NORTHWESTERN UNIVERSITY

Incremental Parallelization of Existing Sequential Runtime Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering and Computer Science Department

By

James Swaine

EVANSTON, ILLINOIS

June 2014

ABSTRACT

Incremental Parallelization of Existing Sequential Runtime Systems

James Swaine

Many language implementations, particularly for high-level and scripting languages, are based on carefully honed runtime systems that have an internally sequential execution model. Adding support for parallelism in the usual form—as threads that run arbitrary code in parallel—would require a major revision or even a rewrite to add safe and efficient locking and communication. This dissertation describes an alternative approach to *incremental parallelization* of runtime systems. This approach can be applied inexpensively to many sequential runtime systems, and we demonstrate its effectiveness in the Racket runtime system and Parrot virtual machine. The evaluation assesses performance benefits, developer effort needed to implement such a system in these two runtime systems, and the ease with which users can leverage the resulting parallel programming constructs without sacrificing expressiveness. We find that incremental parallelization can provide useful, scalable parallelism on commodity multicore processors at a fraction of the effort required to implement conventional parallel threads.

Table of Contents

Part 1

Runtime Systems from a Sequential Era

Many modern high-level or scripting languages are implemented around an interpretive runtime system, often with a JIT compiler. Examples include the Racket [18] runtime system, the Parrot virtual machine, and the virtual machines underlying Perl, Python, Ruby, and other productivity-oriented languages. These runtime systems are often the result of many man-years of effort, and they have been carefully tuned for capability, functionality, correctness, and performance.

For the most part, such runtime systems have not been designed to support parallelism on multiple processors. Even when a language supports constructs for concurrency, they are typically implemented through co-routines or OS-level threads that are constrained to execute one at a time. This limitation has become a serious issue, as it is clear that exploiting parallelism is essential to harnessing performance in future processor generations. Whether computer architects envision the future as involving homogeneous or heterogeneous multicores, and with whatever form of memory coherence or consistency model, the common theme is that the future is parallel and that language implementations must adapt. The essential problem is making the language implementation safe for low-level parallelism, i.e., ensuring that even when two threads are modifying internal data structures at the same time, the runtime system behaves correctly.

One approach to enabling parallelism would be to allow existing concurrency constructs to run in parallel, and to rewrite or revise the runtime system to carefully employ locking or explicit communication. Experience with that approach, as well as the persistence of the global interpreter lock in implementations for Python and Ruby, suggests that such a conversion is extremely difficult to perform correctly. Based on the even longer history of experience in parallel systems, one would also expect the result to scale poorly as more and more processors become available. The alternative of simply throwing out the current runtime and re-designing and implementing it around a carefully designed concurrency model is no better, as it would require discarding years or decades

of effort in building an effective system, and this approach also risks losing much of the language's momentum as the developers are engaged in tasks with little visible improvement for a long period.

This dissertation investigates a new technique for parallelizing runtime systems, called *slow-path barricading*. The technique is based on the observation that the core of many programs—and particularly the part that runs fast sequentially and can benefit most from parallelism—involves relatively few side effects with respect to the language implementation's internal state. Thus, instead of wholesale conversion of the runtime system to support arbitrary concurrency, we add language constructs that focus and restrict concurrency where the implementation can easily support it.

Specifically, the set of primitives in a language implementation is partitioned into *safe* (for parallelism) and *unsafe* categories. The programmer is then given a mechanism to start a parallel task; as long as the task sticks to safe operations, it stays in the so-called *fast path* of the implementation and thus is safe for parallelism. As soon as the computation hits a barricade, the runtime system suspends the computation until the operation can be handled in the more general, purely sequential part of the runtime system.

Although the programming model allows only a subset of language operations to be executed in parallel, this subset roughly corresponds to the set of operations that the programmer already knows (or should know) to be fast in sequential code. Thus, a programmer who is reasonably capable of writing fast programs in the language already possesses the knowledge to write a program that avoids unsafe operations—and one that therefore exhibits good scaling for parallelism. Furthermore, this approach enables clear feedback to the programmer about when and how a program uses unsafe operations.

CHAPTER 1

Thesis Statement

The thesis of this dissertation is:

We can incrementally add effective parallel programming primitives and tool support to legacy sequential runtime systems with a modest investment of effort.

The following chapters support the hypothesis by demonstrating the validity of each of the adjectives found in the thesis statement.

1.1. Incremental

A straightforward, conservative implementation of slow-path barricading yields a system that supports parallel threads, but may not scale because a majority of the language's primitives are barricaded. The key advantage to adding parallelism in this way is that the runtime system implementer can first build a prototype in a fraction of the time required by wholesale rewriting and then work on un-barricading primitives one at a time. Instead of incurring a huge initial overhead with little chance of producing a bug-free, stable initial release, we incur a small one with a good chance of producing a stable (but slow) release and then amortize the remainder of the work. Meanwhile, our users are already using the system and providing feedback regarding both performance and correctness bugs—allowing us to better prioritize which primitives receive the most attention in terms of development man-hours.

1.2. Effective Parallel Programming Primitives

Performance is the most important determinant of the effectiveness of any parallel system. This dissertation will demonstrate, through a series of benchmark implementations and a real-world example, that slow-path barricading implementations in both the Racket and Parrot runtime systems can be used to produce parallel programs which scale well (i.e. they perform within a constant factor of an equivalent implementation in some other well-established language with hardware-threading support such as C or Java).

1.3. Effective Tool Support

While the runtime system implementer works to eliminate barricades from the system, users are left with the problem of how to deal with an implementation which may sometimes restrict parallelism in unexpected ways. If a well-written parallel algorithm is not scaling, the programmer must be able to identify which barricaded primitives are the source of the problem. We show how Racket's parallel profiler, a graphical tool showing performance information about futures, can be used to tune a program that is not scaling due to some limitation of the barricading implementation.

1.4. Modest Investment

Building a stable, correct runtime system is both complicated and time-consuming; retrofitting one to support parallel execution is equally, if not more, difficult. Few sequential language runtime systems have successfully made the transition to a truly parallel implementation, with the Java Virtual Machine being one notable exception [14]. Instead, increasingly exotic methods have been explored in order to deliberately avoid the problem of pervasive rewriting [33]. The slow-path barricading technique makes this undertaking tractable, and details concerning the development overhead incurred in both the Racket and Parrot implementations are given in part V.

CHAPTER 2

Overview

Part II introduces the languages and API's used throughout this dissertation, including the Racket language and its associated parallel libraries and utilities. Part III demonstrates the use of these tools by working through several example programs. Part IV explains in detail the implementation of futures, both in a general sense (for any runtime system) and specific sense (Racket and Parrot), and their accompanying profiling tools. Part V evaluates the futures approach both in terms of its development overhead and the performance of the two libraries our applications of it produced. Part VI surveys related work in high-level parallel languages and runtime systems, as well as relevant work in profiling and visualization tools for parallel systems.

Part 2

Background

CHAPTER 3

Racket

The Racket language is a programming language in the Lisp/Scheme family.[1] It is multi-paradigm, allowing programmers to mix object-oriented, functional, or imperative styles in whatever way is appropriate for the task at hand. Its rich macro system serves as a platform for language design, and the Racket core ships with a litany of specialized languages built using this system. For example, this document was built using `scribble` [17], a documentation-generation language, for typesetting; and `slideshow` [16], a language enabling programmatic assembly of slide-based presentations used here for figure rendering. Though plain-vanilla Racket is dynamically typed, the Typed Racket language [41] offers a statically typed variant. Subsequent chapters demonstrate how this static typing can be leveraged to mitigate some of the restrictions imposed by slow-path barricades.

The following Racket program defines a function which can compute the *n*th Fibonacci number:

```
(define (fib/rec number prev1 prev2)
  (if (<= number 0)
      prev2
      (fib/rec (- number 1) (+ prev1 prev2) prev1)))

(define (fib number)
  (fib/rec number 1 0))
```

[1] `http://racket-lang.org`

```
> (fib 350)
6254449428820551641549772190170184190608177514674331726439961915653414425
```

And the Typed Racket equivalent:

```
(: fib/rec (Integer Nonnegative-Integer
            Nonnegative-Integer -> Nonnegative-Integer))
(define (fib/rec number prev1 prev2)
  (if (<= number 0)
      prev2
      (fib/rec (- number 1) (+ prev1 prev2) prev1)))

(: fib (Integer -> Nonnegative-Integer))
(define (fib number)
  (fib/rec number 1 0))
```

Note that the typed version is identical to the untyped one, except for the addition of function type annotations (via `:`). In most cases the Typed Racket compiler will accept unaltered untyped Racket code after top-level functions have type annotations.

CHAPTER 4

Futures API

A *future* [5] is an object that acts as a proxy or placeholder for a value that is the result of some computation; at some point after the future's creation, the program may ask for the value that is being proxied explicitly, by applying a function such as `touch` or `force` with the future as an argument. The futures described in this dissertation offer *best-effort parallelism*; that is, no guarantee exists that the future's value will be computed in parallel.

Futures are currently the primary means of shared-memory parallel programming in Racket. The language provides `future` to start a parallel computation and `touch` to receive its result:

```
(future thunk) → future?
  thunk : (-> any)
```

Accepts a thunk (i.e. a procedure with no arguments) and may start evaluating it in parallel to the rest of the computation. The return value is a *future descriptor* that can subsequently be touched.

```
(touch ft) → any
  ft : future?
```

Waits for the thunk to complete and returns the value that the thunk produced. If applied to the same future descriptor multiple times, `touch` returns the same result each time (as computed just once by the thunk).

For example, in the program:

```
(define a (powerful-computer))
```

```
(define b (even-more-powerful-computer))

(values
  (life-the-universe-and-everything? a)
  (life-the-universe-and-everything? b))
```

`a` and `b` can compute `life-the-universe-and-everything?` independently. They could be computed in parallel using `future` and `touch` as follows:

```
(define fa (future (λ () (life-the-universe-and-everything? a))))
(define db (future (λ () (life-the-universe-and-everything? b))))

(values
  (touch fa)
  (touch fb))
```

The main computation can proceed in parallel to a future:

```
(define f (future (λ () (life-the-universe-and-everything? a))))
(values
  (life-the-universe-and-everything? b)
  (touch f))
```

There is as much parallelism here as in the version that uses two futures, since `b`'s work will be performed on some thread other than the one evaluating this code. In contrast:

```
(define f (future (λ () (life-the-universe-and-everything? a))))
(values
  (touch f)
  (life-the-universe-and-everything? b))
```

This has no parallelism, because Racket evaluates expressions from left to right; `(life-the-universe-and-everything? b)` is evaluated only after the `(touch f)` expression.

A future's thunk is not necessarily evaluated in parallel to other expressions. In particular, if the thunk's computation relies in some way on the evaluation context, then the computation is

suspended until a `touch`, at which point the computation continues in the context of the `touch`. For example, if a future thunk raises an exception, the exception is raised at the point of the `touch`. (If an exception-raising future is `touch`ed a second time, the second attempt raises a different exception to report that no value is available.)

A future's thunk can perform side effects that are visible to other computations. For example, after

```
(define x 0)
(define (inc!) (set! x (+ x 1)))
(define f1 (future inc!))
(define f2 (future inc!))
(touch f1)
(touch f2)
```

the possible values of `x` include 0, 1, and 2. The `future` and `touch` operations are intended for use with thunks that perform independent computations, though possibly storing results in variables, arrays or other data structures using side effects.

For a slightly more realistic example, figure 1 shows a simple Mandelbrot-set rendering program, a classic embarrassingly-parallel computation.

In an ideal language implementation, the Mandelbrot computation could be parallelized through a future for each point. Figure 2 shows such an implementation, where `for/list` is a list-comprehension form that is used to create a list of list of futures, and then each future is `touch`ed in order. This approach does not improve performance, however, because a single call to `mandelbrot-point` is far simpler than the work of creating a parallel task and communicating the result.

