PlaceIDImpl(0, name),
 4        new PlaceIDImpl(1, name) };
 5    private static int[][] places = { { 0, 2, 4, 6 },
 6        { 1, 3, 5, 7 } };
 7    private static int[] units = { 0, 2, 4, 6, 1, 3, 5, 7 };
 8
 9    public NehalemPlaces() {
10      super(name, placeIDs, places, units);
11    }
12  }
```

Listing 4.5: Places configuration for Intel Nehalem in a two-processor configuration

We define a place for each processor (Lines 3 – 4). Place 0 consists of the physical units 0, 2, 4, and 6 (Line 5) and the physical units 1, 3, 5, 7 belong to place 1 (Line 6). Line 7 defines the mapping of the virtual units to the physical processing units.

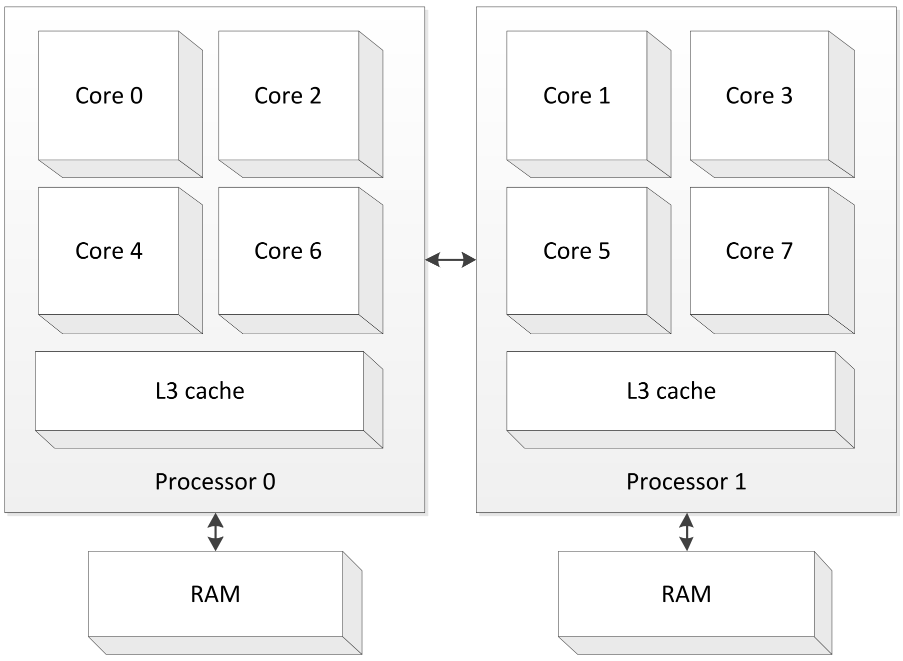


Figure 4.2.: Intel Nehalem in a two-processor configuration

4.2.2. Scheduling Policy

When an interval is ready to run, we first get its place ID. If the place ID is `null`, i.e. the interval is locality-ignorant, we either enqueue it at the place of the currently active worker or add it at a place in a round-robin fashion. If a place ID is given, then the interval is locality-aware and we enqueue it at the specified place.

4.3. Setting Core Affinity of Worker Threads

In recent Java Virtual Machines, threads are implemented with native threads. A Java program using threads is no different from a native program using threads, i.e. a Java thread is just a native thread belonging to a JVM process. This means there is a 1-to-1 correspondence between Java and native threads. When using the GNU C library on Linux, native threads are implemented with the NPTL (Native POSIX Threads Library). NPTL is also a 1-to-1 implementation, meaning that each thread maps to a kernel scheduling entity. Figure 4.3 illustrates this 1-to-1 thread mapping.

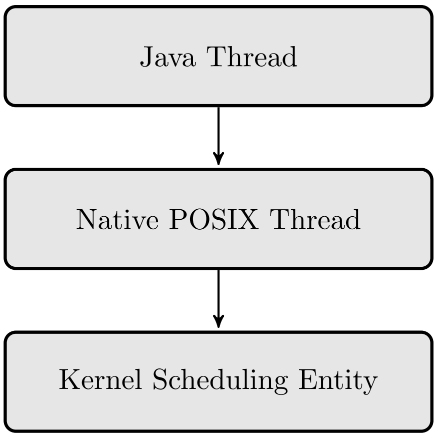


Figure 4.3.: Linux 1-to-1 thread mapping

Unfortunately the Java Threads API does not expose the ability to set the CPU or core affinity despite numerous use cases where setting the affinity of threads would be beneficial – such as improving cache and network performance or real-time applications [39, 14, 17]. There exists a request for enhancement on this issue but it was set to the state *"Closed, Will Not Fix"* [56].

4.3.1. JNI Library

To bind the workers to a specific core, we wrote a small JNI library – see Listing 4.6 for the API. The method `set(int physicalUnit)` (Line 2) is used to bind the current thread to a physical unit. With `set(int[]physicalUnits)` the current thread can be bound to several physical units, for example to a node in a NUMA system (Line 5). The workers of the intervals scheduler use `set(int physicalUnit)` in their `run()` method to set the affinity to a separate physical unit each.

```
1  public class Affinity {
2    public static native void set(int physicalUnit)
3        throws SetAffinityException;
4
5    public static native void set(int[] physicalUnits)
6        throws SetAffinityException;
7
8    static { System.loadLibrary("Affinity"); }
9  }
```

Listing 4.6: `Affinity` class with interface for the native methods

Listing 4.7 contains the native method declarations which are implemented in Listing 4.8. Line 8 declares `set(int physicalUnit)` and on Line 11 we declare the method `set(int[] physicalUnits)`.

```
1  #include <jni.h>
2  #ifndef _Included_ch_ethz_hwloc_Affinity
3  #define _Included_ch_ethz_hwloc_Affinity
4  #ifdef __cplusplus
5  extern "C" {
6  #endif
7
8  JNIEXPORT void JNICALL Java_ch_ethz_hwloc_Affinity_set__I
9    (JNIEnv *, jclass, jint);
10
11 JNIEXPORT void JNICALL Java_ch_ethz_hwloc_Affinity_set___3I
12   (JNIEnv *, jclass, jintArray);
13
14 #ifdef __cplusplus
15 }
16 #endif
17 #endif
```

Listing 4.7: `Affinity` class: JNI C header

The implementation for setting the affinity is shown in Listing 4.8. When setting the affinity, we first initialize the CPU set structure (Lines 15 and 25) and then we set it to the specified physical units (Lines 16 and 32). The actual affinity change is carried out in function `set_affinity()` on Lines 38 – 50.

Before we can set the affinity of a Java thread, we need to get its native thread ID. To get the native thread ID, we use the `pthread_self()` function (Line 40). The function `pthread_setaffinity_np()` (Line 42) sets the CPU affinity mask of the thread to the CPU set pointed to by `cpuset`. If the call is successful, and the thread is not currently running on one of the CPUs in `cpuset`, then it is migrated to one of those CPUs. If there is an error, `pthread_setaffinity_np()` returns a nonzero error number and we throw an exception (Line 45).

```c
#define _GNU_SOURCE
#include <syscall.h>
#include <sched.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>
#include "ch_ethz_hwloc_Affinity.h"

void set_affinity(JNIEnv *env, const cpu_set_t *cpuset);

JNIEXPORT void JNICALL
Java_ch_ethz_hwloc_Affinity_set__I(JNIEnv *env,
    jclass class, jint physical_unit) {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(physical_unit, &cpuset);
  set_affinity(env, &cpuset);
}

JNIEXPORT void JNICALL
Java_ch_ethz_hwloc_Affinity_set___3I(JNIEnv *env,
    jclass class, jintArray physical_units) {
  int i;
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  jsize len = (*env)->GetArrayLength(env, physical_units);
  jint *units = (*env)->GetIntArrayElements(env,
      physical_units, 0);

  for (i = 0; i < len; i++) {
    CPU_SET(units[i], &cpuset);
  }

  set_affinity(env, &cpuset);
}

void set_affinity(JNIEnv *env, const cpu_set_t *cpuset) {
  int s;
  pthread_t thread = pthread_self();

  s = pthread_setaffinity_np(thread, sizeof(cpu_set_t),
      cpuset);