Figure 3 shows a per-line parallelization of the Mandelbrot computation. Each line is rendered independently to a buffer, and then the buffered lines are written in order. This approach is typical

```
(define MAX-ITERS 50)
(define MAX-DIST 2.0)
(define N 1024)

(define (mandelbrot-point x y)
  (define c (+ (- (/ (* 2.0 x) N) 1.5)
               (* +i (- (/ (* 2.0 y) N) 1.0))))
  (let loop ((i 0) (z 0.0+0.0i))
    (cond
      [(> i MAX-ITERS) (char->integer #\*)]
      [(> (magnitude z) MAX-DIST)
       (char->integer #\space)]
      [else (loop (add1 i) (+ (* z z) c))])))

(for ([y (in-range N)])
  (for ([x (in-range N)])
    (write-byte (mandelbrot-point x y)))
  (newline))
```

Figure 1: Sequential Mandelbrot plotting

```
(define fss
  (for*/list ([y (in-range N)]
              [x (in-range N)])
    (future
      (λ () (mandelbrot-point x y)))))
(for ([fs (in-list fss)])
  (for ([f (in-list fs)])
    (write-byte (touch f)))
  (newline))
```

Figure 2: Naive Mandelbrot parallelization

for a system that supports parallelism, and it is a practical approach for the Mandelbrot program in Racket.

```
(define fs
  (for/list ([y (in-range N)])
    (define bstr (make-bytes N))
    (future
     (λ ()
       (for ([x (in-range N)])
         (bytes-set! bstr x (mandelbrot-point x y)))
       bstr))))
(for ([f (in-list fs)])
  (write-bytes (touch f))
  (newline))
```

Figure 3: Per-line Mandelbrot parallelization

Perhaps surprisingly, then, the per-line refactoring for Mandelbrot rendering runs much slower than the sequential version. The problem at this point is not the decomposition approach or inherent limits in parallel communication. Instead, the problem is due to the key compromise between the implementation of Racket and the needs of programmers with respect to parallelization.

Specifically, the problem is that complex-number arithmetic is currently treated as a "slow" operation in Racket, and the implementation makes no attempt to parallelize slow operations, since they may manipulate shared state in the runtime system. Programmers must learn to avoid slow operations within parallel tasks—at least until incremental improvements to the implementation allow the operation to run in parallel.

A programmer can discover the slow operation in this case by enabling debugging profiling, which causes `future` and `touch` to produce output similar to:

```
future: 0 waiting for runtime at 126.741: *
```

The first line of this log indicates that a future computation was suspended because the `*` operation could not be executed in parallel. A programmer would have to consult the documentation to determine that `*` is treated as a slow operation when it is applied to complex numbers.

```
(: mandelbrot-point (Integer Integer -> Integer))
(define (mandelbrot-point x y)
  (define fx (->fl x))
  (define fy (->fl y))
  (define fn (->fl N))
  (define c (make-rectangular
             (- (/ (* 2.0 fx) fn) 1.5)
             (- (/ (* 2.0 fy) fn) 1.0)))
  (let loop : Integer ([i : Integer 0] [z : Float-Complex 0.0+0.0i])
    (cond
      [(> i MAX-ITERS) (char->integer #\*)]
      [else
       (define zq (* z z))
       (if (> (magnitude zq) MAX-DIST)
           (char->integer #\space)
           (loop (add1 i)
                 (+ zq c)))])))
```

Figure 4: Mandelbrot core in Typed Racket

Another way in which an operation can be slow in Racket is to require too much allocation. Debugging-log output of the form:

```
future: 0 waiting for runtime at 126.032: [acquire_gc_page]
```

indicates that a future computation had to synchronize with the main computation to allocate memory. Again, the problem is a result of an implementation compromise, because Racket's memory allocator is basically sequential, although moderate amounts of allocation can be performed in parallel.

Figure 4 shows a Typed Racket version of `mandelbrot-point` for which per-line parallelism offers the expected performance improvement. Note that we are not forced to sacrifice the use of complex numbers. The Typed Racket compiler is smart enough to know that, given a value of

type `Float-Complex`, we can avoid the slow path by extracting each component and performing unchecked arithmetic on each one. The sequence of expressions:

```
(define zq (* z z))
(if (> (magnitude zq) MAX-DIST)
    ...
    ...)
```

Is translated into the following bytecode:

```
(let ((flonum123
       (unsafe-fl-
        (unsafe-fl* arg0-118 arg0-118)
        (unsafe-fl* arg1-119 arg1-119))))
  (let ((flonum130
         (unsafe-fl+
          (unsafe-fl* arg1-119 arg0-118)
          (unsafe-fl* arg0-118 arg1-119))))
    (if (unsafe-fl>
         (unsafe-flsqrt
          (unsafe-fl+
           (unsafe-fl* flonum123 flonum123)
           (unsafe-fl* flonum130 flonum130)))
         '2.0)
         ...)))
```

The generated bytecode has defined two floating-point variables to hold the real and imaginary components of `z`, and performs computations with them using flonum-specific arithmetic (i.e., operations that consume and produce only floating-point numbers). Flonum-specific operations act as a hint to help the compiler "unbox" intermediate flonum results—keeping them in registers or allocating them on a future-local stack, which avoids heap allocation (see chapter 6 for more on mitigating flonum-related issues). Thus we are able to remain on the fast path without sacrificing expressiveness.

This version runs about 30 times as fast as the original version; a programmer who needs performance will always prefer it, whether using futures or not. Meanwhile, for much the same reason that it can run fast sequentially, this version also provides a speedup when run in parallel.

All else being equal, obtaining performance through parallelism is no easier in our design for futures than in other systems for parallel programming. The programmer must still understand the relative cost of computation and communication, and the language's facilities for sequential performance should be fully deployed before attempting parallelization. All else is *not* equal, however; converting the initial Racket program to one that performs well is far simpler than, say, porting the program to C. For more sophisticated programs, where development in Racket buys productivity from the start, futures provide a transition path to parallelism that keep those productivity benefits intact. Most importantly, our approach to implementing futures makes these benefits available at a tractable cost for the implementer of the programming language.

Part 3

Programming with Futures

CHAPTER 5

Conway's Game of Life

As a more complete example, we will implement John Conway's cellular automaton "Game of Life" [22]. With a simple set of rules and no dependencies between loop iterations, this system is ideal for parallelization in the same vein as the Mandelbrot set.

5.1. The Rules

The game is played on a two-dimensional grid (universe) of *cells*, each of which may be in one of two states: *alive* or *dead*. Though the universe can have infinite size, we will model it here as a fixed-size toroidal (wrapping) grid for simplicity. Changes to the state of the universe over time occur in discrete time steps, where on each step the following rules are applied:

- **Underpopulation**. Any cell with fewer than two live neighbors dies.
- **Survival**. Any live cell with two or three live neighbors survives to the next generation.
- **Overpopulation**. Any live cell with greater than three neighbors dies.
- **Reproduction**. Any dead cell with exactly three live neighbors becomes a live cell.

Give some initial universe state (the *seed*), rules are applied on each time step instantaneously to all cells in the grid.

5.2. Racket Implementation

Given these rules, it is straightforward to build a sequential implementation. The Racket code in figure 5 shows one such implementation: because rules must be applied to all cells simultaneously,

```
(define (step univ)
  (define sz (universe-size univ))
  (for* ([r (in-range sz)]
         [c (in-range sz)])
    (define nbs (live-neighbor-count univ r c))
    (update-cell! univ r c
                  (cond
                    [(alive? univ r c)
                     (case nbs
                       [(2 3) 'alive]
                       [else 'dead])]
                    [else
                     (if (= 3 nbs)
                         'alive
                         'dead)])))
  (send (universe-bmp univ)
        set-argb-pixels
        0 0 sz sz
        (universe-write-buf univ))
  (swap-buffers! univ))
```

Figure 5: Sequential time-step function

the code uses a double-buffered approach, where reads are confined to the universe's `read-buf` (the current universe state) and writes confined to `write-buf`.

The neighbor-counting function is given in figure 6— we simply take the sum of the return values of `one-if-alive` for all the current cell's neighbors.

```
1 (define (one-if-alive univ r c)
2   (if (alive? univ r c)
3       1
4       0))
5
6 (define (live-neighbor-count univ r c)
7   (define MAX-IND (- (universe-size univ) 1))
8   (define prev-row (if (zero? r) MAX-IND (- r 1)))
9   (define next-row (if (< r MAX-IND) (+ r 1) 0))
10  (define prev-col (if (zero? c) MAX-IND (- c 1)))
11  (define next-col (if (< c MAX-IND) (+ c 1) 0))
12  (+ (one-if-alive univ prev-row prev-col)
13     (one-if-alive univ prev-row c)
14     (one-if-alive univ prev-row next-col)
15     (one-if-alive univ r next-col)
16     (one-if-alive univ next-row next-col)
17     (one-if-alive univ next-row c)
18     (one-if-alive univ next-row prev-col)
19     (one-if-alive univ r prev-col)))
```

Figure 6: Naive neighbor-counting code

Parallelizing the `step` Function

As noted previously, the time-stepping logic in the game of life is embarrassingly parallel—cell updates in a given step can occur in any order, and only depend on the state of the universe in `read-buf`. Thus, it is trivial to rewrite `step` to perform universe updates using statically-scheduled parallel futures. Our new-and-improved version is shown in figure 7. Our inner loop remains unchanged from the sequential version; the only addition is the `for/list` outer loop on line 5,

```
1 (define (step univ)
2   (define sz (universe-size univ))
3   (define chunk-sz (quotient sz P))
4   (define fs
5     (for/list ([p (in-range P)])
6       (define start (* chunk-sz p))
7       (future
8        (λ ()
9          (for* ([r (in-range start (+ start chunk-sz))]
10                 [c (in-range sz)])
11            (define nbs (live-neighbor-count univ r c))
12            (update-cell! univ r c
13                   (cond
14                     [(alive? univ r c)
15                      (case nbs
16                        [(2 3) 'alive]
17                        [else 'dead])]
18                     [else
19                      (if (= 3 nbs)
20                          'alive
21                          'dead)])))))))
22   (for-each touch fs)
23   (send (universe-bmp univ)
24         set-argb-pixels
25         0 0 sz sz
26         (universe-write-buf univ))
27   (swap-buffers! univ))
```

Figure 7: Parallel time-step function with static scheduling

which gives us a list of futures as large as the number of processors in the machine. We then wait on all of them to finish in line 22.

We can invoke Racket's futures visualizer using the following script, which just advances the state of the universe 10 times for a large (2048 x 2048) universe:

```
(define au (acorn-universe 2048))
(visualize-futures
```

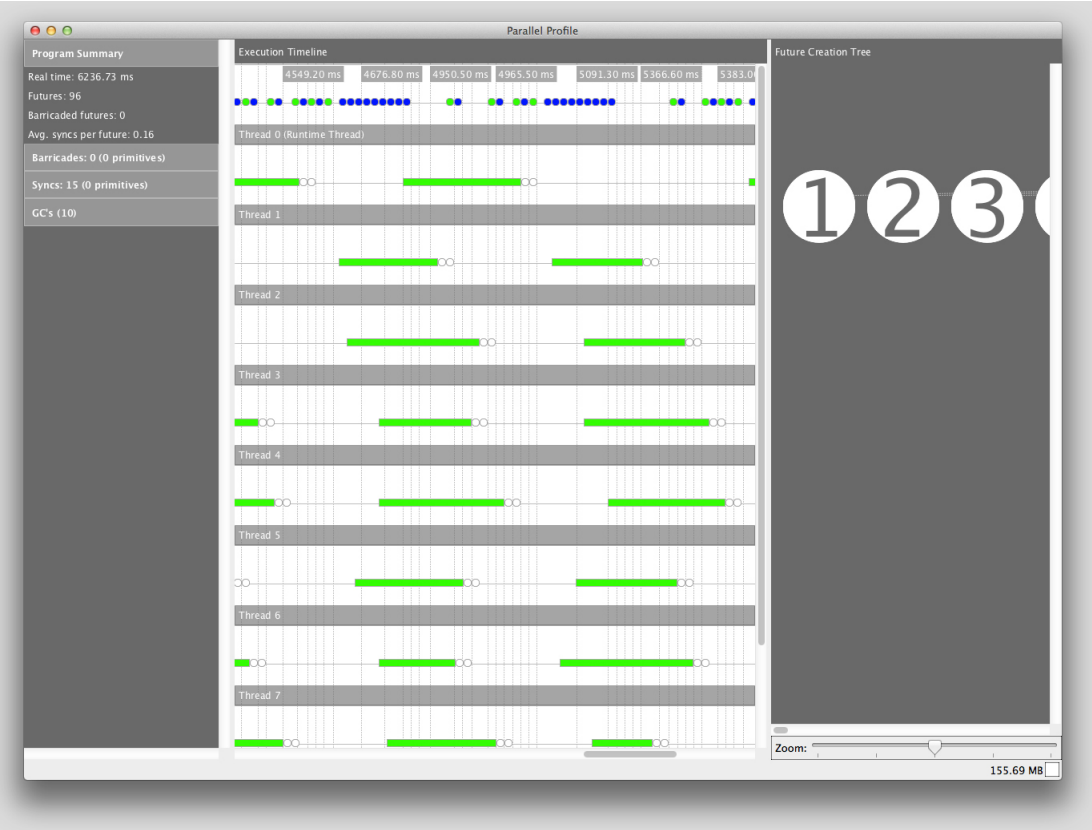

Figure 8: Profile for the parallel game of life

```
(for ([i (in-range 10)])
  (step au)))
```

This yields a window similar to that shown in figure 8. Notice that the program is free of barricades, indicated in the left-hand summary panel. Meanwhile, the execution timeline in the center panel shows our program in action: there are two vertical bands of green bars, with each individual green bar representing a future executing user code. The blue dots on the "runtime thread" line (on top) represent new future creations. Given what we know about our program—each step spawns `processor-count` futures, each with a fixed amount of work—we can infer that these two bands correspond to two `step` invocations. The vertical stacking means we have

lots of parallel work occurring, but even more importantly, each individual future is executing to completion with no interruptions/synchronizations.