  if (s != 0) {
    (*env)->ThrowNew(env, (*env)->FindClass(env,
        "ch/ethz/hwloc/SetAffinityException"),
        "Couldn't set affinity!");
  }
}
```

Listing 4.8: `Affinity` class: JNI C implementation to set the affinity

4.3.2. Restrictions

Portability

As we are directly using functions provided by POSIX threads, our implementation is not portable across operating systems that do not support the POSIX standard. To make our library portable, it could be rewritten using *Portable Linux Processor Affinity (PLPA)* [55] or *Portable Hardware Locality (hwloc)* [54].

Data Locality

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 a NUMA system every node has a direct connection to local memory providing fast access, and an indirect connection to remote memory with slower access. In the Intel Nehalem system for example, local DRAM access takes $\sim$ 60 ns, while remote DRAM access takes $\sim$ 100 ns (Table 3.1).

In the Java HotSpot VM, the NUMA-aware allocator has been implemented to provide automatic memory placement optimizations for Java applications [43, 57, 26]: "The allocator controls the eden space of the young generation of the heap, where most of the new objects are created. It divides the space into regions each of which is placed in the memory of a specific node. The allocator relies on a hypothesis that a thread that allocates the object will be the most likely to use the object. To ensure the fastest access to the new object, the allocator places it in the region local to the allocating thread."

On Linux, the implementation is based on work by Kleen [28]. To enable the NUMA-aware allocator, we invoke the JVM with the parameter `-XX:+UseNUMA` when using LASSI.

Chapter 5

Performance Evaluation

This chapter presents our performance evaluation of LASSI. First we compare the performance of locality-ignorant benchmarks run with LASSI and the original scheduler implementation. It is important that LASSI's performance is comparable to that of the original scheduler. Then we study different locality-aware benchmarks. By scheduling data sharing intervals on the same processor they perform prefetching of shared regions for one another which improves performance. Our locality-aware benchmarks only explore data sharing intervals and do not test how scheduling non-communicating intervals with high memory footprints on different processors could help in reducing cache contention and potential cache capacity problems.

5.1. Methodology

The performance results in this chapter are obtained on an Intel Nehalem system with 2 processors and 8 cores, running Ubuntu 9.04 64-bit with kernel 2.6.29 patched to support perfmon2 [15] (Appendix A.2). The JVM used is Sun Hotspot JDK 1.6.0_20 which is invoked with the following parameters:

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

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. We use perfmon2 to track the following events [37]:

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

The number of cache read hits and misses can also be determined by tracking requests being sent to the uncore's Global Queue, but it is simpler with the events listed above.

The execution times and cache events counts reported are the average of the 3 best benchmark iterations from 10 separate invocations.

5.2. Non-Locality Benchmarks

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 (Appendix B) with our new scheduler implementation. As Figure 5.1 shows, the performance of the locality-ignorant JGF benchmarks executed on LASSI is comparable to the original implementation.

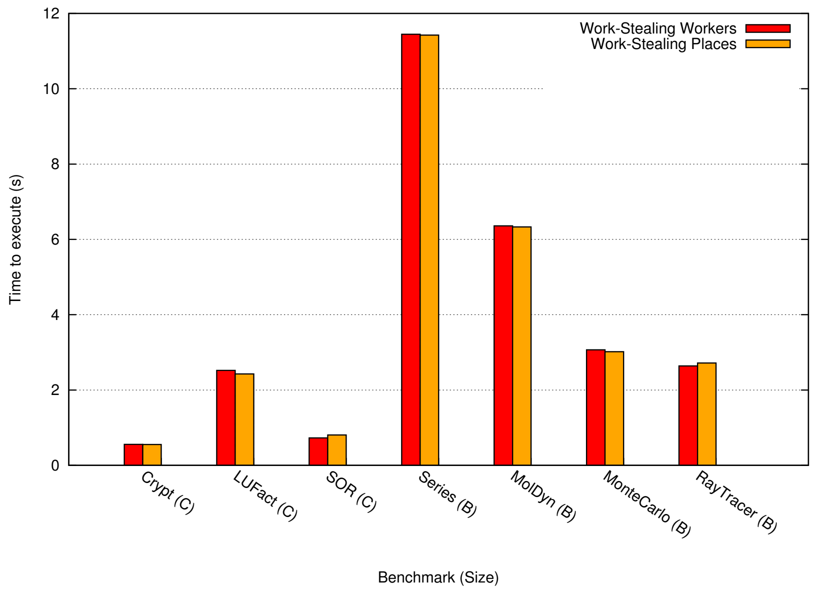


Figure 5.1.: Locality-ignorant JGF benchmarks using LASSI on our Intel Nehalem (Appendix A.2) test machine

5.3. Locality Benchmarks

5.3.1. Cache-Stress Test

We ported the *Cache Stress Test* benchmark from the threaded version developed in Chapter 3 to use intervals.

Like the threaded version, it first randomly initializes two integer arrays of equal size to the last level cache per processor. Then it creates an inline root interval which produces 64 subintervals with their locality set to a specific place: 32 have their locality set for *place 0* and 32 for *place 1*.

One half of the subintervals operate on the elements of the first array and the other half operate on the elements of the second array. Each interval's task adds and multiplies all the elements of its respective array 100 times.

Like in the threaded version, we implement several different variants of the benchmark, each having different locality properties:

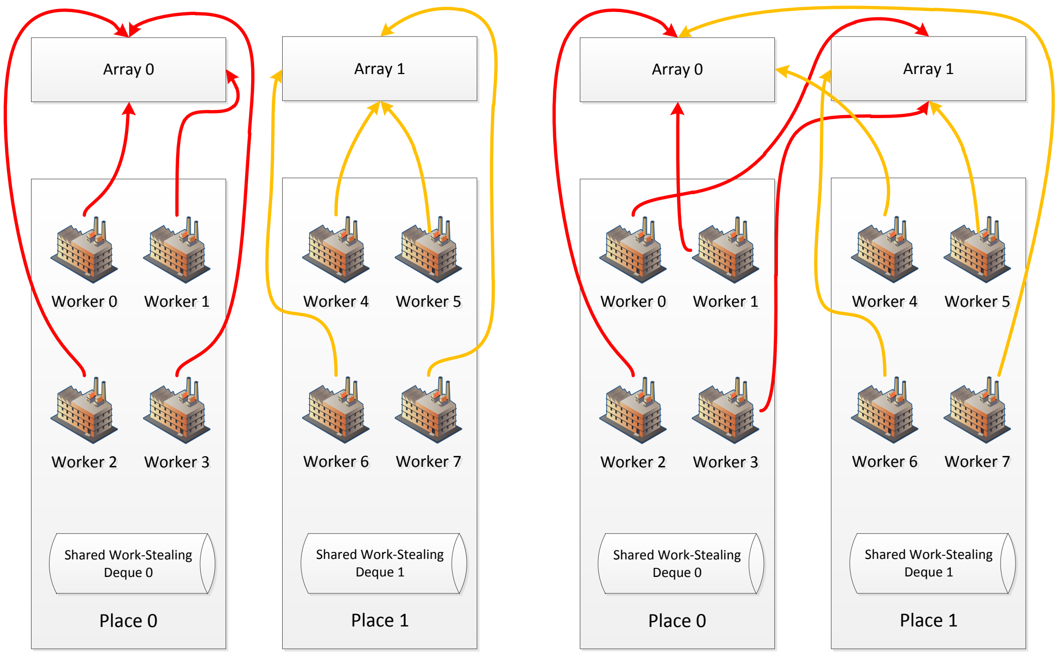
Best Locality: All the intervals working on the first array have their locality set to *place 0* and all intervals working on the second array have their locality set to *place 1*.

Ignorant Locality: The intervals do not have any locality set, i.e. they are locality-ignorant.

Random Locality: The locality of the intervals is set to a *random* place.

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*.

Figure 5.2 shows the place allocations for the benchmark variants with *best* and *worst locality*.



(a) Best Locality

(b) Worst Locality

Figure 5.2.: *Cache Stress Test* with *best* and *worst locality*

When running the intervals implementations of the *Cache Stress Test* benchmarks, we observe similar behavior to the threaded versions. As is shown in Table 5.1 the implementation with *best locality* is the fastest and provides the largest speedup.

In Figure 5.3 we show the execution times normalized to that of the *best locality* implementation. The *best locality* implementation is more than 10% faster compared to the other locality benchmarks.

	Runtime (in seconds)	Speedup (over sequential)
Best Locality	3,596	7,11
Ignorant Locality	4,038	6,33
Random Locality	4,030	6,35
Worst Locality	3,982	6,42
Sequential Implementation	25,571	1

Table 5.1.: *Cache Stress Test* execution times and speedups over the sequential implementation

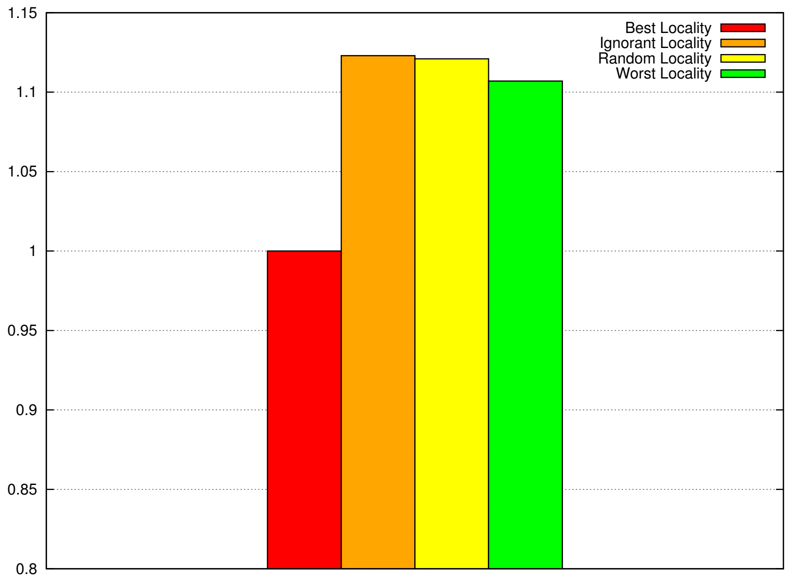


Figure 5.3.: *Cache Stress Test* with execution times normalized to *best locality*

In the *best locality* benchmark, the intervals perform prefetching of the array elements for one another. In the other benchmarks, intervals compete for the L3 cache and overwrite one another's entries. This reflects itself in the number of cache hit and miss events listed in Table 5.2.

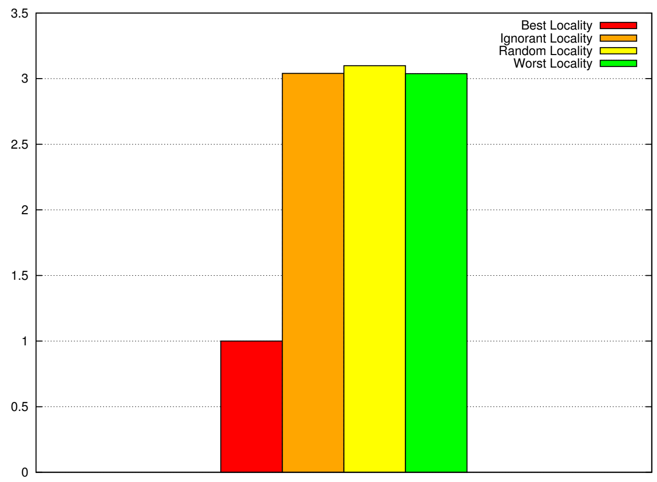
Figure 5.4 shows the cache hits and misses normalized to the measurements of the *best locality* implementation. The *best locality* benchmark has up to 1.5× more L3 cache read hits and 3.1× fewer L3 cache read misses than the other benchmarks.

The intervals implementation of the benchmarks with *ignorant*, *random* and *worst locality* is faster compared to the multi-threaded benchmark implementations of the same localities.

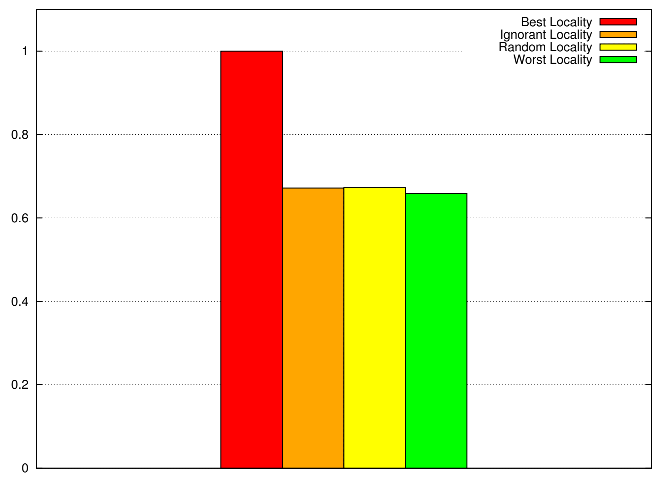
A reason for this is the use of *Work-Stealing Places*. They have a positive effect on the runtime due to their load-balancing properties. As the number of cache hits and misses shows, they do not produce counter-productive steals for the *Cache Stress Test* benchmarks.

	L3 Cache Read Hits	L3 Cache Read Misses
Best Locality	2 335	275
Ignorant Locality	1 568	837
Random Locality	1 570	854
Worst Locality	1 539	837

Table 5.2.: *Cache Stress Test* L3 cache read hits and misses (rounded to the nearest million)



(a) L3 Cache Read Misses



(b) L3 Cache Read Hits

Figure 5.4.: *Cache Stress Test* with L3 cache read hits and misses normalized to *best locality*

5.3.2. Merge Sort

The *Merge Sort* benchmark uses divide-and-conquer to recursively sort 4 194 304 randomly initialized integer values. Those integers 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, i.e. $8 \times 8\,192$ sorters overall. Each sorter randomly initializes an array of size $4\,194\,304/(8 \times 8\,192)$ and sorts it with `Arrays.sort()`. Merger intervals merge two neighboring sorted arrays into one sorted array of doubled size until all subarrays are merged into a large array of size 4 194 304. Figure 5.5 shows an example of the tree-like hierarchy of sorter and merger intervals used to sort a randomly initialized list of integers with the help of 8 sorters and 7 mergers.

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

Best Locality: Half the sorter intervals have locality for *place 0* and the other half have their locality set for *place 1*. Merger intervals copy the locality of the interval they get the left array to merge from, i.e. they are scheduled on the same place as their left predecessors in the tree hierarchy.

Ignorant Locality: The sorter and merger intervals do not have any locality set, i.e. they are locality-ignorant.

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, i.e. if the left predecessor was scheduled on *place 0*, then the merger is scheduled on *place 1* and vice versa.

Figure 5.5 shows the place allocations for the benchmark variant with *best locality*.

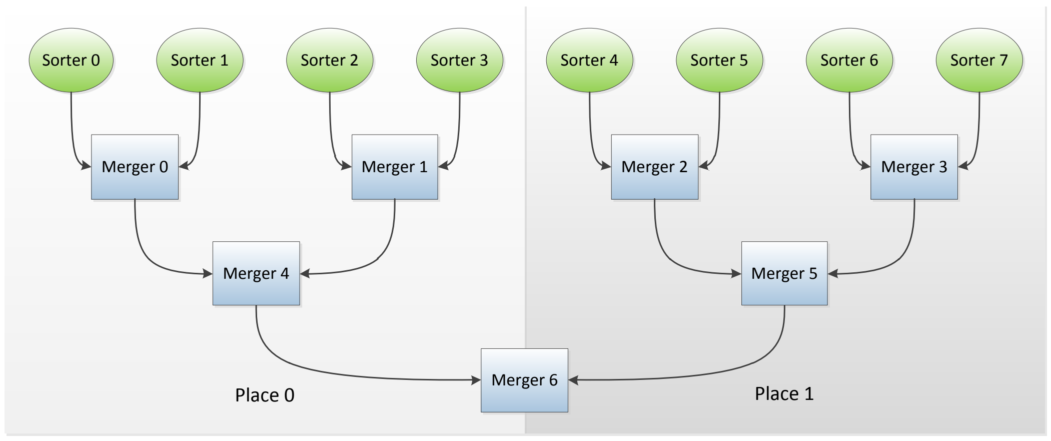


Figure 5.5.: *Merge Sort* with *best locality*

The *best locality* implementation puts data sharing intervals onto the same *place*. They perform prefetching for each other, i.e. they help to obtain and maintain the frequently used integers in the local L3 cache. With the *worst locality* we try to achieve the opposite: Merger intervals compete for the L3 caches and overwrite each other's entries.

Table 5.3 shows the execution times and the speedups over the *ignorant locality* implementation for the *best* and *worst locality* implementations. As expected, the implementation with *best locality* is the fastest and provides the largest speedup.

	Runtime (in ms)	Speedup (over *worst locality*)
Best Locality	672,667	1,10
Worst Locality	726	1,02
Ignorant Locality	741,333	1

Table 5.3.: *Merge Sort* execution times and speedups over the *ignorant locality* implementation

Figure 5.6 illustrates the execution times normalized to that of the *best locality* implementation. The *best locality* implementation provides a speedup of up to 1.1× compared to the other locality benchmarks.

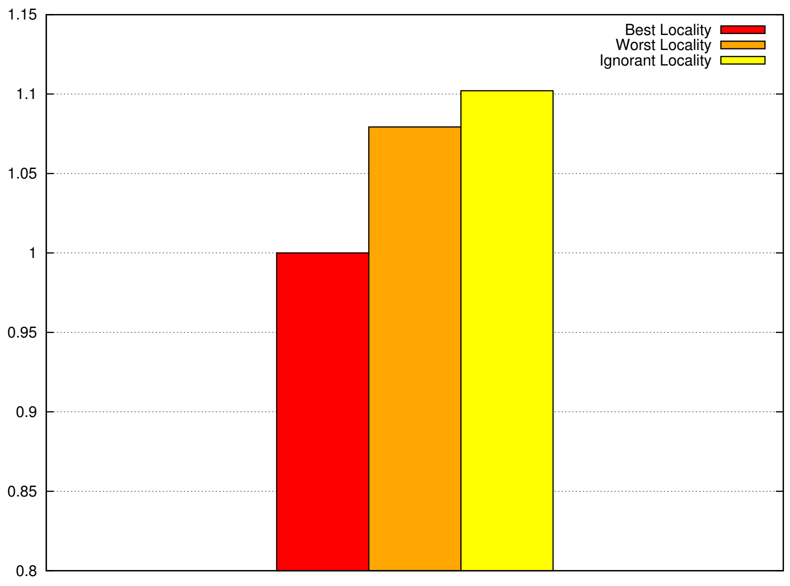


Figure 5.6.: *Merge Sort* with execution times normalized to *best locality*

In the *best locality* benchmark, the intervals perform prefetching of the array elements for one another. In the other benchmarks, intervals compete for the L3 cache and overwrite each other's entries. This reflects itself in the number of cache hit and miss events listed in Table 5.4.

	L3 Cache Read Hits	L3 Cache Read Misses
Best Locality	100	39
Worst Locality	99	41
Ignorant Locality	98	42

Table 5.4.: *Merge Sort* L3 cache read hits and misses (rounded to the nearest million)

The difference between the number of cache read hits and misses is only marginal: The *best locality* variant of the *Merge Sort* benchmark 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. Still, the speedup of 1.1× of the *best locality* variant over the *ignorant locality* variant shows that last level cache misses can have a significant impact on the runtime.

5.3.3. Block Matrix Multiplication

The *Block Matrix Multiplication* benchmark multiplies two $n \times n$ matrices A and B using intervals. The implementation employs the following recursion:

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

Thus, the $n \times n$ matrix multiplication can be reduced to 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices. The 8 multiplications can be calculated in parallel and when they are done, the 4 additions can also be computed in parallel.

We run the *Block Matrix Multiplication* benchmark to multiply two random 2048×2048 matrices. A 2048×2048 matrix needs about 16 MB of memory and should just about fit in the L3 cache of the two processors of our test machine.

The algorithm recursively splits the matrices A and B into quadrants until the base case of size 32×32 is reached (Figure 5.7). Then it multiplies the corresponding 32×32 submatrices of A and B using sequential matrix multiplication to add them up.

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

Best 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.

Worst Locality: The *worst locality* benchmark runs the multiplication and addition intervals in different places, destroying cache locality.

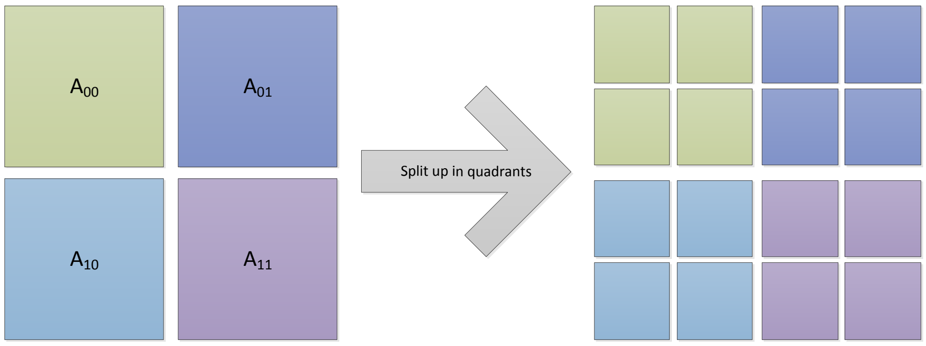
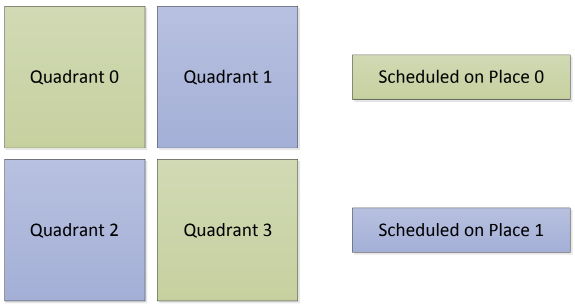
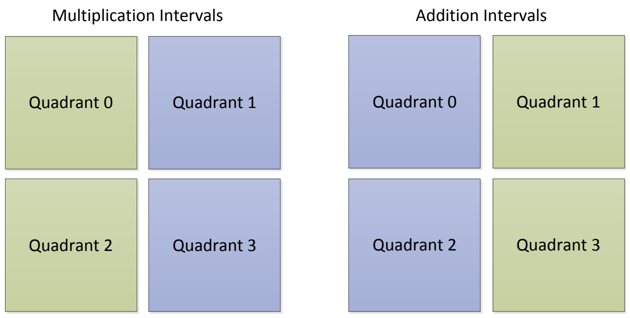


Figure 5.7.: Splitting matrix A into quadrants

Figure 5.8 illustrates the division of matrix quadrants between places for both variants.



(a) Best Locality



(b) Worst Locality

Figure 5.8.: *Block Matrix Multiplication* quadrants with *best* and *worst locality*