CHAPTER 6

Parallel Ray Tracer

To demonstrate the value of the futures visualizer in spotting implementation-imposed performance problems—as well as Typed Racket's usefulness in avoiding such problems—we will add support for parallel computation to selected functions in Racket's `images` package. This package offers functions for generating ray-traced images and icons (some of which are used in the DrRacket IDE)—one such function is `plt-logo`, which produces the image shown in figure 9. Because ray tracing is computationally intensive and involves several independent passes over large matrices, it is a good candidate for futures.

Each matrix in Racket's ray tracer is represented as a `flomap` data structure, which is a container of *m* x *n* floating-point vectors of length *k*. They are conceptually similar to bitmaps, but with floats instead of whole numbers and with a user-specified number of components for each element. An object is represented by four such `flomap`s representing four properties: alpha, RGB, z (or height map), and surface normals.

The `plt-logo` function constructs two flomaps—one containing ARGB values and one containing a height map—and passes them to library code which splits the former into separate flomaps (alpha and RGB), and infers the normal flomap from the height map. Ray tracing is done in two separate passes: pass (1) traces light from a single directional light source, calculating both a *shadow map* (analogous to a photosensitve sheet of paper laying flat on a table beneath the rendered object, collecting light that either passes through or is reflected off the object) and a diffuse map; pass (2) traces light from objects in the scene to the viewer's eye. Timing analysis shows

Figure 9: Ray-traced PLT logo

that the core of the ray-tracing work can constitute roughly 85.0% of the overall running time of `plt-logo`, and pass (1) tends to do the lion's share of work, constituting as much as 67.0% of `plt-logo`'s running time. Thus, pass (1) is the primary focus of the parallelization effort. Note that, by Amdahl's Law, we cannot hope for ideal speedup: even if we assume ideal speedup for some parallelization of the ray-tracing core (both passes), 15.0% of the algorithm's running time remains serial. Maximum possible speedup as a function of processor count is showin in figure 10.

The ray-tracing library has two properties which make it ideal for parallelization using futures: (1) it is written in Typed Racket, allowing the compiler to replace calls to arithmetic functions such as `*` with their unsafe counterparts where appropriate; and (2) it is pervasively imperative, dominated by arithmetic computation and updates to `flvector` containers.

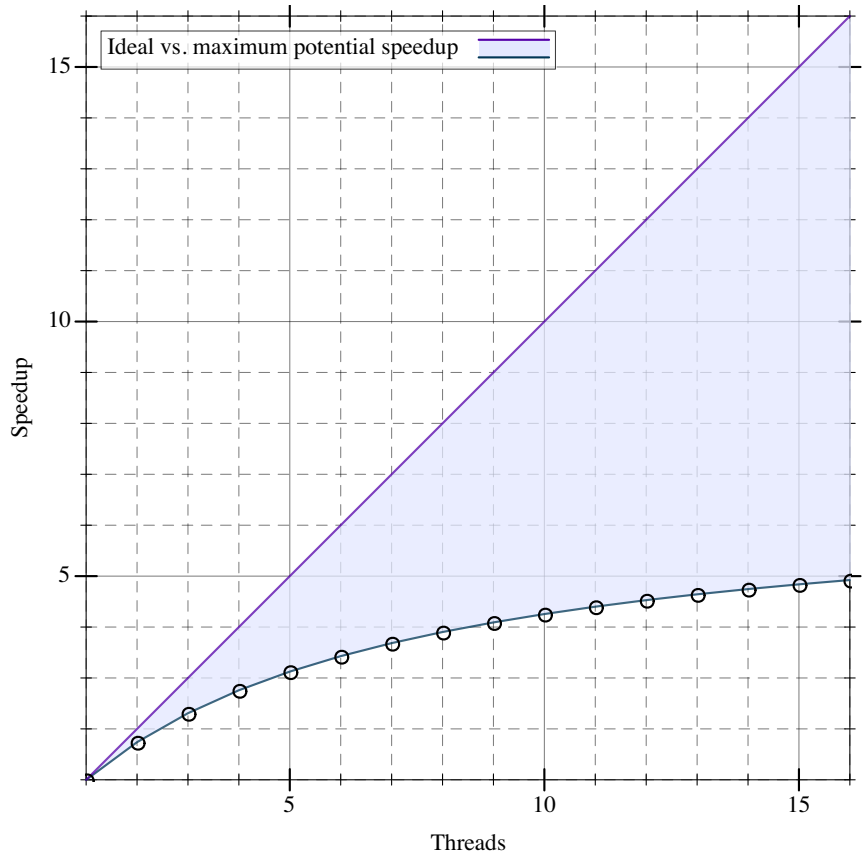

Figure 10: Possible theoretical plt-logo speedup with parallel ray-tracing core

6.1. Parallelizing plt-logo

Because the shadow map computation step depends on work done in the diffuse map step, these two tasks cannot be run in parallel; instead, the core loops for each one are parallelized using static scheduling. Each one consists of two nested loops, where the goal on each iteration is to compute some value for a particular "dot" in a flomap. Parallelizing these loops yielded no barricades,

but did show large amounts of allocation. The sources of this allocation, and the steps required to mitigate them, are detailed in the following sections. Note that, though the solutions to each problem were motivated by the need to reduce synchronizations in parallel programs, all of them ultimately yield better performance in sequential programs as well.

Issue #1: Right-hand conditional boxing

Consider the following:

```
#lang typed/racket

(λ (fs i)
  (define f (flvector-ref fs i))
  ...)
```

This expression might be compiled into the bytecode shown in figure 11. Here, the compiler has wrapped `flvector-ref` access with a bounds check, where its unsafe counterpart can be called if the index is within the vector's bounds. However, it is easy to see that in either case `local120` will be a flonum.

This problem is unavoidable without changes to the Racket compiler. Specifically, we improved the compiler such that it is able to detect such occurrences of `if` expressions where evaluation of either branch is guaranteed to return a flonum (i.e., a given branch is either a reference to a variable known to be a flonum or is an application of some function known by the compiler to always return a flonum).

Issue #2: Cross-module function calls returning multiple values

Figure 12 shows a program implemented in two modules, `A` and `B`. The `B` module contains a function returning multiple values, each of which is a flonum; `A` calls this function inside a loop body.

```
(let ([local120
       (if (if (unsafe-fx>= fixnum106 '0)
               (unsafe-fx<
                fixnum106
                (unsafe-flvector-length
                 arg6-98))
               '#f)
           (unsafe-flvector-ref arg6-98 fixnum106)
           (flvector-ref arg6-98 fixnum106))])
      ...)
```

Figure 11: Flonum definition with right-hand-side conditional

```
;; Module A
(require racket/future
         racket/flonum
         math/flonum
         "B.rkt")
(for ([i (in-range 0 (- SZ 6) 6)])
               (define-values (x y z)
                 (make-values (flvector-ref vec i)
                              (flvector-ref vec (+ i 1))
                              (flvector-ref vec (+ i 2))
                              (flvector-ref vec (+ i 3))
                              (flvector-ref vec (+ i 4))
                              (flvector-ref vec (+ i 5))))
               (flvector-set! vec i (+ x y z)))

;; Module B
(: make-values (Flonum Flonum Flonum
                       Flonum Flonum Flonum
                       -> (Values Flonum Flonum Flonum)))
(define (make-values a b c d e f)
  (values (* a b)
          (* c d)
          (* e f)))
```

Figure 12: Allocation in define-values with a right-hand-side function call

```
(let ((localv34 ?) (localv35 ?) (localv36 ?))
    (begin
      (set!-values (localv34 localv35 localv36)
        (_make-values
         (flvector-ref (#%checked _vec) arg1-31)
         (flvector-ref (#%checked _vec) (+ arg1-31 '1))
         (flvector-ref (#%checked _vec) (+ arg1-31 '2))
         (flvector-ref (#%checked _vec) (+ arg1-31 '3))
         (flvector-ref (#%checked _vec) (+ arg1-31 '4))
         (flvector-ref (#%checked _vec) (+ arg1-31 '5))))
      (flvector-set!
       (#%checked _vec)
       arg1-31
       (unsafe-fl+
        (unsafe-fl+ localv34 localv35)
        localv36))))
```

Figure 13: Bytecode for module A

```
(define-syntax (make-values stx)
  (syntax-case stx ()
    [(_ a b c d e f)
     #'(values (* a b)
               (* c d)
               (* e f))]))
```

Figure 14: Inlining via macro

The bytecode compiler will emit code for the loop body similar to that shown in figure 13. The localv34, localv35, and localv36 variables correspond to x, y, and z; however, the compiler has created boxes for each of them, which are then updated with the results of the call to make-values. We can see the problem in action by wrapping the code in a future and passing it to visualize-future, which yields the trace in figure 15.

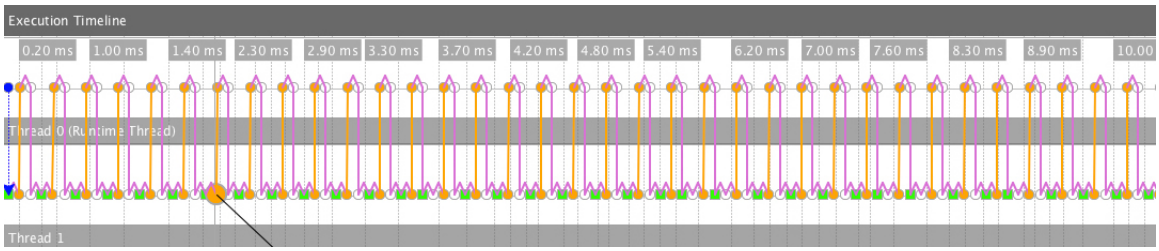

Figure 15: Future visualizer trace for flonum-allocating program

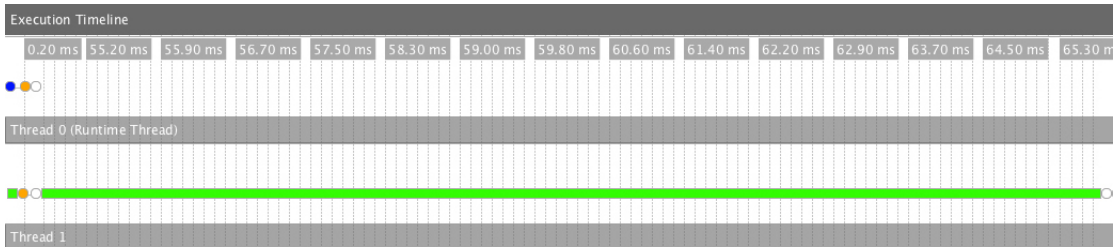

Figure 16: Future visualizer trace for program with flonum unboxing

```
(define (foo x y z out-vec)
  (for* ([i (in-range x)]
         [j (in-range y)])
    (vector-set! out-vec i (+ (* x i)
                              (* y j)
                              (* z i)))))
```

Figure 17: Nested loop with captures

Eliminating this problem altogether requires non-trivial improvements to the compiler, but it is easily sidestepped by rewriting `make-values` as a macro (effectively inlining it in the loop body in module `A`), as shown in figure 14. The futures visualizer shows a dramatic difference in figure 16.

```
(#%closed
  for-loop23
  ;arg0-106 = i, arg1-107 = x, arg2-108 = y
  ;arg3-109 = z, arg4-110 = out-vec, arg6-112 = j
  (lambda (arg0-106 arg1-107 arg2-108
           arg3-109 arg4-110 arg6-112)
      (...)
      (if (< arg6-112 arg2-108)
        (begin
          (vector-set!
           arg4-110
           arg0-106
           (+
            (* arg1-107 arg0-106)
            (* arg2-108 arg6-112)
            (* arg3-109 arg0-106)))
          (for-loop23
           arg0-106
           arg1-107
           arg2-108
           arg3-109
           arg4-110
           (+ arg6-112 '1)))
        (void))))
```

Figure 18: Loop closure bytecode

Issue #3: Closure argument-count limits

Consider figure 17, which defines a function with a nested loop (the `for*` form generates an n-level nested loop for n loop variables). Disregarding loop unrolling, the compiler will ultimately transform the inner loop into a recursive function similar to that of figure 18. Note that we get a set of arguments which roughly correspond to the program variables required by the loop body.