Table 5.5 shows the execution times and the speedups over the sequential algorithm for the *best locality* and *worst locality* benchmark implementations. As expected, the implementation with *best locality* is faster than the one for the *worst locality*.

	Runtime (in seconds)	Speedup (over sequential)
Best Locality	4,724	3,96
Worst Locality	5,075	3,69
Sequential Implementation	18,703	1

Table 5.5.: *Block Matrix Multiplication* execution times and speedups over the sequential implementation

Figure 5.9 depicts the execution time of the *worst locality* implementation normalized to that of the *best locality* implementation. The *best locality* implementation shows a speedup over the *worst locality* of about 1.07×.

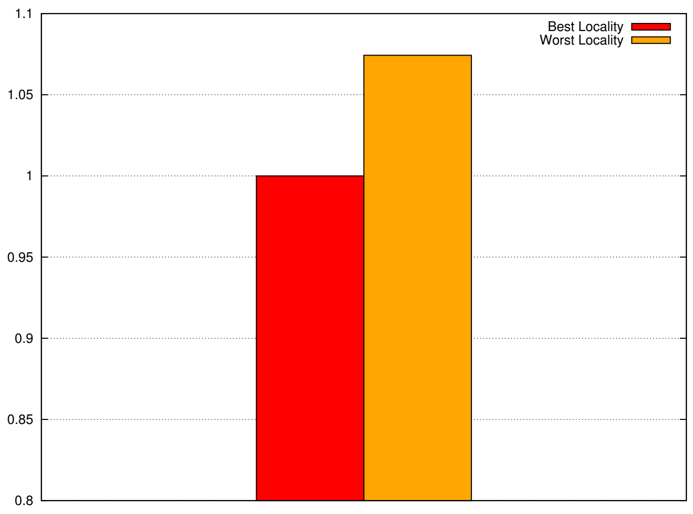


Figure 5.9.: *Block Matrix Multiplication* with execution times normalized to *best locality*

The difference between the number of cache read hits and misses is little (Table 5.6): 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. Last level cache misses can have a significant impact on the runtime nevertheless as is shown by the speedup of 1.07× of the *best locality* variant over the *worst locality*.

	L3 Cache Read Hits	L3 Cache Read Misses
Best Locality	1 283	352
Worst Locality	1 256	374

Table 5.6.: *Block Matrix Multiplication* L3 cache read hits and misses (rounded to the nearest million)

Chapter 6

Related Work

Locality-aware scheduling is a popular area of research. In the beginning, most research was done with shared task pools. As work-stealing scheduling is gaining in popularity, a lot of research on locality-aware scheduling is done with work-stealing schedulers nowadays. An alternative algorithm to work-stealing is parallel depth-first scheduling which is designed specifically for constructive cache sharing.

Shared Task Pools

Squillante and Lazowska [63] explore the importance of using processor-cache affinity information in shared-memory multiprocessor scheduling. They implement and compare several scheduling algorithms which trade off load balancing against processor-cache affinity.

Philbin et al. [58] use fine-grained threads to decompose a sequential program. They schedule these threads so as they improve the program's data locality. Like LASSI, the algorithm relies upon hints provided at the time of thread creation to determine a thread execution order likely to reduce cache misses.

Work-Stealing Scheduler

Acar, Blelloch, and Blumofe [1] present a theoretical bound for the number of cache misses for the work-stealing algorithm. They also provide an implementation of a locality-aware work-stealing scheduler. This algorithm is designed for single-core multiprocessor systems, while LASSI supports chip multiprocessor systems.

X10 [8, 61] is designed for parallel programming using the partitioned global address space model and place locality. Computations are divided among a set of places. Each of those places holds some data and hosts one or more activities that operate on those data. Agarwal et al. [2] present a novel framework for statically establishing place locality in X10.

The Habanero Java [22] language is based on early versions of the X10 language. Like X10, it supports locality control with task and data distributions using places.

Yan et al. [64] introduce *Hierarchical Place Trees* and integrate them into the Habanero Java compiler and runtime system. In contrast to our single level *Work-Stealing Places*, *Hierarchical Place Trees* support co-location of data and computation at multiple levels of a memory hierarchy.

Guo et al. [20] introduce SLAW, a scalable locality-aware adaptive work-stealing scheduler. Like LASSI, SLAW groups workers into places and requires locality hints provided by the programmer or compiler. In contrast to LASSI, SLAW disables cross-place steals and additionally supports adaptive scheduling by selecting a work-first or help-first policy for a task at runtime [21].

Zeldovich et al. [65] present libasync-smp, an asynchronous programming library allowing event-driven applications to run code for event handlers in parallel using work-stealing scheduling. Event-coloring is used to control the parallelism between events: events with the same color are handled serially and events with different colors are handled concurrently. Gaud et al. [19] extend the previous work by introducing heuristics aimed at improving the performance of the work-stealing algorithm. Like LASSI, the *locality-aware stealing* heuristics aims to preserve cache locality. Other heuristics introduced are *time-left stealing* and *penalty-aware stealing*. Unlike in our locality-aware scheduler, those heuristics only require little involvement from the application programmers.

The implementation of Threading Building Blocks [11, 59] uses a work-stealing scheduler which tries to limit unneeded migration of tasks and data. Work-stealing places of our intervals scheduler can be modified to do the same.

Parallel Depth-First Scheduler

Parallel depth-first scheduling was introduced by Blelloch, Gibbons, and Matias [5]. In parallel depth-first scheduling, tasks are assigned priorities in the same ordering as they would be executed in a sequential program. This means, tasks that would be executed earlier are given higher priorities than those that would be executed later. As a result, parallel depth-first scheduling tends to employ constructive cache sharing [38, 10] because it co-schedules threads in a way that simulates the sequential execution order. In work-stealing scheduling cores tend to have disjointed working sets. However, the concept of places can be used to enable constructive cache sharing in work-stealing schedulers.

Chapter 7

Conclusions and Future Work

7.1. Conclusions

In this thesis, we introduced LASSI, a locality-aware scheduler for intervals. It is designed for locality-aware scheduling using locality hints provided by the programmer. Instead of employing work-stealing workers, LASSI uses *Work-Stealing Places*. *Work-Stealing Places* are a novel data structure providing locality-awareness to the intervals scheduler.

While LASSI can improve performance of programs written with locality in mind, it is important to note that the performance of existing locality-ignorant programs 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 for data sharing benchmarks show that *best locality* placement of intervals can achieve up to 1.15× speedup over *worst* or *ignorant locality* placement. Cache hits can be increased by up to 1.5× and cache misses can be reduced by up to 3.1×.

Our benchmarks only perform limited tests. For example they 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.

In general, it is not easy to write benchmarks exploiting the memory hierarchy of NUMA systems. Many benchmarks mentioned in the literature do not show the desired effects anymore due to advances in hardware and software.

7.2. Future Work

Scheduling of lightweight threads is a very broad area of research and there are many potential directions we could further extend our work.

A possible area of future work would be to improve the API of *Work-Stealing Places* and locality-aware intervals. In the current implementation, places have to be manually configured for each system. This should be automated by making the underlying machine transparent to the user. To make our library portable, we could use *Portable Linux Processor Affinity (PLPA)* [55] or *Portable Hardware Locality (hwloc)* [54]. The way programmers have to provide locality hints to intervals is not very convenient and should be made more intuitive.

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 [22, 8, 61]. It might be interesting to see how *Work-Stealing Places* would benefit from supporting multiple levels of a memory hierarchy as it is done by Yan et al. [64]. *Hierarchical Place Trees* are designed to run on both homogeneous (CPU) and heterogeneous (GPU) multicore parallel systems.

Load balancing across work-stealing places could lead to counter-productive stealing. One possible direction for future work would be to avoid counter-productive steals. Agarwal et al. [2] present a novel framework for statically establishing place locality in a work-stealing scheduler. The implementation of Threading Building Blocks [11, 59] uses a work-stealing scheduler which tries to limit unneeded migration of tasks and data. Gaud et al. [19] introduce heuristics for *time-left stealing* and *penalty-aware stealing* which only require little involvement from the application programmers.

Every worker thread executes on another core which eliminates direct contention between them. But they still share their assigned core with other processes in the system. To further enhance our locality-aware scheduler, we could employ online contention detection. Mars et al. [42] describe how online contention detection can be used to dynamically reduce or increase the number of worker threads depending on the system load. Agrawal, He, and Leiserson [3] develop and analyze `A-STEAL`, an adaptive work-stealing algorithm with parallelism feedback.

Another possible direction for future research would be to explore *Parallel Depth-First Scheduling* as a possible replacement for *Work-Stealing Scheduling* in intervals. In parallel depth-first scheduling, tasks are assigned priorities in the same ordering they would be executed in a sequential program. As a result, parallel depth-first scheduling tends to employ constructive cache sharing [38, 10] as it co-schedules threads in a way that simulates the sequential execution order.

Part II.

Work-Stealing Queue Implementations

Chapter 8

Introduction

8.1. Motivation

The intervals implementation uses a work-stealing scheduler that employs a fixed number of threads called workers. Each worker has a local double-ended queue, or deque, to maintain its own pool of ready intervals from which it obtains work. When a worker's pool becomes empty, it tries to steal an interval from the pool of a victim worker chosen at random.

To enable efficient and scalable execution, management of the intervals must be made as fast as possible. In the non-blocking work-stealing algorithm,[1] the deques are implemented with non-blocking synchronization [4]. That is, instead of using mutual exclusion, it uses atomic synchronization primitives such as Compare-and-Swap [53]. The current deque implementation of intervals however uses mutual exclusion when trying to steal.

Our hypothesis is that we can improve the performance of the intervals scheduler with non-blocking queues. Thus, as a separate effort, we design and explore alternative non-blocking queues.

8.2. Overview

Chapter 9 summarizes the properties of work-stealing queues and introduces the *Work-Stealing Lazy Deque*, the queue currently used by the intervals scheduler. Chapter 10 describes the investigated queue implementations. None of the approaches we developed as part of this research produced a queue that was improving work-stealing performance on the machines we had to test them with (Appendix A.1 and A.2). Possible reasons for this are given in the performance evaluation in Chapter 11. Chapter 12 concludes and summarizes our findings and encountered problems in order to preserve this research for future reference.

[1]Non-blocking – in contrast to wait-free [25] – means that it is possible for a worker to starve while trying to steal from other workers. Live-locks cannot occur as if one worker starves then others must be making progress.

Chapter 9

Background

9.1. Work-Stealing Queues

In a work-stealing scheduler each worker has a local queue to maintain its own pool of ready tasks from which it obtains work. When a worker finds that its pool is empty, it becomes a thief and steals a task from the pool of a victim worker chosen at random.

Depending on the desired extraction strategy we can implement the work-stealing queues differently. Most work-stealing schedulers use work-stealing deques [4, 1, 7, 18, 13] but there are also implementations for LIFO or FIFO extraction [52].

A work-stealing deque is like a traditional deque [29] except that only the deque's owner thread accesses the deque's bottom end to put and take local work. Thiefs steal elements from the deque's top. This minimizes synchronization overhead for the deque's owner.

All work-stealing queues provide the following three methods in their interface:

- `put(WorkItem workItem)`: Puts `workItem` into the queue.
- `WorkItem take()`: Takes an object from the queue and returns it if the queue is not empty, otherwise returns `null`.
- `WorkItem steal()`: Returns `null` if the queue is empty. Otherwise, returns the element successfully stolen from the queue, or `null` if this worker loses a race with another worker to steal or take a work item.