The ray tracer's diffuse-map computation in pass (1) resembles such a program, but with a much larger set of variables dependencies (32). For historical reasons, Racket's JIT compiler will

```
(define (foo/loop arg-vec out-vec)
  (for* ([i (in-range x)]
         [j (in-range y)])
    (define x (vector-ref arg-vec 0))
    (define y (vector-ref arg-vec 1))
    (define z (vector-ref arg-vec 2))
    (vector-set! out-vec i (+ (* x i)
                              (* y j)
                              (* z i)))))

(define (foo x y z out-vec)
  (foo/loop (vector x y z) out-vec))
```

Figure 19: Reducing loop-body closure arity with argument vectors

not perform flonum unboxing for closures accepting more than 25 arguments; thus we end up with hot-loop allocation which causes constant synchronization. Though the compiler could be modified to support flonum unboxing in this case, such situations are fairly rare, and it is much easier to work around the limitation: figure 19 lifts the loop out into a separate function taking a single vector argument, the contents of which are extracted inside the loop body.

Part 4

Implementation

CHAPTER 7

Futures

Since `future` does not promise to run a given thunk in parallel, a correct implementation of `future` and `touch` is easily added to any language implementation; the result of `future` can simply encapsulate the given thunk, and `touch` can call the thunk if no previous `touch` has called it. Of course, the trivial implementation offers no parallelism. At the opposite extreme, in an ideal language implementation, `future` would immediately fork a parallel task to execute the given thunk—giving a programmer maximal parallelism, but placing a large burden on the language implementation to run arbitrary code concurrently.

The `future` and `touch` constructs are designed to accommodate points in between these two extremes. The key is to specify when computations can proceed in parallel in a way that is (1) simple enough for programmers to reason about and rely on, and (2) flexible enough to accommodate implementation limitations. In particular, we are interested in starting with an implementation that was designed to support only sequential computation, and we would like to gradually improve its support for parallelism.

To add futures to a given language implementation, the language's set of operations is partitioned into three categories:

- A *safe* operation can proceed in parallel to any other computation without synchronization. For example, arithmetic operations are often safe. An ideal implementation categorizes nearly all operations as safe.

- An *unsafe* operation cannot be performed in parallel, either because it might break guarantees normally provided by the language, such as type safety, or because it depends on the evaluation context. Its execution must be deferred until a `touch` operation. Raising an exception, for example, is typically an unsafe operation. The simplest, most conservative implementation of futures categorizes all operations as unsafe, thus deferring all computation to `touch`.
- A *synchronized* operation cannot, in general, run in parallel to other tasks, but by synchronizing with other tasks, the operation can complete without requiring a `touch`. It thus allows later safe operations to proceed in parallel. Operations that allocate memory, for example, are often synchronized.

In a language like Racket, the key to a useful categorization is to detect and classify operations dynamically and at the level of an operator plus its arguments, as opposed to the operator alone. For example, addition might be safe when the arguments are two small integers whose sum is another small integer, since small integers are represented in Racket as immediates that require no allocation. Adding an integer to a string is unsafe, because it signals an error and the corresponding exception handling depends on the context of the `touch`. Adding two flonums, meanwhile, is a synchronized operation if space must be allocated to box the result; the allocation will surely succeed, but it may require a lock in the allocator or a pause for garbage collection.

This partitioning strategy works in practice because it builds on an implicit agreement that exists already between a programmer and a language implementation. Programmers expect certain operations to be fast, while others are understood to be slow. For example, programmers expect small-integer arithmetic and array accesses to be fast, while arbitrary-precision arithmetic or dictionary extension are relatively slow. From one perspective, implementations often satisfy such expectations though “fast paths” in the interpreter loop or compiled code for operations that are

expected to be fast, while other operations can be handled through a slower, more generic implementation. From another perspective, programmers learn from experimentation that certain operations are fast, and those operations turn out to be fast because their underlying implementations in the runtime have been tuned to follow special, highly optimized paths.

The key insight behind this work is that fast paths in a language implementation tend to be safe to run in parallel and that it is not difficult to barricade slow paths, preventing them from running in parallel. An implementation's existing internal partitioning into fast and slow paths therefore provides a natural first cut for distinguishing safe and unsafe operations. The implementation strategy is to set up a channel from `future` to the language implementation's fast path to execute a future in parallel. If the future's code path departs from the fast path, then the departing operation is considered unsafe, and the computation is suspended until it can be completed by `touch`.

The details of applying the technique depend on the language implementation. Based on experience converting two implementations and knowledge of other implementations, certain details may be expected to be common among many implementations. For example, access to parallelism normally builds on a POSIX-like thread API. Introducing new threads of execution in a language implementation may require that static variables within the implementation are converted to thread-local variables. The memory manager may need adjustment to work with multiple threads; as a first cut, all allocation can be treated as an unsafe slow path. To support garbage collection and similar global operations, the language implementation's fast path needs hooks where the computation can be paused or even shifted to the slow path.

Figure 20 illustrates the general methodology. The process begins with the addition of low-level support for parallelism, then experimenting with the paths in the implementation that are affected. Based on that exploration, one can derive a partitioning of the language's operations

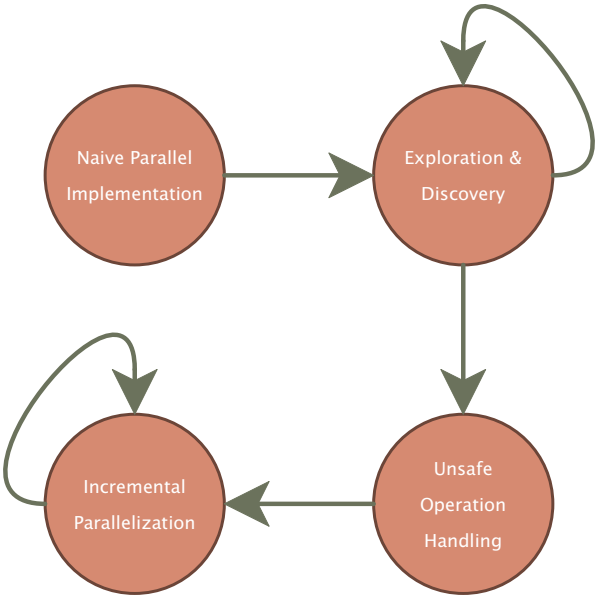

Figure 20: Incremental parallelization methodology

into safe and unsafe. Having partitioned the operations, we must implement a trapping mechanism capable of suspending a parallel task before it executes an unsafe operation. Finally, we refine the partitioning of the operations (perhaps designating and implementing some operations as synchronized), *incrementally*, as guided by the needs of applications.

7.1. Futures in Racket

The Racket runtime system is implemented by roughly 100k lines of C code. It includes a garbage collector, macro expander, bytecode compiler, bytecode interpreter, just-in-time (JIT) compiler, and core libraries. The core libraries include support for threads that run concurrently

at the Racket level, but internally threads are implemented as co-routines (i.e., they are "user threads").

Execution of a program in the virtual machine uses a stack to manage the current continuation and local bindings. Other execution state, such as exception handlers and dynamic bindings, are stored in global variables within the virtual-machine implementation. Global data also includes the symbol table, caches for macro expansion, a registry of JIT-generated code, and the garbage collector's metadata. In addition, some primitive objects, such as those representing I/O streams, have complex internal state that must be managed carefully when the object is shared among concurrent computations.

The virtual machine's global and shared-object states present the main obstacles to parallelism for Racket programs. An early attempt to implement threads as OS-level threads—which would provide access to multiple processors and cores as managed by the operating system—failed due to the difficulty of installing and correctly managing locks within the interpreter loop and core libraries. Since that early attempt, the implementation of Racket has grown even more complex.

A related challenge is that Racket offers first-class continuations, which allow the current execution state to be captured and later restored, perhaps in a different thread of execution. The tangling of the C stack with execution state means that moving a continuation from one OS-level thread to another would require extensive changes to representation of control in the virtual machine.

The design of futures side-steps the latter problem by designating operations that inspect or capture the current execution state as unsafe; thus, they must wait until a `touch`. Meanwhile, the notions of unsafe and synchronized operations correspond to using a single "big lock" to protect other global state in the virtual machine.

The following sections provide details regarding both the adjustment of the implementation of execution state and the operation-partitioning in Racket.

Compilation, Execution, and Safety Categorization

Execution of a Racket program uses two phases of compilation. First, the bytecode compiler performs the usual optimizations for functional languages, including constant and variable propagation, constant folding, inlining, loop unrolling, closure conversion, and flonum unboxing. The bytecode compiler is typically used ahead of time for large programs, but it is fast enough for interactive use. Second, when a function in bytecode form is called, the JIT compiler converts the function into machine code. The JIT creates inline code for simple operations, including type tests, arithmetic on small integers or flonums, allocations of cons cells, array accesses and updates, and structure-field operations. When the JIT compiler is disabled, bytecode is interpreted directly.

The first step in supporting useful parallelism within Racket was to make the execution-state variables thread-local at the level of OS threads, so that futures can be executed speculatively in new OS-level threads. To simplify this problem, attention was confined to the execution state that is used by JIT-generated code. Consequently, the first cut at categorizing operations was to define as safe any operation that is implemented directly in JIT-generated code (i.e. any operation that can be translated by the JIT compiler into machine instructions which do not include function calls back into runtime code), and any other operation was unsafe. This first-cut strategy offered a convenient starting point for the incremental process, in which performance-critical unsafe operations are modified to make them future-safe.

When a future is created in Racket, the corresponding thunk is JIT-compiled and then added to a queue of ready futures. The queue is served by a pool of OS-level future threads, each of which begins execution of the JIT-generated code. At points where execution would exit JIT-generated

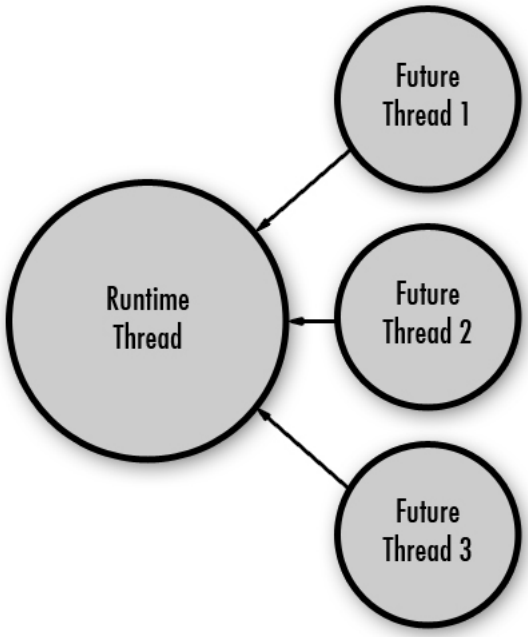

Figure 21: Parallel threads in Racket.

code, a future suspends to wait on the result of an unsafe operation. The operation is eventually performed by the original runtime thread when it executes a `touch` for the future. In the current implementation, a future remains blocked as long as it waits for the runtime thread to complete an unsafe operation; however, a future is not bound to any specific OS thread for the duration of its work. When a future needs to synchronize, it can be suspended and the OS thread executing it freed to execute other ones.