```
interface WorkStealingQueue {
  public void put(WorkItem workItem);
  public WorkItem take();
  public WorkItem steal();
}
```

Listing 9.1: Work-stealing queue interface

Note that the `put()` and `take()` methods are invoked only by the queue's owner.

9.2. Current Queue Implementation

The queue currently used by the intervals scheduler is the *Work-Stealing Lazy Deque*. This deque is unbounded and uses a cyclic array to store its work items.

It is called *lazy* because the owner of the deque only lazily updates the location of the deque's head. This means it only updates the head when its owner tries to take a work item and finds it was stolen by a competing thief.

The members of the *Work-Stealing Lazy Deque* are defined as:

```java
public class WorkStealingLazyDeque implements WorkStealingQueue {
  static class ThiefData {
    int head = 0;
  }

  private AtomicReferenceArray<WorkItem> workItems =
    new AtomicReferenceArray<WorkItem>(1024);
  int ownerHead = 0, ownerTail = 0;
  private final ThiefData thief = new ThiefData();
  // ...
}
```

The `workItems` array contains the work items of the queue. `ownerHead` and `ownerTail` are indices in the array and represent the head and tail of the queue for the owner. `thief` represents the head for the thief and is also the lock object used when a thief tries to steal a work item.

Listing 9.2 defines the `put()` method which puts `workItem` onto the bottom of the deque. The method calls `expand()` when the array containing the work items is full (Line 7).

```java
 1 public void put(WorkItem workItem) {
 2   assert workItem != null;
 3   while (true) {
 4     final int length = workItems.length();
 5     final int tail = ownerTail;
 6
 7     if (tail - ownerHead >= length) {
 8       expand();
 9       continue;
10     }
11
12     workItems.set(tail % length, workItem);
13     ownerTail = tail + 1;
14     return;
15   }
16 }
```

Listing 9.2: Work-Stealing Lazy Deque: `put()` method

When the `workItems` array is full, `put()` calls the method `expand()` (Listing 9.3). `expand()` can only be called by the owner of the deque when no thieves are active (Line 2). It allocates a new doubled size array and copies over the work items from the old array to the new array. Then it resets the array indices and replaces the old array with the new one (Lines 15 – 17).

```
1  private void expand() {
2    synchronized (thief) {
3      int oldSize = workItems.length();
4      int newSize = 2 * oldSize;
5      int newTail = 0;
6      AtomicReferenceArray<WorkItem> newWorkItems =
7        new AtomicReferenceArray<WorkItem>(newSize);
8
9      for (int i = ownerHead; i < ownerTail; i++) {
10       newWorkItems.set(newTail % newSize,
11         workItems.get(i % oldSize));
12       newTail++;
13     }
14
15     ownerTail = newTail;
16     ownerHead = thief.head = 0;
17     workItems = newWorkItems;
18   }
19 }
```

Listing 9.3: Work-Stealing Lazy Deque: `expand()` method

Method `take()` is defined in Listing 9.4. It takes an object from the bottom of the deque if the deque is not empty, otherwise it returns `null`. On Line 8 we use Compare-and-Set[1] to check if another worker stole the item we wanted to take. If the item is gone, we know that all previous items must have been stolen too and we can update our notion of the head of the deque (Line 12).

```
1  public WorkItem take() {
2    if (ownerHead == ownerTail) // Deque is empty
3      return null;
4
5    final int lastTail = ownerTail - 1;
6    final int lastIndex = lastTail % workItems.length;
7    WorkItem workItem = workItems.get(lastIndex);
8    if (!workItems.compareAndSet(lastIndex, workItem, null))
9      workItem = null;
10
11   if (workItem == null) {
12     ownerHead = ownerTail;
13     return null;
14   }
15
16   ownerTail = lastTail;
17   return workItem;
18 }
```

Listing 9.4: Work-Stealing Lazy Deque: `take()` method

[1]Also known as Compare-and-Swap [27]. `compareAndSet(int i, E expect, E update)` atomically sets the element at position `i` to the given object `update` if the current value equals `expect`. The method returns `true` if successful and `false` else.

Listing 9.5 shows the implementation of `steal()` using a synchronized block to make sure there can only be one thief at any given time (Lines 2 – 17).
There are three cases where `steal()` returns `null` (Line 12):

- The deque is empty.
- The thief lost a race with another thief trying to steal the topmost work item.
- The thief lost a race for the last work item caused by a concurrent `take()` operation.

Otherwise, `steal()` increases the head and returns the successfully stolen work item (Line 16).

```
1  public WorkItem steal() {
2    synchronized (thief) {
3      final int head = thief.head;
4      final int index = head % workItems.length;
5      WorkItem workItem = workItems.get(index);
6
7      if (!workItems.compareAndSet(index, workItem, null)) {
8        workItem = null;
9      }
10
11     if (workItem == null) {
12       return null;
13     }
14
15     thief.head++;
16     return workItem;
17   }
18 }
```

Listing 9.5: Work-Stealing Lazy Deque: `steal()` method

Chapter 10

Investigated Queues

Besides the *Work-Stealing Deque* (Section 10.1) and *Idempotent Work-Stealing Deque* (Section 10.2) we also implemented the alternative work-stealing queues *Dynamic Work-Stealing Deque* (Section 10.3.1) and *Duplicating Work-Stealing Queue* (Section 10.3.2).

10.1. Work-Stealing Deque

The *Work-Stealing Deque* is an unbounded double-ended queue that dynamically resizes itself as needed. Its design is based on the *Dynamic Circular Work-Stealing Deque* [9, 36].

The `WorkStealingDeque` class has three fields, `workItems`, `bottom`, and `top`:

```java
public class WorkStealingDeque implements WorkStealingQueue {
  private volatile WorkItem[] workItems = new WorkItem[1024];
  private volatile int bottom = 0;
  private AtomicInteger top = new AtomicInteger(0);
  // ...
}
```

`workItems` is a cyclic array with `top` and `bottom` indicating the two ends of the deque. An important property of `top` is that it is never decreased. The deque is empty if `top` is greater than or equal to `bottom`.

The `put()` method (Listing 10.1) first checks whether the current circular array is full (Line 7). If it is full, we call `expand()` (Line 8) to enlarge it by copying the deque's elements into a bigger array. Afterwards we can put the new work item in the location specified by `bottom`, and then increment `bottom` by 1 (Line 13).

Listing 10.2 shows the `expand()` method. It allocates a new doubled size array and copies the old array's elements into the new array. By using modular arithmetic we ensure that even though the array's size has increased, there is no need to update the `top` or `bottom` fields.

In Listing 10.3 we define the `take()` method. If the deque is empty, we reset it to an empty state where `bottom == top` and return `null` (Lines 9 – 11). If taking

```
public void put(WorkItem workItem) {
  int oldBottom = bottom;
  int oldTop = top.get();
  WorkItem[] currentWorkItems = workItems;
  int size = oldBottom - oldTop;

  if (size >= currentWorkItems.length - 1) {
    currentWorkItems = expand(currentWorkItems, oldBottom, oldTop);
    workItems = currentWorkItems;
  }

  currentWorkItems[oldBottom % currentWorkItems.length] = workItem;
  bottom = oldBottom + 1;
}
```

Listing 10.1: Work-Stealing Deque: `put()` method

```
private WorkItem[] expand(WorkItem[] currentWorkItems,
    int bottom, int top) {
  WorkItem[] newWorkItems =
    new WorkItem[currentWorkItems.length * 2];

  for (int i = top; i < bottom; i++)
    newWorkItems[i % newWorkItems.length] =
      currentWorkItems[i % currentWorkItems.length];

  return newWorkItems;
}
```

Listing 10.2: Work-Stealing Deque: `expand()` method

a work item does not make the deque empty, the owner can take it without using a Compare-and-Swap (CAS) operation (Lines 17 – 18). If the owner is trying to take the last work item, it must perform a Compare-and-Swap operation on `top` to check whether it won or lost any race with a concurrent `steal()` operation to take the last item (Line 20). Regardless whether the CAS operation succeeds, the value of `top` is incremented by 1 and the deque is empty: If the CAS in `take()` fails, then some concurrent `steal()` operation succeeded in stealing the last work item and incremented `top`. Therefore the operation completes by storing the incremented top value in `bottom` which resets the deque to an empty state (Line 23).

The `steal()` method (Listing 10.4) first reads `top`, then `bottom`. The order is important: If a thread reads `bottom` after `top` and sees it is no greater, the queue is indeed empty because a concurrent modification of `top` could only have increased the `top` value.

If the deque is empty, `steal()` returns `null` (Lines 11 – 12). Otherwise it reads the element stored in the `top` position of the cyclic array, and tries to increment `top` using a CAS operation. When the Compare-and-Swap fails, it implies that a concurrent `steal()` successfully removed an element from the deque, so the thief tries to steal again. If the CAS succeeded, `steal()` returns stolen work item.

```java
public WorkItem take() {
  int oldBottom = this.bottom;
  WorkItem[] currentWorkItems = workItems;
  oldBottom = oldBottom - 1;
  this.bottom = oldBottom;
  int oldTop = top.get();
  int size = oldBottom - oldTop;

  if (size < 0) {
    bottom = oldTop;
    return null;
  }

  WorkItem workItem =
    currentWorkItems[bottom % currentWorkItems.length];

  if (size > 0)
    return workItem;

  if (!top.compareAndSet(oldTop, oldTop + 1))
    workItem = null; // queue is empty

  bottom = oldTop + 1;
  return workItem;
}
```

Listing 10.3: Work-stealing Deque: `take()` method

Any `take()` that empties the deque tries to modify `top` using a CAS operation. This is to prevent `steal()` from returning the deque's last work item if it was already taken by a concurrent `take()` after `bottom` is read (Line 7), but before the CAS operation is executed (Line 16),

```java
 1  public WorkItem steal(Worker thiefWorker) {
 2    int oldTop, oldBottom;
 3    WorkItem workItem;
 4
 5    while (true) {
 6      oldTop = top.get();
 7      oldBottom = bottom;
 8      WorkItem[] currentWorkItems = workItems;
 9      int size = oldBottom - oldTop;
10
11      if (size <= 0)
12        return null; // empty
13
14      workItem = currentWorkItems[oldTop % currentWorkItems.length];
15
16      if (top.compareAndSet(oldTop, oldTop + 1))
17        break;
18    }
19    return workItem;
20  }
```

Listing 10.4: Work-Stealing Deque: `steal()` method

10.2. Idempotent Work-Stealing Deque

The *Idempotent Work-Stealing Deque* is based on ideas from [35] and [52]. It is an unbounded double-ended queue that can resize itself if needed.

Unlike the *Work-Stealing Lazy Deque* (Section 9.2) or the *Work-Stealing Deque* (Section 10.1), the *Idempotent Work-Stealing Deque* does not guarantee that each inserted work item is extracted *exactly* once. Instead it uses the relaxed semantics of guaranteeing that each inserted work item is extracted *at least* once.

While this nondeterminism might be dangerous in many applications, it is fine for our usage of the deque as we modified the interval's `exec()` method to be idempotent (Section 10.2.1).[1]

10.2.1. Idempotent Interval

Listing 10.5 shows the idempotent interval implementation. Each interval has an associated state `RunningState` (Line 2). Upon initialization, the state is set to `INIT` (Line 5). The internal `exec()` method performs an atomic CAS operation to try to switch from `INIT` to `RUNNING` (Line 11). If the CAS operation is successful, the associated task is executed and the state is set to `DONE` afterwards (Line 13).