Using Thread-Local Variables

Since the JIT compiler was designed to work for a non-parallelized runtime system, the code that it generates uses several global variables to manage execution state. In some cases, state is kept primarily in a register and occasionally synchronized with a global variable. Changing the relevant

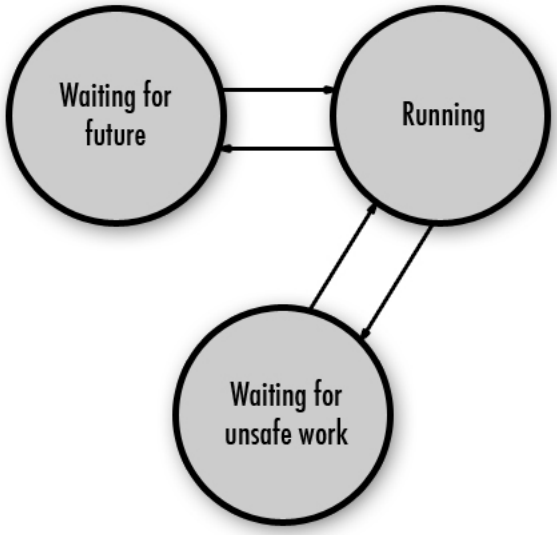

Figure 22: The lifetime of a parallel future thread.

```
typedef (*primitive)(int argc, Scheme_Object **argv);

Scheme_Object *handler(int argc, Scheme_Object **argv, primitive func) {
  Scheme_Object *retval;
  if (pthread_self() != g_runtime_thread_id) {
    /* Wait for the runtime thread */
    retval = do_runtimecall(func, argc, argv);
    return retval;
  } else {
    /* Do the work directly */
    retval = func(argc, argv);
    return retval;
  }
}
```

Figure 23: Typical primitive trap handler

global variables to be thread-local variables in the C source of the Racket implementation allowed

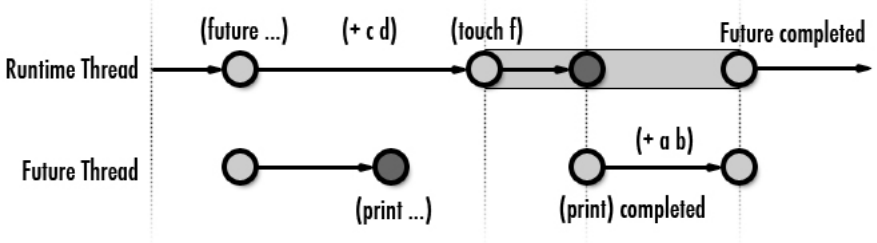

Figure 24: Timeline for a future with an unsafe operation

multiple JIT-generated code blocks to execute in parallel.[1] To make this work, the JIT compiler was adjusted to access a global variable through an thread-specific indirection. The thread-specific indirection is supplied on entry to JIT-generated code.

Handling Unsafe Operations

When JIT-generated code invokes an operation that is not implemented inline, it invokes one of a handful of C functions that call back into the general interpreter loop. When a future thread takes this path out of JIT-generated code, the call is redirected to send the call back to the runtime thread and wait for a response. Figure 23 illustrates the general form of such functions. Each first checks whether it is already executing on the runtime thread. If so, it performs the external call as usual. If not, the work is sent back to the runtime thread via `do_runtimecall`.

The runtime thread does not execute the indirect call until `touch` is called on the corresponding future. Figure 24 illustrates the way that an unsafe operation suspends a future until its value can be computed by the runtime thread in response to a `touch`. Note that the `touch` function itself is considered unsafe, so if `touch` is called in a future thread, then it is sent back to the runtime thread. Thus, the `touch` function need only work in the runtime thread.

[1]On some platforms, we could simply annotate the variable declaration in C with `thread`. On other platforms, we use pre-processor macros and inline assembly to achieve similar results.

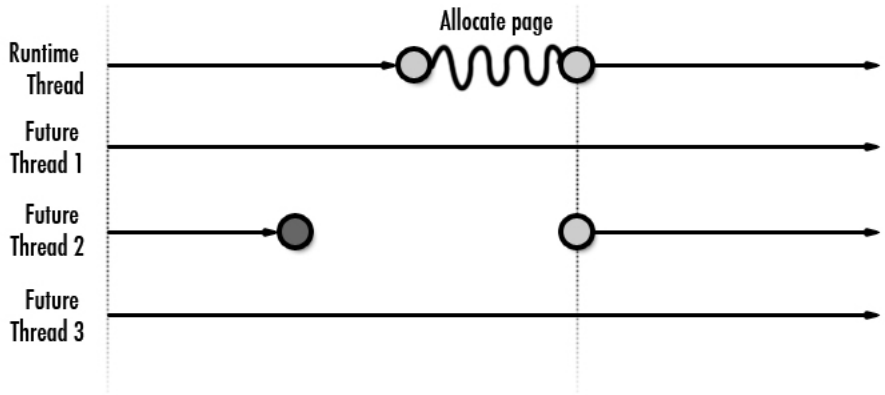

Figure 25: Timeline for a synchronized operation

Synchronized Operations

Like unsafe operations, synchronized operations always run on the runtime thread. Unlike unsafe operations, however, the runtime thread can perform a synchronized operation on a future thread's behalf at any time, instead of forcing the future thread to wait until `touch` is called.

As part of its normal scheduling work to run non-parallel threads, the runtime system checks whether any future thread is waiting on a synchronized operation. If so, it immediately performs the synchronized operation and returns the result; all synchronized operations are short enough to be performed by the scheduler without interfering with thread scheduling.

Currently, the only synchronized operations are allocation and JIT compilation of a procedure that has not been called before. More precisely, allocation of small objects usually can be performed in parallel (as described in the next section), but allocation of large objects or allocation of a fresh page for small objects requires cooperation and synchronized with the memory manager. Figure 25 illustrates the synchronized allocation of a new page with the help of the runtime thread.

Memory Management

Racket uses a custom garbage collector that, like the rest of the system, was written for sequential computation. Specifically, allocation updates some global state and collection stops the world. As in many runtime systems, the virtual machine and its garbage collector cooperate in many small ways that make inserting a third-party concurrent garbage collector prohibitively difficult. Similarly, converting the garbage collector to support general concurrency would be difficult. Fortunately, adapting the collector to support a small amount of concurrency is relatively easy.

The garbage collector uses a nursery for new, small objects, and then compacting collection for older objects. The nursery enables inline allocation in JIT-generated code by bumping a pointer. That is, a memory allocation request takes one of the following paths:

- *Fast Path* — the current nursery page has enough space to accommodate the current request. In this case, a page pointer is incremented by the size of the object being allocated, and the original page pointer is returned to the caller. This path is executed purely in line (a small number of instructions with no callbacks into unsafe C functions).
- *Slow Path* — the current page being used by the allocator does not have enough space to accommodate the current request. In this case, a new page must either be fetched from either the virtual machine's own internal page cache, or must be requested from the operating system. If the entire heap space has been exhausted, a garbage collection is triggered.

The nursery itself is implemented as a collection of pages, so adding additional thread-specific pages was straightforward. As long as it is working on its own page, a future thread can safely execute the inline-allocation code generated by the JIT compiler. Figure 26 illustrates the use of future-specific nurseries.

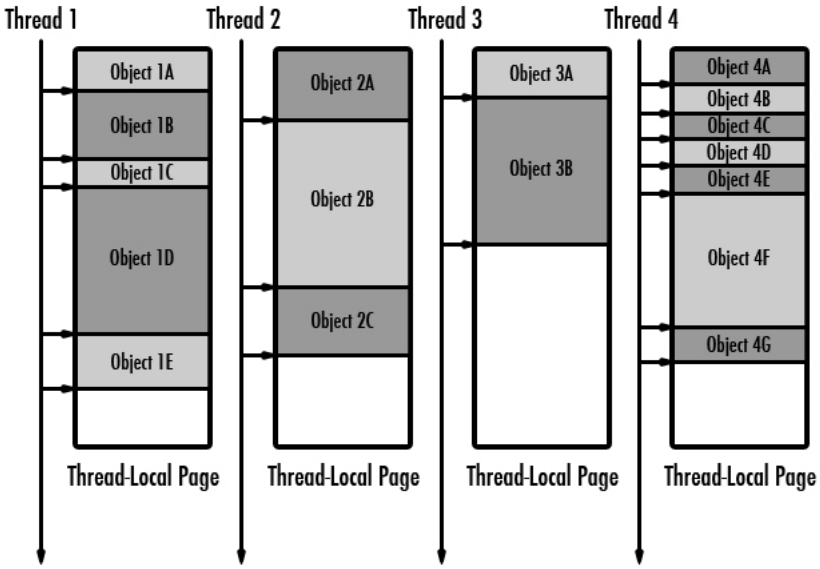

Figure 26: Per-future allocation

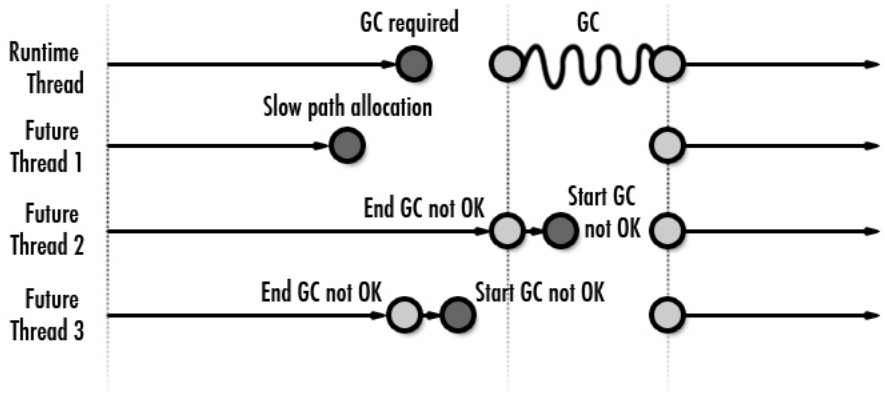

Figure 27: Rendezvous for garbage collection

Acquiring a fresh nursery page, in contrast, requires synchronization with the runtime thread, as described in the previous section. The size of the nursery page adapts to the amount of allocation that is performed by the future requesting the page.

Garbage collection still requires stopping the world, which includes all future threads. The JIT compiler generates code that includes safe points to swap Racket-level threads. When safe code is running in a future, it never needs to stop for other Racket threads, but the same safe points can be repurposed as garbage-collection safe points. That is, the inlined check for whether the computation should swap threads is instead used as a check for whether the future thread should pause for garbage collection. Meanwhile, garbage collection in the runtime thread must not be allowed unless all future threads are blocked at a safe point. Figure 27 illustrates the synchronization required for a garbage collection.

Besides providing support for thread-specific nursery pages, the garbage collector requires minor adjustments to support multiple active execution contexts to be treated as roots. Roughly, the implementation uses fake Racket threads that point to the execution state of a computation in a future thread.

Implementing `touch`

To tie all of the preceding pieces together, the implementation of `touch` is as follows:

- If the future has produced a result already, return it.
- If a previous `touch` of the future aborted (e.g., because the future computation raised an exception), then raise an exception.
- If the future has not started running in a future thread, remove it from the queue of ready futures and run it directly, recording the result (or the fact that it aborts, if it does so).
- If the future is running in a future thread, wait until it either completes or encounters an unsafe operation:
  - If the future thread has encountered an unsafe operation, perform the unsafe operation, return the result, and wait again. If performing the unsafe operation results in an

exception or other control escape, tell the future thread to abort and record the abort for the future.

- If the future completes in a future thread, record and return the result.

In addition, the scheduler loop must poll future threads to see if any are waiting on synchronized operations; if so, the operation can be performed and the result immediately returned to the future thread. By definition, a synchronized operation cannot raise an exception.

7.2. Futures in Parrot

Parrot is a register-based virtual machine with heap-allocated continuation frames. Compilers target Parrot by emitting programs in the Parrot intermediate language, which is a low-level, imperative assembly-like programming language, but with a few higher-level features, including garbage collection, subroutines, dynamic container types, and a extensible calling convention.

Three key characteristics made adding futures to the Parrot VM machine relatively easy:

- an existing abstraction and wrapping of OS-level threads;
- a concurrency or green thread implementation that abstracts and encapsulates thread of execution state; and
- a pluggable *runloop* (i.e., interpreter loop) construct that allows switching between different interpreter cores.

With such groundwork in place, the following enhancements to the Parrot C implementation were needed:

- refactoring of representation of threads to allow them to be reused for futures,
- allowing an unfinished future computation to be completed by the main interpreter after an OS-level join, and

- creating a new runloop that executes only safe (for parallelism) operations and reverts back to the main thread for unsafe operations.

In Parrot, spawning a `future` consists of creating a new interpreter and specifying what data, if any, to share between the parent and child OS-level threads. Parrot futures have their own execution stack, but they share the same heap and bytecodes. To implement `touch`, Parrot waits for the future thread to return and then checks to see if the future returned a value, in which case the value is returned. If the future encountered an unsafe instruction, the future thread returns a computation, which is completed in the caller.

Parrot's runloop is the core of the interpreter, where bytecodes are fetched and executed. Parrot has several different runloops that provide debugging, execution tracing, profiling, and experimental dispatching. Parallel futures adds a future runloop that checks each bytecode just before it is executed to see if it is safe to execute or if the future needs to be suspended and executed sequentially in the main interpreter. This runtime safety checking includes argument type checking and bounds checking on container data structures.

Parrot is a highly dynamic virtual machine. Many bytecodes are actually virtual function calls on objects that can be user-defined; for example, the array `get` and `set` opcodes may be translated into virtual method calls on a wide variety of container object types. Some container object types have fixed size at construction time, while others grow dynamically. This indirection inherent in the bytecode makes compile-time safety checking difficult. To deal with this problem, the future runloop checks at run time that an operand container object is of the fixed-size variety (and thus safe to run in parallel), not a dynamically growing variant.

Because it is mostly a proof-of-concept, Parrot futures implementation does not include special handling for synchronized operations or future-local allocation, and the future runloop treats most opcodes as unsafe operations. Arithmetic, jumps, and fixed-container accesses and updates are

the only operations designated as safe—enough to run a parallel benchmark which scales with multiple processors. Experience working with both the Parrot and Racket runtime systems suggest that the work required to add those features to the Parrot would be similar in nature to that of the Racket system.

CHAPTER 8

Future Visualizer

Racket's future visualizer, introduced in part III, provides feedback regarding CPU utilization, allocation frequency, and barricading penalties incurred by both `future` and `place` programs. It is essentially a trace-reconstruction tool, taking as input a collection of program log messages extracted via Racket's logging system (as shown briefly in chapter 4) and producing a visualization of those messages.

8.1. Profiling Futures

Whenever a future is started, blocked, etc., the runtime system logs an event that records the future's identity, the OS-level thread in which the event took place, a symbol representing the future-related action, the wall-clock time at which the action occurred, and optionally the name of a Racket-level procedure to help correlate the event with the program source. The visualizer can then reconstruct a trace of the computation from information in the logged events.

Since logging support is always enabled in a Racket build, the visualizer requires no additional low-level hooks into the runtime system. Like other logging systems, Racket's logging system keeps track of event consumers, so that log-entry producers can detect whether events are worth reporting and avoid the overhead of logging information that would be ignored. To track consumers, the log is not an object that can be read directly; instead, logged events are received through *log receiver* objects that include a particular level, such as "error," "warning," or "debug."

The current effective logging level (the highest level at which a receiver is active) can be queried and compared against the level of a potential log event.

The log-level test is cheap enough for the runtime thread to use at any time, but it involves enough objects and caches that it would not be safe within a future. Instead of using the log directly, each future thread maintains its own queue of events, which the runtime thread periodically converts into regular log events (if there is any relevant receiver). A future thread's log queue is a fixed-size array that contains only atomic values, which minimizes its locking and memory-management requirements. Furthermore, entries are added to the log queue only for events that require some other synchronization, such as changing the state of a future, so log-queue locking piggy-backs on existing locks. Since the log queue has a fixed size, it can overflow, in which case a "queue overflow" event replaces the most recent event; log overflow is rare, and the explicit event ensures that the visualizer can fall back to reporting approximate information if an overflow occurs.

Despite efforts to minimize the cost of logging, when many events are generated and consumed by a receivers, overhead is unavoidable. Such a setting, however, corresponds to a slow program whose performance is being analyzed, so the overhead remains small relative to the computation and worthwhile to the user.

8.2. Profiling Places

Racket is also multi-paradigm in its approach to parallelism; while futures offer lightweight, task-based, shared-memory parallelism, *places* [39, 40] offer a complementary approach with a different set of restrictions. In Racket, places are separate instances of the runtime system, each with their own memory space and garbage collector. They may communicate via *message passing*, in which values are sent across channels and copied from one place to another.

Because each place retains its own internal runtime structures, the dangers to these structures inherent in a shared-memory setting do not exist; thus they offer unrestricted parallelism in the sense that no slow-path barricades exist. However, a place is more expensive to create than a future, and the set of communicable objects is limited to a simple class of mutable values or immutable ones.

Because futures and places may be used in conjunction with one another in the same program, the futures visualizer supports profiling of both per-place future traces and places creation/communication patterns.

8.3. Trace Analysis

The futures visualizer expects input in the form of an S-expression, where atoms are log messages which may be any of the following structure types: `future-event`, `gc-info`, or `place-event`. Though the implementation of the logging mechanism generally means the structure of this S-expression will mimic the topology of the places network used in the program (with futures-only programs generating a single flat list), the visualizer does not rely on this structure to reconstruct the trace, because places log events recording every place creation and communication. Thus the visualizer effectively flattens the input tree before building the trace history.

For each `place` in the program, the analysis constructs a `trace` structure, which contains an entry for each OS-level thread in the system. These entries contain lists (sorted chronologically) of future-related events, so visual representation of thread timelines is straightforward. In addition, each future event is wrapped by an `event-info` structure which also contains forward and back pointers to other events which occurred on the same future as its own.

Part 5

Evaluation

CHAPTER 9

Developer Effort

Because the slow-path barricading approach was developed in the context of the Racket runtime system, the parallel Racket implementation is the most mature. Given that, one might expect to see substantial development effort; however, this is not the case. Figure 28 lists the overall development costs (in person-hours) required to apply the approach to Racket. Costs are partitioned

	Person Hours	
Task	expert	non-expert
General Steps		
Naive implementation	6	40
Exploration and discovery	-	480
Unsafe-operation handling	6	16
Blocked-future logging	1	-
Total General:	13	536
Implementation-specific Steps		
Thread-local variables	8	-
Future-local allocation	8	-
Garbage-collection sync	6	-
Thread-local performance	6	-
Total Specific:	28	-
Overall Total:	41	536

Figure 28: Racket implementation effort by task. Adding a release-quality futures implementation to Racket using our approach required only one week of expert time, and one academic quarter of non-expert time.

	Person Hours
Task	expert
General Steps	
Naive implementation	8
Exploration and discovery	24
Unsafe-operation handling	8
Total General:	40
Implementation-specific Steps	
Wrapping OS-level threads	8
Future runloop	4
Total Specific:	12
Overall Total:	52

Figure 29: Parrot implementation effort by task. Adding a proof-of-concept implementation of futures to Parrot using our approach required only about a week of expert time.

into two categories: general and implementation-specific. The general category reflects the first three steps described in figure 20; the implementation-specific category reflects the incremental parallelization step, which includes work that was necessary for Racket, but may not apply in other runtime adaptation work.

The columns in figure 28 show the efforts of two developers, the expert being the designer and primary implementer of the Racket runtime and the non-expert being a first-year graduate student (working in a different city). Notice that with roughly one week of expert time, and one academic quarter of non-expert time, it was possible to apply the approach to a widely used,[1] mature[2] runtime system and achieve reasonable performance results. Furthermore, the effort produced a parallel futures implementation that has been part of the main-line release of Racket for several years.

Having gained experience with our approach, we also applied it to the Parrot VM to ensure our experience is not Racket-specific. Figure 29 lists the development time we required to add a first-cut futures implementation. The Parrot effort was undertaken by a third-year Ph.D. student who,

[1]The Racket distribution is downloaded more than 300 times per day.
[2]The runtime system has been in continuous development since 1995.

while not a core developer of Parrot, is intimately familiar with its internals (and was also familiar with the Racket effort). The upshot of figure 29 is that adding a proof-of-concept implementation of futures to Parrot using SPB required only about a week of expert time.

That the same approach has been applied successfully and efficiently to two very different runtime systems suggests that it is quite general.

CHAPTER 10

Performance

10.1. Testbed, Metrics, and Benchmarks

Performanc of the two futures implementations was evaluated using two different machines and five benchmarks. The commonplace parallel systems performance metrics of strong scalability and raw performance are used here, and results are compared against the same algorithms implemented in other languages.

Machines

Performance evaluations were conducted on both a high-end desktop workstation with two quad-core processors (8 cores), and a mid-range server machine with four quad-core processors (16 cores). The detailed configuration of these machines is given in figure 30. During the execution of a given benchmark, no other significant load was placed on the machine. I verified that separate threads of execution used by the runtime system were in fact mapped to separate processing cores.

Metrics

The number of threads used by each of the benchmarks is a runtime parameter. I measured the wall-clock execution time of each benchmark as a function of this parameter, and present both the raw numbers and a speedup curve. The speedup curve shows the wall-clock time of the parallel implementation using a single thread divided by the wall-clock time of the parallel implementation

	Penghu	Cosmos
OS	OS X 10.6.2	CentOS 5.4
Processor Type	Intel Xeon	AMD Opteron 8350
Processors	2	4
Total Cores	8	16
Clock Speed	2.8 GHz	2.0 GHz
L2 Cache	12 MB	4x512 KB
Memory	8 GB	16 GB
Bus Speed	1.6 GHz	1 GHz
Racket	5.0 (32-bit)	5.0 (64-bit)
GCC	4.2.1 (Apple)	4.1.2 (Red Hat)
Java	Java SE 1.6	OpenJDK 1.6

Figure 30: Machine configurations used for benchmarks

using the indicated number of threads. The problem size remains constant as the number of threads increases; thus the speedup curve measures "strong scaling."

For several benchmarks, I also measured the wall-clock time of sequential implementations in various languages, including optimized C and Java.

Benchmarks

Figure 31 lists the benchmarks used to evaluate the performance of parallel futures in the Racket and Parrot VM implementations. Several of the evaluation programs are not drawn from a particular benchmark suite; rather, they are common implementations of well-known algorithms. Note that not all programs have a Parrot VM implementation.

Program	Implementation
Microbenchmarks (self-developed)	
MV-Sparse	Racket
Mergesort	Racket
Signal Convolution	Racket, Parrot
NAS Parallel Benchmarks [4]	
Integer Sort	Racket, Java
Fourier Transform	Racket, Java

Figure 31: Benchmarks, sources, and parallel implementations. Sequential implementations in C, Java, Racket, and Parrot are also used for comparison.

Signal convolution is a signal-processing algorithm used to determine the output signal of a system given its *impulse response* or *kernel*, which defines how the system will respond given an impulse function applied to the input signal. For any input signal x, we can compute each value in the corresponding output signal y using the following equation:

$$y^n = \sum\{k \,..\, -k\}\; x^k * h^{n-k}$$

where k is time and h is the impulse response/kernel. My implementation computes an output signal given a one-dimensional input signal and kernel, both of which are made up of floating-point values.

Signal convolution was implemented in sequential Racket, Racket with futures, sequential Parrot, Parrot with futures, and sequential C.

Mergesort sorts a vector of floating-point numbers. Two variants of the mergesort algorithm are considered: one that is readily parallelizable, and one that is not, but runs significantly faster than the parallel version on one processor [27]. I implemented both of the algorithms in sequential Racket, Racket with futures, and sequential C.

MV-Sparse does sparse matrix-vector multiplication using the compressed row format to store the matrix. For those unfamiliar with compressed row format, the essential idea is to flatten the

matrix into a single 1D array and combine it with parallel 1D arrays indicating both where rows begin and what the column indices are. I implemented MV-Sparse in sequential Racket, Racket with futures, and sequential C. The Racket version employs the higher-level nested data parallel primitive called a *gather*, which we have implemented using futures.

The NAS Parallel Benchmarks [4] are a suite of benchmarks derived from computational fluid dynamics applications of interest to NASA. They are widely used in the parallel systems community as application benchmarks. Only two of them are considered here.

NAS Integer Sort (IS) sorts an array of integer keys where the range of key values is known at compile-time. Sorting is performed using the histogram-sort variant of the bucket sort algorithm. NAS IS was implemented both in sequential and parallel Racket. The performance of those two implementations is compared against the publicly available sequential and parallel Java reference implementations [21].

NAS Fourier Transform (FT) is the computational kernel of a 3-dimensional Fast Fourier Transform. Each iteration performs three sets of one-dimensional FFT's (one per dimension). As with Integer Sort, NAS FT was implemented in both sequential and parallel Racket. I again compare against the publicly available sequential and parallel Java reference implementations.

10.2. Performance is Reasonable

Using futures implemented via the SPB approach to incrementally parallelizing existing sequential runtime systems, it is possible to achieve both reasonable raw performance and good scaling for the benchmarks we tested.