```
 1 public abstract class Interval extends WorkItem {
 2   enum RunningState { INIT, RUNNING, DONE }
 3
 4   private final AtomicReference<RunningState> runningState =
 5     new AtomicReference<RunningState>(RunningState.INIT);
 6
 7   // The "main" method for this interval: invoked when we are
 8   // scheduled. Simply invokes "exec()".
 9   void exec(Worker worker) {
10     if (runningState.compareAndSet(RunningState.INIT,
11         RunningState.RUNNING)) {
12       exec();  // execute the associated action
13       runningState.set(RunningState.DONE);
14     }
15   }
16
17   // ...
18 }
```

Listing 10.5: Idempotent interval

This ensures that each interval is only executed once, or stated differently: running an interval is an idempotent operation.

Idempotent intervals would also simplify the mailbox style implementation for locality-aware scheduling introduced by Acar, Blelloch, and Blumofe [1].

[1] If the application can tolerate duplicated work, for example parallel garbage collectors [16] or constraint solvers, we do not have to make the `exec()` method idempotent.

10.2.2. Implementation

In contrast to the *Work-Stealing Lazy Deque* (Section 9.2) or the *Work-Stealing Deque* (Section 10.1), the `take()` method of the *Idempotent Work-Stealing Deque* does not have to use an expensive CAS operation.

The `IdempotentWorkStealingDeque` class has an inner class, `ArrayData`, and two fields, `anchor` and `workItems`:

```
public class IdempotentWorkStealingDeque
    implements WorkStealingQueue {
  class ArrayData {
    final int head, size;

    public ArrayData(int head, int size) {
      this.head = head;
      this.size = size;
    }
  }

  private AtomicStampedReference<ArrayData> anchor =
    new AtomicStampedReference<ArrayData>(new ArrayData(0, 0), 0);

  private WorkItem[] workItems = new WorkItem[1024];

  // ...
}
```

The array `workItems` is used in a cyclic way with head and size encapsulated in an `ArrayData` reference. The `ArrayData` reference is maintained by the atomic stamped reference `anchor` together with an integer stamp. Our algorithm needs to guard against the ABA problem[2] in the `steal()` operation and uses the stamp as an ABA-prevention tag.[3]

Listing 10.6 shows the `put()` method. First the owner reads the anchor to get the head and size of the queue as well as the ABA-prevention tag (Lines 5 – 8). Then the owner checks whether the array is full of not (Line 10). If it is full, the owner expands the array by calling `expand()` and loops around (Line 11). Otherwise it puts the work item onto the tail of the queue (Line 17). In Line 19 the owner updates the anchor by incrementing the queue's size and ABA-prevention tag.

The method `expand()` is defined in Listing 10.7. For the owner to expand a full queue, it allocates a new array with double the current capacity (Line 5) and copies the work items from the current array to the newly allocated one (Lines 7 – 9). After that, it sets `workItems` to the new array (Line 12).

Listing 10.8 presents the method `take()`. The owner reads the anchor variable to get the head and size of the queue, and also the ABA-prevention tag (Lines 2 –

[2][51]: When a thread reads a value A from a shared variable, computes a new value, and then tries a Compare-and-Swap operation, it is possible that the CAS succeeds even if it should not. This could happen for example when between the read and the CAS some other thread changed the A back to B and then back to A again.

[3]Instead of using a tag, we could also use another ABA prevention mechanism, like bounded tags [53] or hazard pointers [50]

```java
public void put(WorkItem workItem) {
  int head, size, tag;

  while (true) {
    ArrayData arrayData = anchor.getReference();
    head = arrayData.head;
    size = arrayData.size;
    tag = anchor.getStamp();

    if (size == workItems.length) {
      expand();
    } else {
      break;
    }
  }

  workItems[(head + size) % workItems.length] = workItem;

  anchor.set(new ArrayData(head, size + 1), tag + 1);
}
```

Listing 10.6: Idempotent Work-Stealing Deque: `put()` method

```java
private void expand() {
  ArrayData arrayData = anchor.getReference();
  int head = arrayData.head;
  int size = arrayData.size;
  WorkItem[] tempWorkItems = new WorkItem[2 * size];

  for (int i = 0; i < size; i++) {
    tempWorkItems[(head + i) % tempWorkItems.length] =
      workItems[(head + i) % workItems.length];
  }

  workItems = tempWorkItems;
}
```

Listing 10.7: Idempotent Work-Stealing Deque: `expand()` method

5). Then it checks if the queue is empty (Line 7). If it is empty, `take()` returns `null`. Else, it reads the work item at the tail of the queue (Lines 12). In Line 19 the method updates the anchor by decrementing the queues size to indicate the extraction of a work item.

The `steal()` method (Listing 10.9) starts by reading the anchor variable to get the head and size of the queue as well as the ABA-prevention tag (Lines 3 – 6). In Line 8 the thread checks if the queue is empty. If it is empty, `steal()` returns `null`. Otherwise it gets a pointer to `workItems` (Line 12) and reads the work item at the head of the queue (Line 13).

The `compareAndSet()` in Line 17 checks that no work item was lost: Checking of the ABA-prevention tag makes sure that since the reads in Lines 3 – 6 the deque's owner has not overwritten the work item read in Line 13. If the `compareAndSet()`

```java
public WorkItem take() {
  ArrayData arrayData = anchor.getReference();
  int head = arrayData.head;
  int size = arrayData.size;
  int tag = anchor.getStamp();

  if (size == 0) {
    return null;
  }

  WorkItem workItem =
    workItems[(head + size - 1) % workItems.length];

  anchor.set(new ArrayData(head, size - 1), tag);

  return workItem;
}
```

Listing 10.8: Idempotent Work-Stealing Deque: `take()` method

is successful, it updates the anchor with the incremented head and decremented size to indicate the stealing and returns the stolen work item. Else, the thread tries to steal again.

```java
public WorkItem steal(Worker thiefWorker) {
  while (true) {
    ArrayData arrayData = anchor.getReference();
    int head = arrayData.head;
    int size = arrayData.size;
    int tag = anchor.getStamp();

    if (size == 0) {
      return null;
    }

    WorkItem[] tempWorkItems = workItems;
    WorkItem workItem = tempWorkItems[head % tempWorkItems.length];
    int newHead = head + 1 % Integer.MAX_VALUE;

    if (anchor.compareAndSet(arrayData,
        new ArrayData(newHead, size - 1), tag, tag)) {
      return workItem;
    }
  }
}
```

Listing 10.9: Idempotent Work-Stealing Deque: `steal()` method

10.3. Alternative Implementations

10.3.1. Dynamic Work-Stealing Deque

The *Dynamic Work-Stealing Deque* uses a list of small arrays to manage work items. It is based on the work-stealing deque algorithm presented by Hendler et al. [24].

The algorithm uses instances of the class `Node` to build a doubly linked list:

```java
public class DynamicWorkStealingDeque implements WorkStealingQueue {
  class Node {
    static final int SIZE = 512;
    WorkItem[] workItems = new WorkItem[SIZE];
    Node next, prev;
  }

  class Index {
    Node node;
    int index;

    Index(Node node, int index) {
      this.node = node;
      this.index = index;
    }
  }

  private Index bottom;
  private AtomicStampedReference<Index> top;

  public DynamicWorkStealingDeque() {
    Node node1 = new Node();
    Node node2 = new Node();
    node1.next = node2;
    node2.prev = node1;

    bottom = new Index(node1, Node.SIZE - 1);
    top = new AtomicStampedReference<Index>(
        new Index(node1, Node.SIZE - 1), 0);
  }

  // ...
}
```

`top` and `bottom` indicate the two ends of the queue. They are instances of the `Index` class pointing to a deque's node with an offset into that node's array.

For the doubly linked list to have a good performance, it must only fall back to using a costly CAS operation when a potential conflict requires it. The potential conflict occurs when a `take()` and `steal()` concurrently try to remove the last item from the deque.

For being able to do this, we need to have an efficient mechanism to allow detection of this situation using the relations between the `top` and `bottom` pointers. We have to be careful as these could point to entries residing in different nodes. We observe that given one pointer – ignoring which array it resides in – the distance of the other, cannot be more than 1 if the deque is empty. Hendler et al. [24] describe the method in more detail.

As the evaluation in Section 11.3.1 shows, the dynamic work-stealing deque's complexity makes it slower than other work-stealing queues.

10.3.2. Duplicating Work-Stealing Queue

The *Duplicating Work-Stealing Queue* provides an alternative to the *Idempotent Work-Stealing Deque*. Its design is based on the Task Parallel Library (TPL) [35].

Like the *Idempotent Work-Stealing Deque*, the *Duplicating Work-Stealing Queue* potentially returns a pushed element more than once. In particular, the `put()` and `take()` operations behave like normal, but the `steal()` operation is allowed to either take an element and remove it from the queue, or to just duplicate an element in the queue. By allowing duplication we can avoid an expensive CAS instruction in the `take()` operation.

As we are also using idempotent intervals together with duplicating work-stealing queues, this nondeterminism is fine for our usage.

Whereas the idempotent work-stealing deque implementation relies on atomic CAS operations and uses a tag to prevent the ABA problem, the duplicating work-stealing queue implementation uses a lock on all but the critical paths. This simplifies the implementation but we could not find any drastic performance difference (Section 11.3.3).

Chapter 11

Performance Evaluation

We evaluate the performance of the different work-stealing queue implementations when used by the intervals scheduler with a variety of parallel Java Grande Forum benchmarks (Appendix B) on two different machines:

- Intel Core2 Duo with one processor and two cores, running Ubuntu 10.04 64-bit with kernel 2.6.32 and Sun Hotspot JDK 1.6.0_20 (Appendix A.1)
- Intel Nehalem with two processors and eight cores, running Ubuntu 9.04 64-bit with kernel 2.6.29 and Sun Hotspot JDK 1.6.0_20 (Appendix A.2)

Both machines invoke the JVM with the following parameters:

```
-server -Xmx2048M -Xms2048M -Xss8m
```

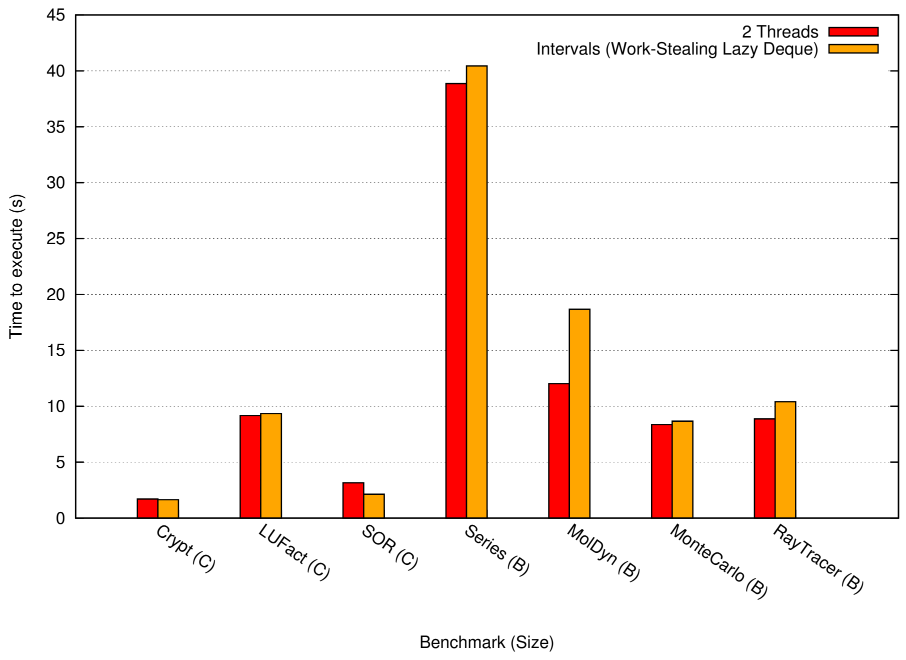
The execution time reported is the average of the 3 best benchmark iterations from 10 separate invocations.

11.1. Intervals

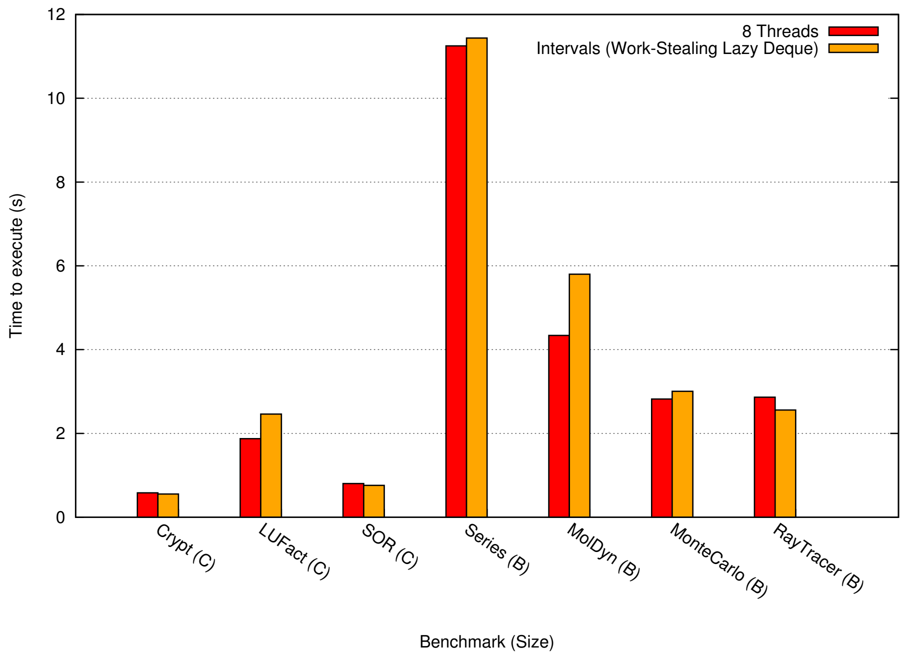
The performance of the intervals implementation of the JGF benchmarks is comparable to the threaded implementation as is shown in Figure 11.1.

The only benchmarks where intervals are considerably slower than threads are *LuFact* and *MolDyn*. The main reason is that the way work is distributed between intervals makes the workers run out of work too often. When a worker runs out of work, it becomes idle and only wakes up if a new work item is added to its queue. Idle workers acquire a local semaphore which blocks until release is invoked by some other worker and are added to a global idle list guarded by a shared lock. If there is an idle worker, adding a new work item releases the semaphore of the worker and removes it from the idle list.

Putting a worker to sleep and waking it up again is quite expensive. Compared to other benchmarks, workers in *LuFact* and *MolDyn* run out of work more often (Table 11.1).



(a) Results for the Intel Core2 Duo machine



(b) Results for the Intel Nehalem machine

Figure 11.1.: Threads and intervals benchmarks running on our Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2) test machines

	LuFact (C)	*MolDyn* (B)	*Crypt* (C)	*SOR* (C)	*RayTracer* (B)
Idle	3 533	271	9	6	6
Woke up	3 531	269	7	4	4

(a) Results for the Intel Core2 Duo machine

	LuFact (C)	*MolDyn* (B)	*Crypt* (C)	*SOR* (C)	*RayTracer* (B)
Idle	11 226	1 057	27	22	17
Woke up	11 218	1 051	19	14	9

(b) Results for the Intel Nehalem machine

Table 11.1.: Number of times workers run out of work, become idle, and get woken up again

11.2. Work-Stealing Queues

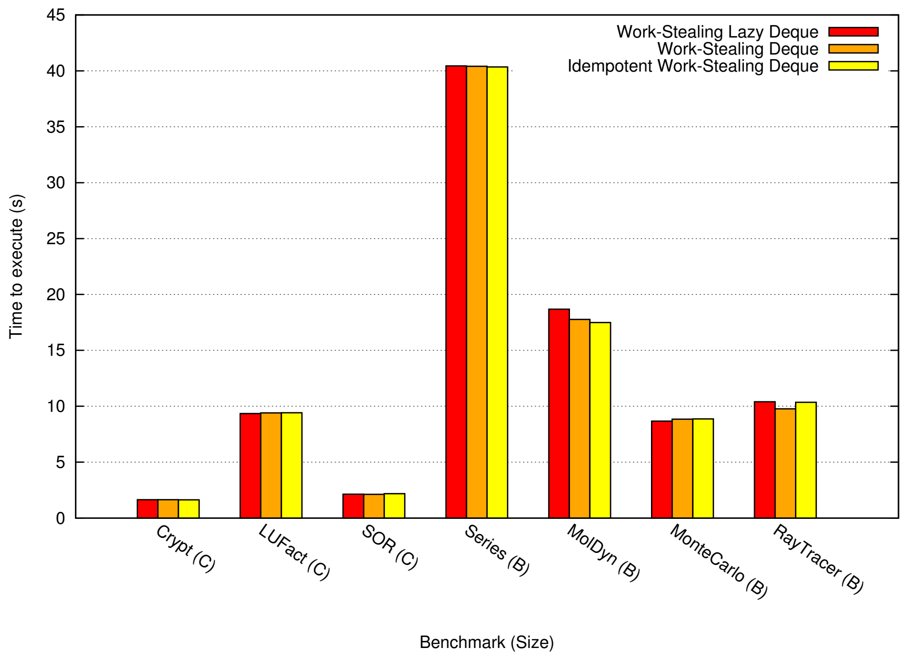
When comparing our implementations of the work-stealing queue, *Work-Stealing Deque* (Section 10.1) and *Idempotent Work-Stealing Deque* (Section 10.2), with the original *Work-Stealing Lazy Deque* (Section 9.2) we do not see a significant difference in the runtime of the JGF benchmarks (Figure 11.2) on both machines we tested them on.

The reason for this is that the *Work-Stealing Lazy Deque* implementation is very efficient already. Using a lock in the `steal()` method is of less impact than we thought it would be.

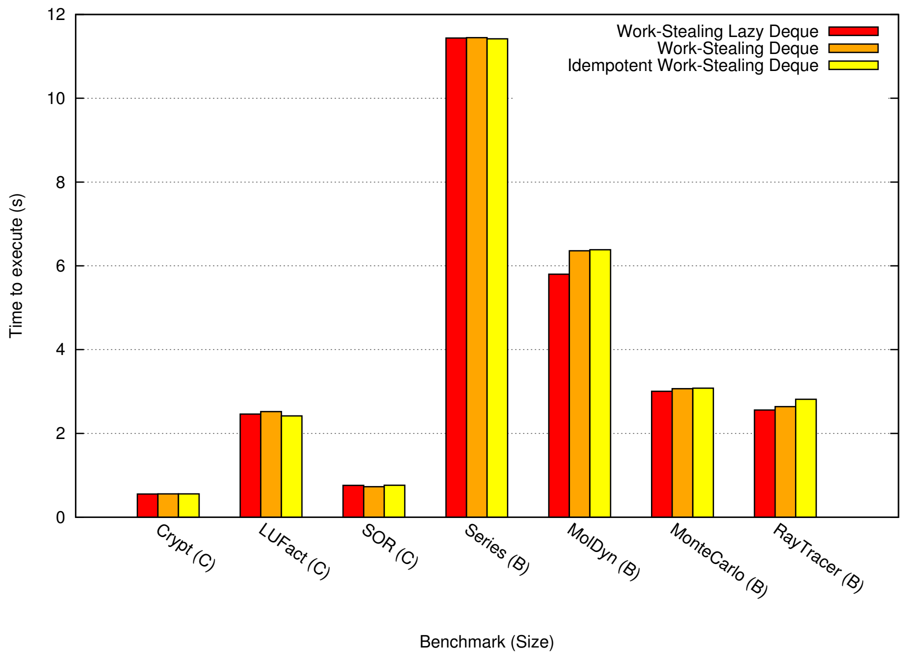
The *Work-Stealing Lazy Deque* uses an `AtomicReferenceArray` to maintain work items. The `AtomicReferenceArray` provides volatile access semantics for its array elements, which is not supported for ordinary arrays. Because of this, *Work-Stealing Lazy Deque* does not have to use additional volatile or atomic variables like the other deque implementations do. This allows for a simpler implementation.

The location of the head of the *Work-Stealing Lazy Deque* is only lazily updated by its owner. This means the location of the head is only updated when its owner tries to take a work item and finds it was stolen by a competing thief – as opposed to the methods `take()` and `steal()` of the *Work-Stealing Deque* which attempt to update the location of the head on every execution with a Compare-and-Swap operation. Similarly the methods `take()` and `steal()` of the *Idempotent Work-Stealing Deque* have to update the size in the anchor on every execution.

Another reason for the lack of difference between the queue implementations might be the small number of cores of the systems we had to test them on. See Section 11.4 for more details.



(a) Results for the Intel Core2 Duo machine



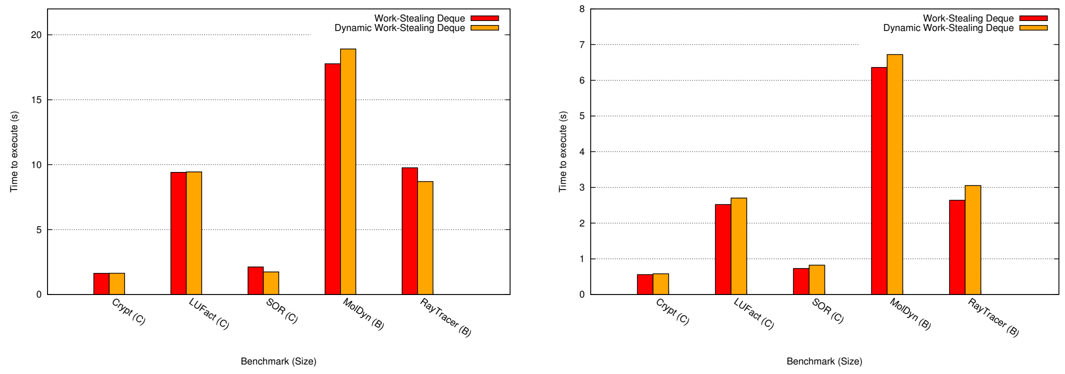
(b) Results for the Intel Nehalem machine

Figure 11.2.: Work-stealing deques benchmarks running on Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2)

11.3. Alternative Work-Stealing Queues

11.3.1. Dynamic Work-Stealing Deque

The implementation of the *Dynamic Work-Stealing Deque* (Section 10.3.1) requires extra work for the list's maintenance which reflects in its performance: Most of the benchmarks are slightly slower in comparison to other queue implementations. Figure 11.3 shows the benchmark runtimes of the *Dynamic Work-Stealing Deque* in comparison to the *Work-Stealing Deque*.



(a) Results for the Intel Core2 Duo machine **(b)** Results for the Intel Nehalem machine

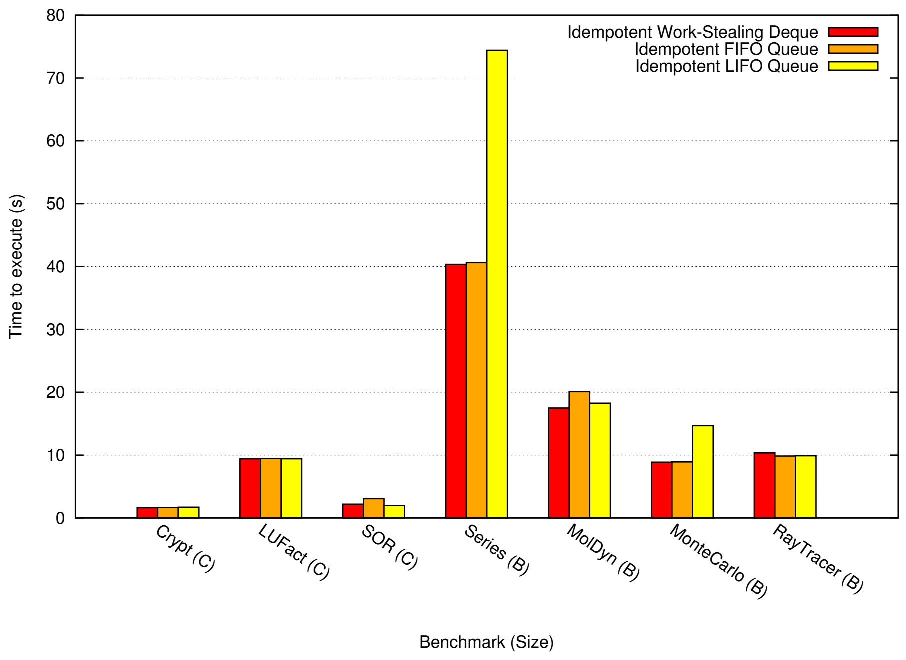
Figure 11.3.: Dynamic work-stealing deques benchmarks running on Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2)

11.3.2. Idempotent Work-Stealing Queues

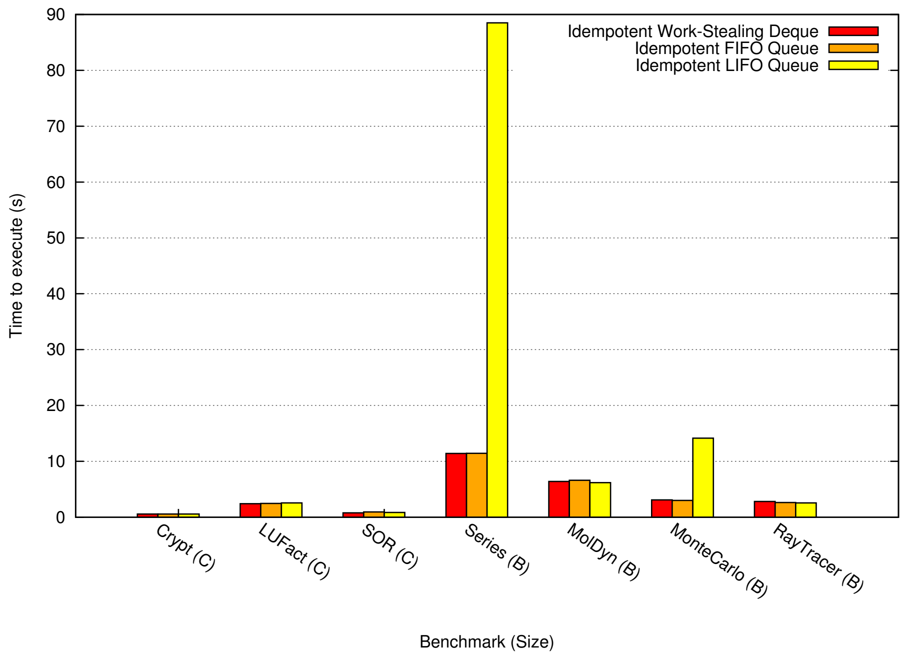
When experimenting with idempotent queues, we also implemented the *Idempotent Work-Stealing FIFO Queue* and *Idempotent Work-Stealing LIFO Queue* to compare them to the *Idempotent Work-Stealing Deque*. It turned out that the *Idempotent Work-Stealing Deque* has the best performance in combination with the intervals scheduler (Figure 11.4).

The *Idempotent Work-Stealing FIFO Queue* behaves like a normal queue: `put()` puts tasks at the tail, and `take()` and `steal()` remove tasks at the head. As `take()` and `steal()` work on the same end of the queue, they have to deal with increased contention. This often reflects itself in a lower performance due to a higher number of take and steal failures in the *Idempotent Work-Stealing FIFO Queue* compared to the *Idempotent Work-Stealing Deque* (Table 11.2). This is especially visible in the runtimes of the *SOR* and *MolDyn* benchmarks (Figure 11.4). In the *Idempotent Work-Stealing Deque* there is less contention between `take()` and `steal()` as they work on opposite ends of the queue.

The *Idempotent Work-Stealing LIFO Queue* behaves like a stack: `put()` puts tasks at the tail, and `take()` and `steal()` remove tasks at the tail too. As `put()`, `take()`



(a) Results for the Intel Core2 Duo machine



(b) Results for the Intel Nehalem machine

Figure 11.4.: Idempotent work-stealing queues benchmarks running on Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2)

	SOR (C)		*MolDyn* (B)	
	Deque	Queue	Deque	Queue
Take Attempts	149 937	150 021	3 662	3 667
Take Failures	504	27 450	607	673
Steal Attempts	650	27 684	615	673
Steal Failures	153	238	265	266

(a) Results for the Intel Core2 Duo machine

	SOR (C)		*MolDyn* (B)	
	Deque	Queue	Deque	Queue
Take Attempts	150 051	150 118	14 197	14 076
Take Failures	2 108	16 293	3 327	4 421
Steal Attempts	4 778	34 190	12 742	15 095
Steal Failures	2 734	17 930	10 490	11 718

(b) Results for the Intel Nehalem machine

Table 11.2.: Comparing take and steal attempts and failures of the *Idempotent Work-Stealing Deque* with the *Idempotent Work-Stealing FIFO Queue*

	Series (C)		*MonteCarlo* (B)	
	Deque	Stack	Deque	Stack
Take Attempts	100 037	100 008	60 078	60 005
Take Failures	49 705	78	29 979	41
Steal Attempts	49 921	886 483 626	30 257	151 331 850
Steal Failures	223	886 483 555	283	151 331 813

(a) Results for the Intel Core2 Duo machine

	Series (C)		*MonteCarlo* (B)	
	Deque	Stack	Deque	Stack
Take Attempts	101 446	100 049	61 989	60 017
Take Failures	88 867	175	54 333	125
Steal Attempts	391 742	436 934 995	244 771	71 468 669
Steal Failures	302 891	436 934 868	190 454	71 468 560

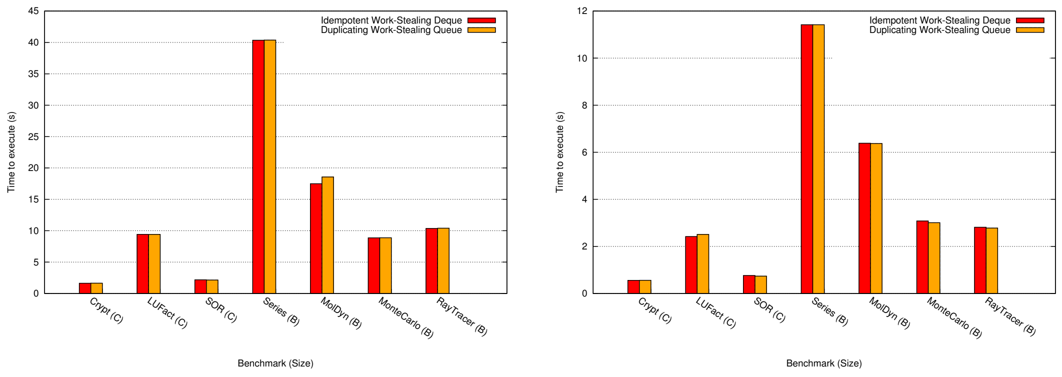
(b) Results for the Intel Nehalem machine

Table 11.3.: Comparing take and steal attempts and failures of the *Idempotent Work-Stealing Deque* with the *Idempotent Work-Stealing LIFO Queue*

and `steal()` work on the same end of the queue, they have to deal with increased contention. This especially shows in the performance of the *Series* and *MonteCarlo* benchmarks with an extremely high number of steal attempts and failures compared to the *Idempotent Work-Stealing Deque* (Table 11.3).

11.3.3. Duplicating Work-Stealing Queue

We implemented the *Duplicating Work-Stealing Queue* (Section 10.3.2) as an alternative to the *Idempotent Work-Stealing Deque*. Both use idempotent intervals, but whereas the idempotent work-stealing deque relies on atomic Compare-and-Swap operations and uses a tag to prevent the ABA problem, the duplicating work-stealing queue uses a lock on all but the critical paths. Despite the usage of locks in the duplicating work-stealing queue we could not find any significant performance difference to the idempotent work-stealing deque (Figure 11.5).



(a) Results for the Intel Core2 Duo machine **(b)** Results for the Intel Nehalem machine

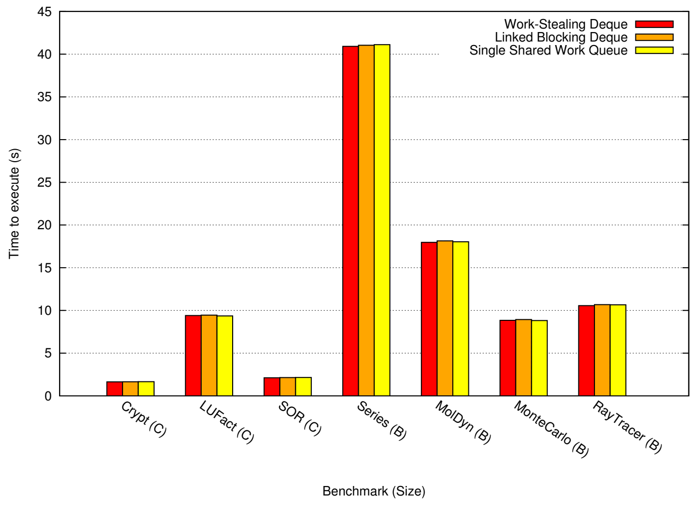
Figure 11.5.: Duplicating benchmarks running on Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2)

11.4. Single Shared Work Queue

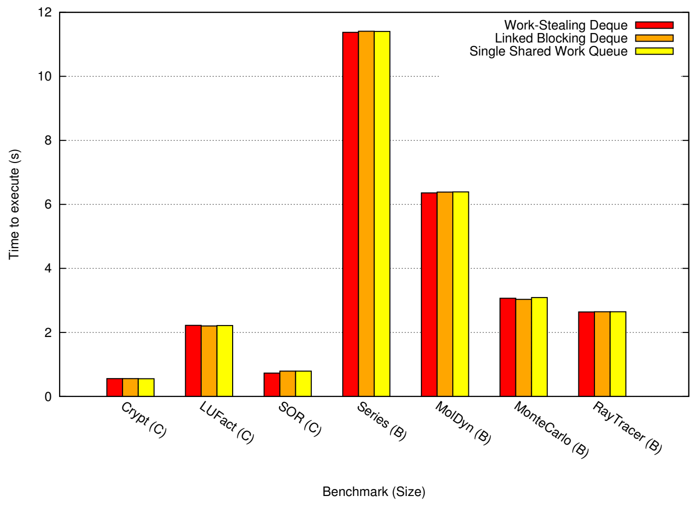
None of the work-stealing queues we developed significantly improves work-stealing performance (Section 11.2) on the machines we had to test them with (Appendix A). Saha et al. [60] state that there is no noticeable difference between the speedup of work-stealing and a global shared work queue when not using more than 8 cores. If this is the case, then the optimizations we did in our work-stealing queue implementations most likely will not have any significant effect either.