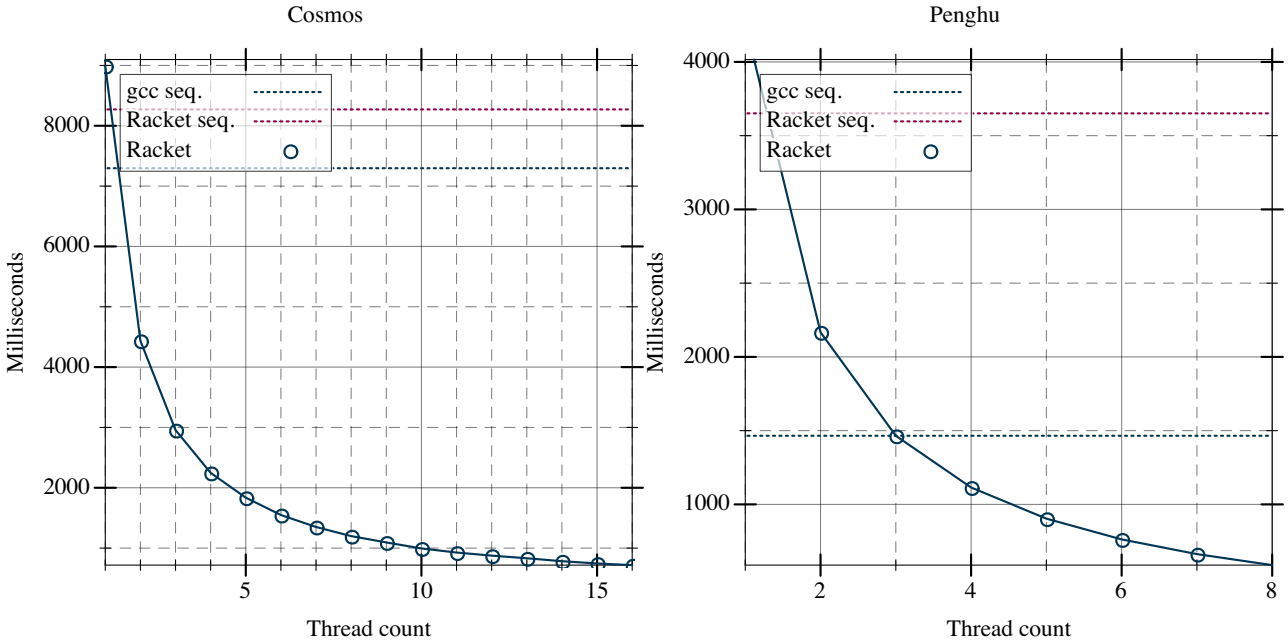

Figure 32: Signal convolution wall-clock times

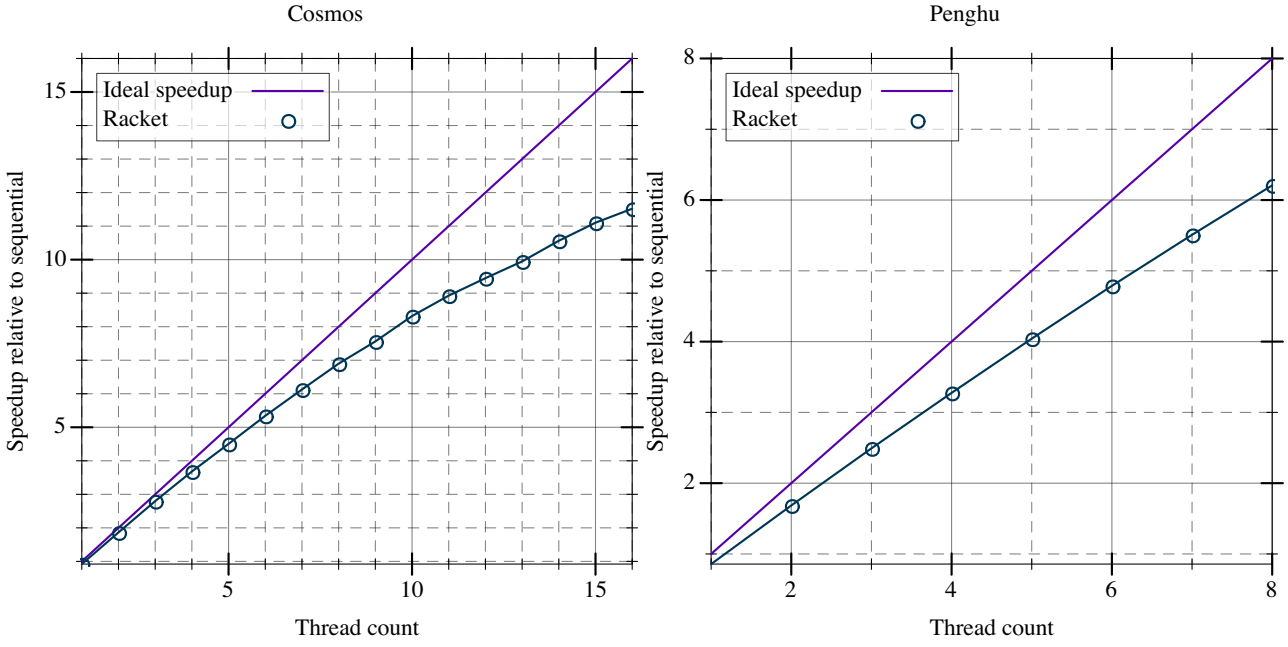

Figure 33: Signal convolution speedups

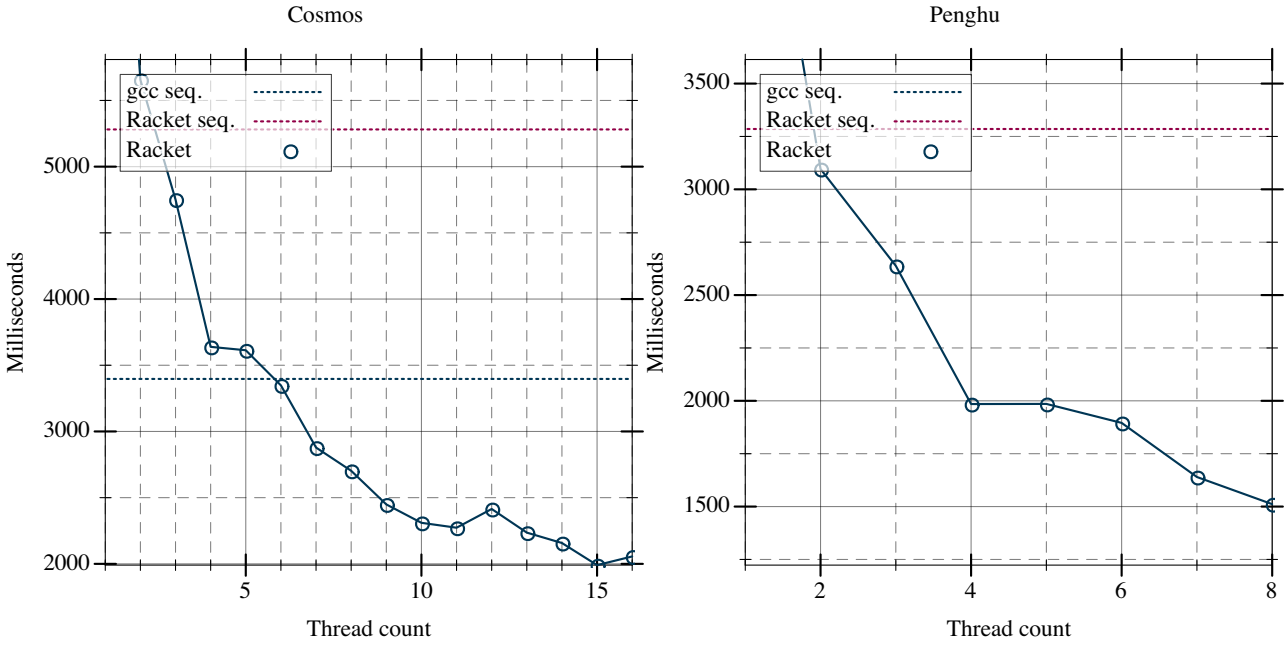

Figure 34: Mergesort wall-clock times

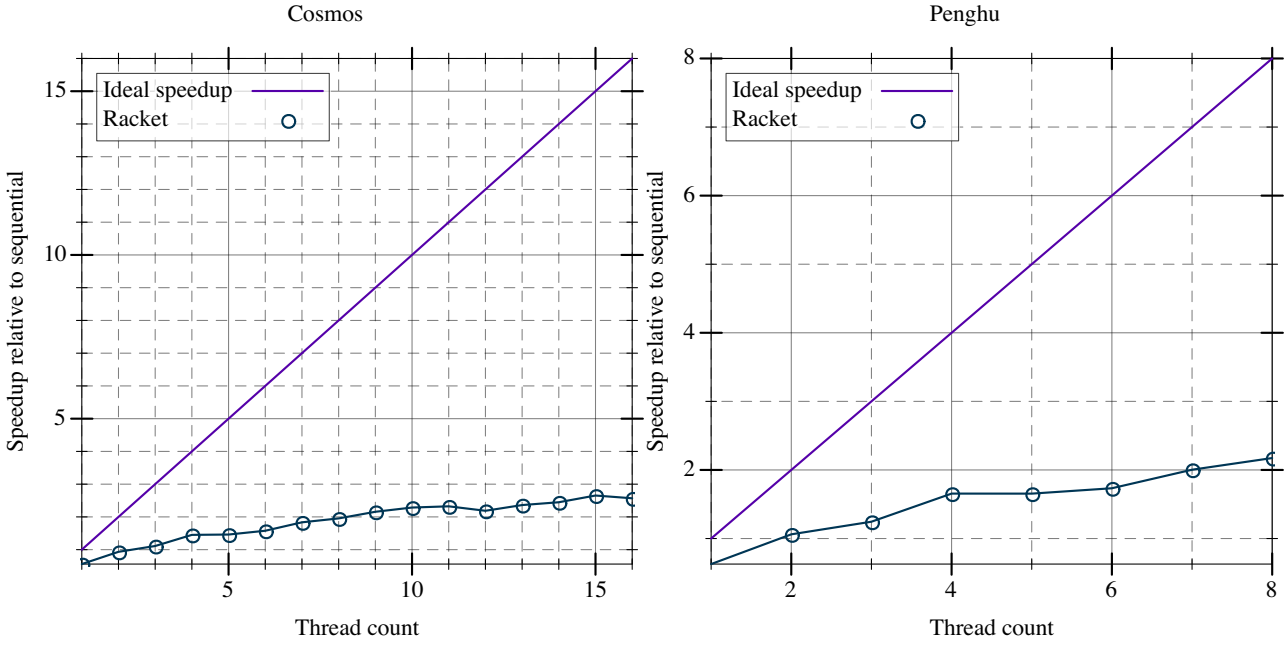

Figure 35: Mergesort speedups

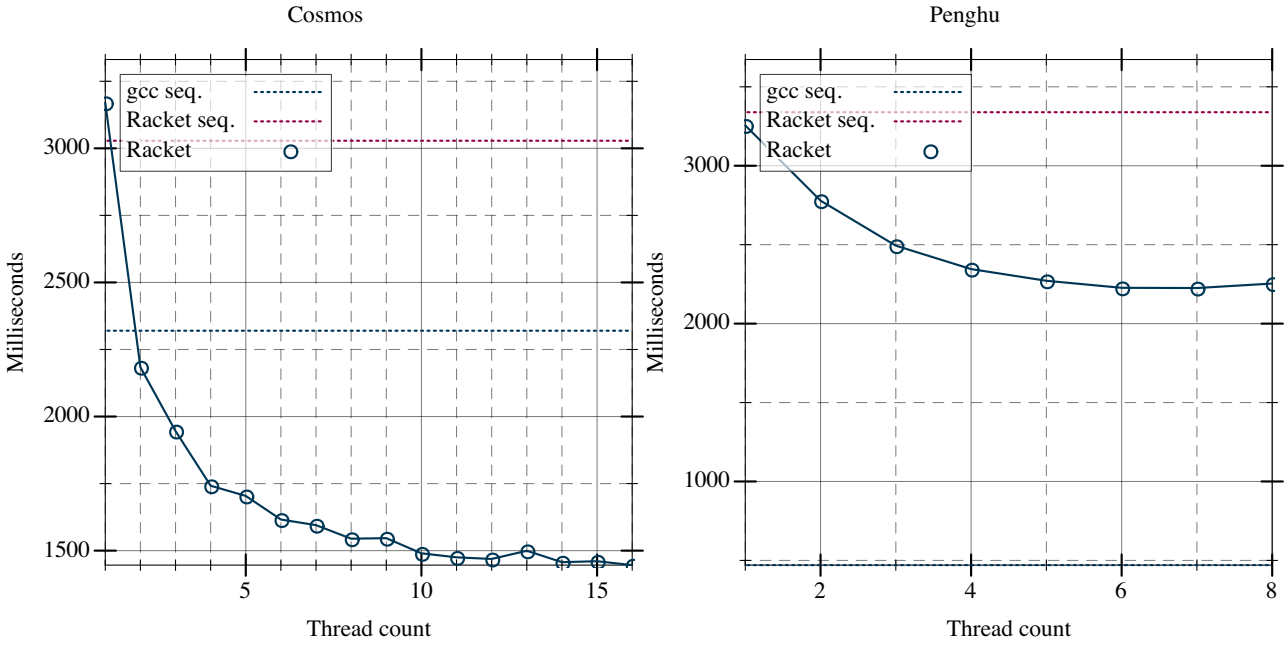

Figure 36: Sparse matrix-vector multiplication wall-clock times

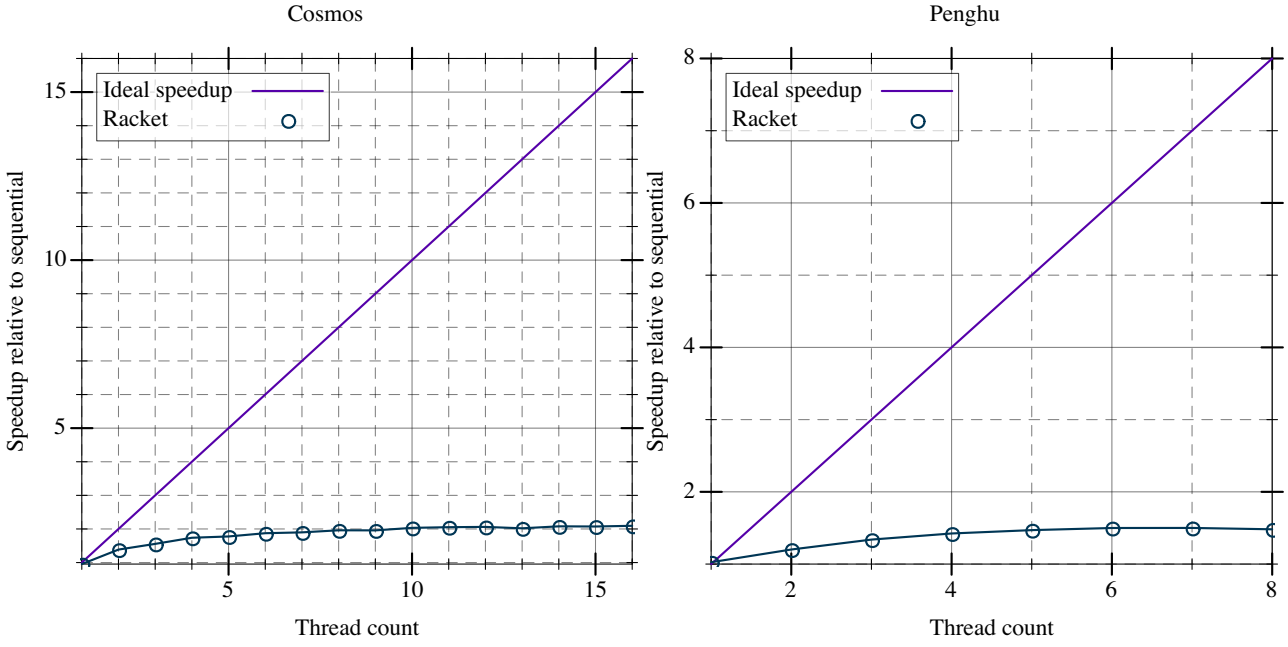

Figure 37: Sparse matrix-vector multiplication speedups

Racket

Figures 32, 34, and 36 show running time for the Racket implementations of the three microbenchmarks listed in figure 31; figures 33, 35, and 37 show their corresponding speedup curves. The results confirm that using futures, implemented using the developer-efficient incremental parallelization approach, it is feasible to achieve reasonable parallel performance on commodity desktops and servers, both in terms of raw performance and speedup.

Though the Racket implementations are slower than the optimized C versions in the sequential case, all three parallel Racket versions are able to yield better performance than sequential C after employing relatively small numbers of processors (2 for both convolution and MV-sparse, and 6 for mergesort). The parallel convolution implementation exhibits good scaling through the maximum number of processors available on both machine