To check this hypothesis, we rewrote the intervals scheduler to use a single shared work deque. The shared work deque is implemented with `LinkedBlockingDeque` from the `java.util.concurrent` package in a stack-like manner: Items are added with `addLast()` at the end of the queue and taken with `pollLast()` from the end of the queue. For comparison we added another work-stealing implementation changing the original intervals scheduler to use `LinkedBlockingDeque`.

Our experiments confirm the hypothesis for the JGF benchmarks. There is no significant difference between the runtimes of the different implementations: Whether we use work-stealing with the *Work-Stealing Deque* or the *Linked Blocking Deque*, or the *Single Shared Work Queue* instead of work-stealing, we get almost the same results (Figure 11.6).



(a) Results for the Intel Core2 Duo machine



(b) Results for the Intel Nehalem machine

Figure 11.6.: Shared queue benchmarks running on Intel Core2 Duo (Appendix A.1) and Intel Nehalem (Appendix A.2)

To really study the impact of work-stealing queues on the intervals implementation, we need to do extend our experiments to machines with more than 8 cores.

Chapter 12

Conclusions and Future Work

12.1. Conclusions

Our hypothesis was that we could improve the performance of the intervals scheduler with non-blocking work-stealing queues. Thus, we designed and implemented several work-stealing queues.

We evaluated the performance of our queue implementations 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: The runtimes of the benchmarks using *Work-Stealing Deque* and *Idempotent Work-Stealing Deque* compared to the original *Work-Stealing Lazy Deque* are almost the equal. The *Dynamic Work-Stealing Deque*, *Idempotent FIFO Work-Stealing Queue*, and *Idempotent LIFO Work-Stealing Queue* are often significantly slower than the other queues. The *Duplicating Work-Stealing Queue*'s performance is comparable to the *Idempotent Work-Stealing Deque*.

Apart from the more complex implementation in comparison to the original *Work-Stealing Lazy Deque*, a possible reason for this could be the rather small number of cores of our benchmark machines (Appendix A). As Saha et al. [60] state, there is no noticeable difference between the speedup of work-stealing and a global shared work queue when not using more than 8 cores. Our experiments confirmed those findings for the JGF benchmarks: There is no significant difference between the runtimes of the different implementations – whether we use work-stealing or the single shared work queue, we get almost the same results.

12.2. Future Work

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 [23].

One disadvantage of the queues using arrays to maintain work items is that they do not shrink them again once the queue contains less work items which might lead to a waste of memory. Chase and Lev [9] present a way of shrinking an array without copying by keeping the reference to the smaller array when expanding. When the queue shrinks back its array to the previous array, only the elements that were modified while the larger array was active need to be copied.

Another optimization Chase and Lev [9] mention is 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.

Part III.

Appendices

Appendix A

Experimental Setup

The performance results were obtained on two different machines, an Intel Core2 Duo and an Intel Nehalem system.

A.1. Intel Core2 Duo

The machine has one Intel Core2 Duo Processor T9550 with two cores and 4 GB RAM. Each core has its separate level 1 cache and the 6 MB level 2 cache is shared between the two cores (Figure A.1).

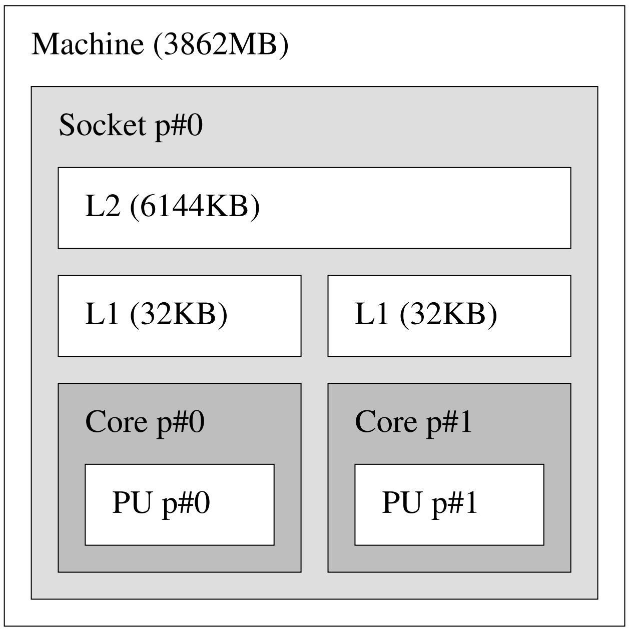


Figure A.1.: Intel Core2 Duo

The system runs Ubuntu 10.04 64-bit with kernel 2.6.32 and the JVM used is Sun Hotspot JDK 1.6.0_20.

A.2. Intel Nehalem

This system includes two Intel Xeon E5520 quad-core processors based on the Intel Nehalem micro architecture. The machine has 12 GB RAM and each processor has a direct connection to half of the memory space via an integrated memory controller (Figure A.2). Additionally, each processor has two QuickPath Interconnect interfaces [40], one connecting to the remote processor and one to the Input/Output hub.

While every core has its separate level 1 and level 2 caches, the per-processor 8 MB level 3 cache is shared between all cores of the same processor. The level 3 cache and the memory controllers of a processor are considered to be a separate subsystem and the Intel documentation refers to this subsystem as the *uncore*.

We sometimes refer to a processor as a node to emphasize its participation in forming a (large-scale) parallel system.

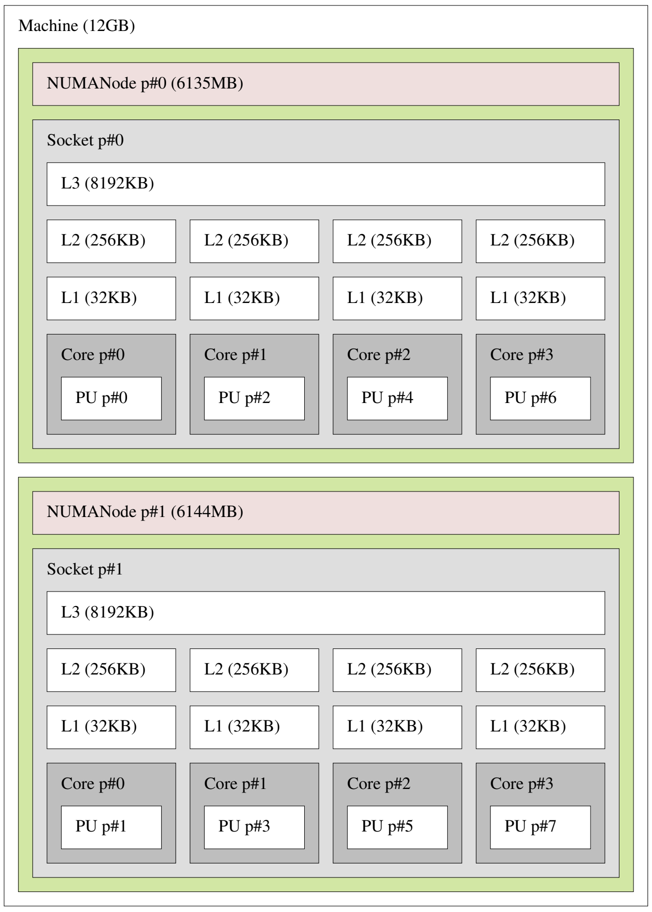


Figure A.2.: Intel Nehalem in a two-processor configuration

The system runs Ubuntu 9.04 64-bit with kernel 2.6.29 patched to support perfmon2 [15] and the JVM used is Sun Hotspot JDK 1.6.0_20.

Appendix B

Java Grande Forum Benchmarks

We evaluate the intervals scheduler implementation with a variety of parallel Java Grande Forum benchmarks [62, 44, 12] that have been ported to use intervals. The remainder of this chapter is a summary of the benchmarks as described in [62].

Crypt

Crypt performs IDEA encryption and decryption of an array of N bytes. This algorithm involves two principle loops, whose iterations are independent and are divided using fork-join sections between intervals in a block fashion.

LUFact

This benchmark solves an $N \times N$ linear system using LU factorization followed by a triangular solve. Iterations of the double loop over the trailing block of the matrix are independent and the work is divided between intervals in a cyclic fashion using fork-join sections.

SOR

The benchmark performs iterations of successive over-relaxations on an $N \times N$ grid. It features an outer loop over iterations and two inner loops, each looping over the grid. To parallelize the loop over array rows, a “red-black” ordering mechanism is used. The work is distributed between intervals in a block manner with help of point-to-point synchronization.

Series

This benchmark computes the first N Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval $[0, 2]$. It uses fork-join sections to distribute the loop over the Fourier coefficients between intervals.

MolDyn

The *MolDyn* benchmark models particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The calculation is distributed between intervals in a cyclic manner and synchronization is done using barriers.

MonteCarlo

This benchmark is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The work is divided between intervals by using fork-join sections.

RayTracer

The *RayTracer* benchmark measures the performance of a 3D ray tracer rendering a scene containing 64 spheres at a resolution of $N \times N$ pixels. The loop over rows of pixels is distributed to intervals using fork-join sections.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. “The data locality of work stealing”. In: *SPAA ’00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. Bar Harbor, Maine, United States: ACM, 2000, pp. 1–12.

[2] Shivali Agarwal et al. “Static Detection of Place Locality and Elimination of Runtime Checks”. In: *APLAS ’08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. Bangalore, India: Springer-Verlag, 2008, pp. 53–74.

[3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. “Adaptive work stealing with parallelism feedback”. In: *PPoPP ’07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. San Jose, California, USA: ACM, 2007, pp. 112–120.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. “Thread scheduling for multiprogrammed multiprocessors”. In: *SPAA ’98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129.

[5] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. “Provably efficient scheduling for languages with fine-grained parallelism”. In: *J. ACM* 46.2 (1999), pp. 281–321.

[6] Robert D. Blumofe and Charles E. Leiserson. “Scheduling multithreaded computations by work stealing”. In: *J. ACM* 46.5 (1999), pp. 720–748.

[7] Robert D. Blumofe et al. “Cilk: an efficient multithreaded runtime system”. In: *SIGPLAN Not.* 30.8 (1995), pp. 207–216.

[8] Philippe Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. San Diego, CA, USA: ACM, 2005, pp. 519–538.

[9] David Chase and Yossi Lev. “Dynamic circular work-stealing deque”. In: *SPAA ’05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. Las Vegas, Nevada, USA: ACM, 2005, pp. 21–28.

[10] Shimin Chen et al. “Scheduling threads for constructive cache sharing on CMPs”. In: *SPAA ’07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. San Diego, California, USA: ACM, 2007, pp. 105–115.

[11] Gilberto Contreras and Margaret Martonosi. “Characterizing and Improving the Performance of Intel Threading Building Blocks”. In: *Electrical Engineering* (2008), pp. 1–10.

[12] Charles Daly et al. “Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark suite”. In: *JGI ’01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. Palo Alto, California, United States: ACM, 2001, pp. 106–115.

[13] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. “The JCilk Language for Multithreaded Computing”. In: *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. San Diego, California 2005.

[14] Eli Dow. *Take charge of processor affinity*. 2005. URL: http://www.ibm.com/developerworks/linux/library/l-affinity.html (visited on 08/21/2010).

[15] Stéphane Eranian. “What can performance counters do for memory subsystem analysis?” In: *MSPC ’08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*. Seattle, Washington: ACM, 2008, pp. 26–30.

[16] Christine H. Flood et al. “Parallel garbage collection for shared memory multiprocessors”. In: *JVM’01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*. Monterey, California: USENIX Association, 2001, pp. 21–21.

[17] Annie Foong, Jason Fung, and Don Newell. *Improved Linux SMP Scaling: User-directed Processor Affinity*. 2008. URL: http://software.intel.com/en-us/articles/improved-linux-smp-scaling-user-directed-processor-affinity/ (visited on 08/21/2010).

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The implementation of the Cilk-5 multithreaded language”. In: *SIGPLAN Not.* 33.5 (1998), pp. 212–223.

[19] Fabien Gaud et al. *Mely: Efficient Workstealing for Multicore Event-Driven Systems*. Research Report. INRIA, 2010, p. 23. URL: http://hal.inria.fr/inria-00449530/PDF/RR-7169.pdf.

[20] Yi Guo et al. “SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems”. In: *SIGPLAN Not.* 45.5 (2010), pp. 341–342.

[21] Yi Guo et al. “Work-first and help-first scheduling policies for async-finish task parallelism”. In: *IPDPS ’09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[22] *Habanero Java*. URL: http://habanero.rice.edu/hj (visited on 08/21/2010).

[23] Danny Hendler and Nir Shavit. “Non-blocking steal-half work queues”. In: *PODC ’02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*. Monterey, California: ACM, 2002, pp. 280–289.

[24] Danny Hendler et al. *A dynamic-sized nonblocking work stealing deque*. Research Report. Mountain View, CA, USA: Sun Microsystems, Inc., 2005. URL: http://labs.oracle.com/techrep/2005/abstract-144.html.

[25] Maurice Herlihy. “Wait-free synchronization”. In: *ACM Trans. Program. Lang. Syst.* 13.1 (1991), pp. 124–149.

[26] Charles Humble. *Sun Releases Java 6 Update 18 With Significant Performance Improvements and Windows 7 Support*. 2010. URL: http://www.infoq.com/news/2010/01/java6u18 (visited on 08/21/2010).

[27] IBM. *IBM System/370 Principles of Operation*. 4th. 1974.

[28] Andi Kleen. “An NUMA API for Linux”. In: *SUSE Labs* (2004).

[29] Donald Knuth. “Double-Ended Queue”. In: *The Art of Computer Programming: Volume 1 - Fundamental Algorithms*. 3rd. Addison-Wesley, 1997, pp. 238–243.

[30] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[31] Doug Lea. “A Java fork/join framework”. In: *JAVA ’00: Proceedings of the ACM 2000 conference on Java Grande*. San Francisco, California, United States: ACM, 2000, pp. 36–43.

[32] Doug Lea. *Concurrency JSR-166 Interest Site*. 2006. URL: http://gee.cs.oswego.edu/dl/concurrency-interest/index.html (visited on 08/21/2010).

[33] Doug Lea. *Fork/Join Parallelism in Java*. 2000. URL: http://gee.cs.oswego.edu/dl/cpjslides/fj.pdf (visited on 08/21/2010).

[34] Doug Lea et al. *JSR 166: Concurrency Utilities*. 2004. URL: http://jcp.org/en/jsr/detail?id=166 (visited on 08/21/2010).

[35] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. “The design of a task parallel library”. In: *SIGPLAN Not.* 44.10 (2009), pp. 227–242.

[36] Yossi Lev and D.R. Chase. *Nonblocking Cyclic Extendable Deque for the ABP work stealing algorithm*. 2005. URL: http://www.cs.brown.edu/~levyossi/Pubs/MyTalks/ExtendebleCircularDeque.ppt (visited on 08/21/2010).

[37] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. 2009. URL: http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf (visited on 08/21/2010).

[38] Vasileios Liaskovitis et al. "Brief Announcement: Parallel Depth First vs. Work Stealing Schedulers on CMP Architectures". 2006.

[39] Robert Love. *CPU Affinity*. 2003. URL: http://www.linuxjournal.com/article/6799 (visited on 08/21/2010).

[40] R. A. Maddox, G. Singh, and R. J. Safranek. *A First Look at the Intel QuickPath Interconnect*. 2009. URL: http://www.intel.com/intelpress/files/A_First_Look_at_the_Intel(r)_QuickPath_Interconnect.pdf (visited on 08/21/2010).

[41] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java memory model". In: *SIGPLAN Not.* 40.1 (2005), pp. 378–391.

[42] Jason Mars et al. "Contention aware execution: online contention detection and response". In: *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. Toronto, Ontario, Canada: ACM, 2010, pp. 257–265.

[43] Jon Masamitsu. *Help for the NUMA Weary*. 2008. URL: http://blogs.sun.com/jonthecollector/entry/help_for_the_numa_weary (visited on 08/21/2010).

[44] J. A. Mathew, P. D. Coddington, and K. A. Hawick. "Analysis and development of Java Grande benchmarks". In: *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*. San Francisco, California, United States: ACM, 1999, pp. 72–80.

[45] Nicholas D. Matsakis. "Formal Definition and Safety Proof for the Interval and Effect Analysis". 2009.

[46] Nicholas D. Matsakis. *Intervals*. 2010. URL: http://intervals.inf.ethz.ch/ (visited on 08/21/2010).

[47] Nicholas D. Matsakis and Thomas R. Gross. "Handling Errors in Parallel Programs Based on Happens Before Relations". In: *15th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2010)*. 2010.

[48] Nicholas D. Matsakis and Thomas R. Gross. "Programming with Intervals". In: *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC 2009)*. 2009.

[49] Nicholas D. Matsakis and Thomas R. Gross. "Reflective Parallel Programming". In: *HotPar '10: 2nd USENIX Workshop on Hot Topics in Parallelism*. 2010.

[50] Maged M. Michael. “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”. In: *IEEE Trans. Parallel Distrib. Syst.* 15.6 (2004), pp. 491–504.

[51] Maged M. Michael and Michael L. Scott. “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. In: *PODC ’96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. Philadelphia, Pennsylvania, United States: ACM, 1996, pp. 267–275.

[52] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. “Idempotent work stealing”. In: *PPoPP ’09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Raleigh, NC, USA: ACM, 2009, pp. 45–54.

[53] Mark Moir. “Practical implementations of non-blocking synchronization primitives”. In: *PODC ’97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*. Santa Barbara, California, United States: ACM, 1997, pp. 219–228.

[54] OpenMPI. *Portable Hardware Locality (hwloc)*. 2010. URL: `http://www.open-mpi.org/projects/hwloc/` (visited on 08/21/2010).

[55] OpenMPI. *Portable Linux Processor Affinity (PLPA)*. 2010. URL: `http://www.open-mpi.org/projects/plpa/` (visited on 08/21/2010).

[56] Oracle. *Bug ID: 4234402 (Thread.setCPUAffinity)*. 1999. URL: `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4234402` (visited on 08/21/2010).

[57] Oracle. *Java HotSpot Virtual Machine Performance Enhancements - JDK 7*. 2010. URL: `http://download.oracle.com/docs/cd/E17409_01/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html` (visited on 08/21/2010).

[58] James Philbin et al. “Thread scheduling for cache locality”. In: *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. Cambridge, Massachusetts, United States: ACM, 1996, pp. 60–71.

[59] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O’Reilly Media, Inc., 2007. ISBN: 0596514808.

[60] Bratin Saha et al. “Enabling scalability and performance in a large scale CMP environment”. In: *SIGOPS Oper. Syst. Rev.* 41.3 (2007), pp. 73–86.

[61] Vijay Saraswat. *Report on the Programming Language X10*. Language Specification. IBM, 2010. URL: `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`.

[62] L. A. Smith, J. M. Bull, and J. Obdrzálek. “A Parallel Java Grande Benchmark Suite”. In: *Supercomputing ’01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. Denver, Colorado: ACM, 2001, pp. 8–8.

[63] M. S. Squillante and E. D. Lazowska. “Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling”. In: *IEEE Trans. Parallel Distrib. Syst.* 4.2 (1993), pp. 131–143.

[64] Yonghong Yan et al. “Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement”. In: *LCPC.* 2009, pp. 172–187.

[65] Nickolai Zeldovich et al. “Multiprocessor Support for Event-Driven Programs”. In: *Proc. USENIX 2003 Annual Technical Conference, June 2003.* 2003.