configurations, owing to the tight nested-loop structure of the algorithm, which involves only floating-point computations. Here the parallel convolution is able to avoid slow-path exits by using Racket's floating point-specific primitives. The Racket benchmarks also use unsafe versions of the arithmetic and array indexing operations.

Figures 38 and 40 show running time for the Racket implementations of the NAS IS and FT benchmarks, while figures 39 and 41 show speedup curves. Racket's performance is compared with both sequential and parallel Java reference implementations.

While sequential Racket implementations for these benchmarks are considerably slower than the Java implementations, the parallel implementations scale better. However, this scaling does not allow us to catch up with the parallel Java implementation in absolute terms. It is likely that, especially in the case of the IS benchmark, this is due to the majority of work being performed in the parallel portion of the benchmark being array accesses (e.g. `vector-ref` and `vector-set!` in Racket), operations, which are more heavily optimized in the Java runtime system. The

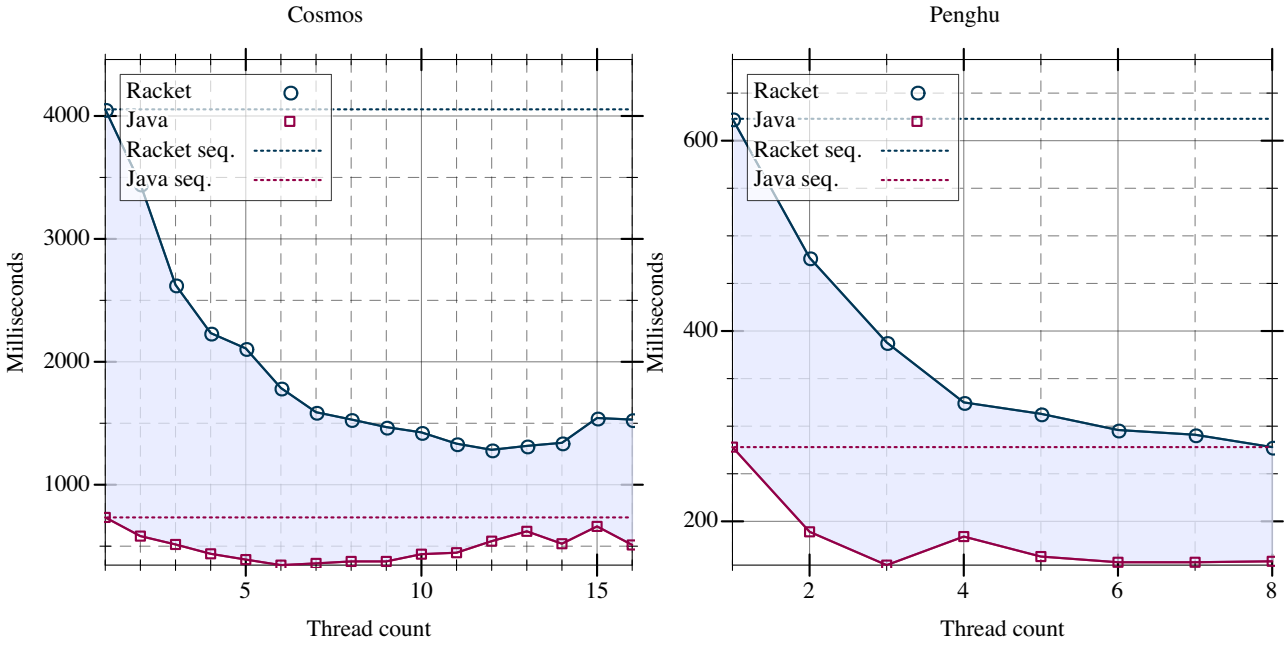

Figure 38: NAS Integer Sort wall-clock time

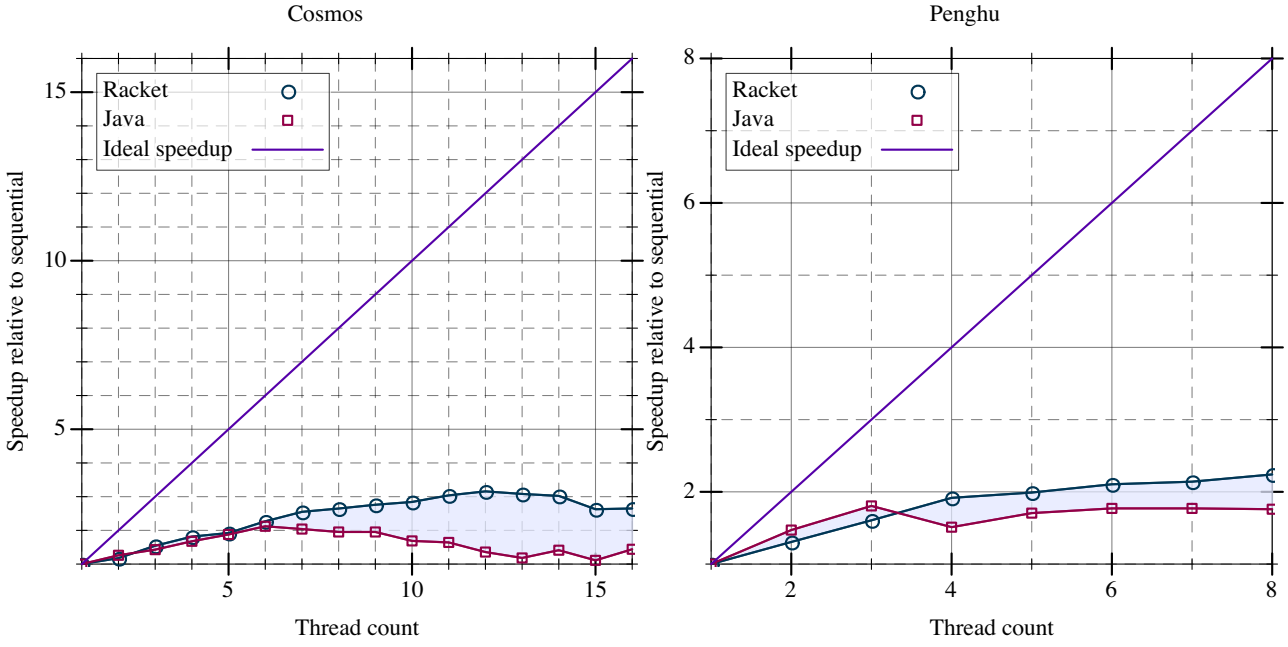

Figure 39: NAS Integer Sort speedups

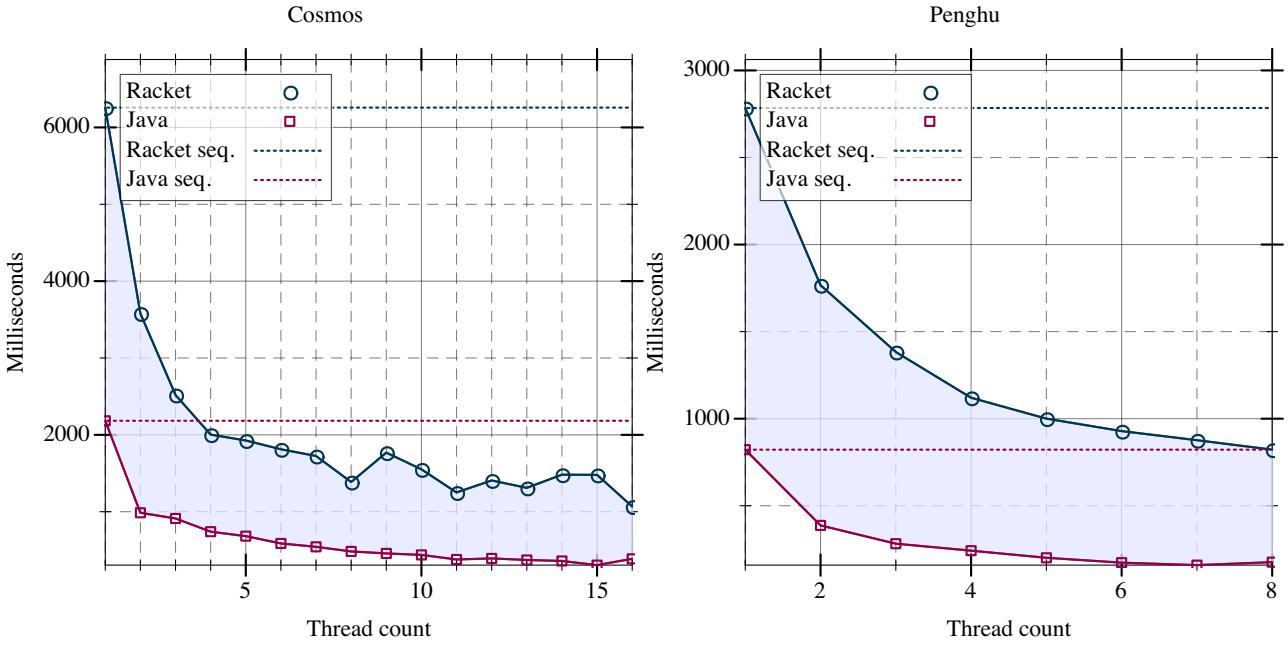

Figure 40: NAS Fourier Transform wall-clock time

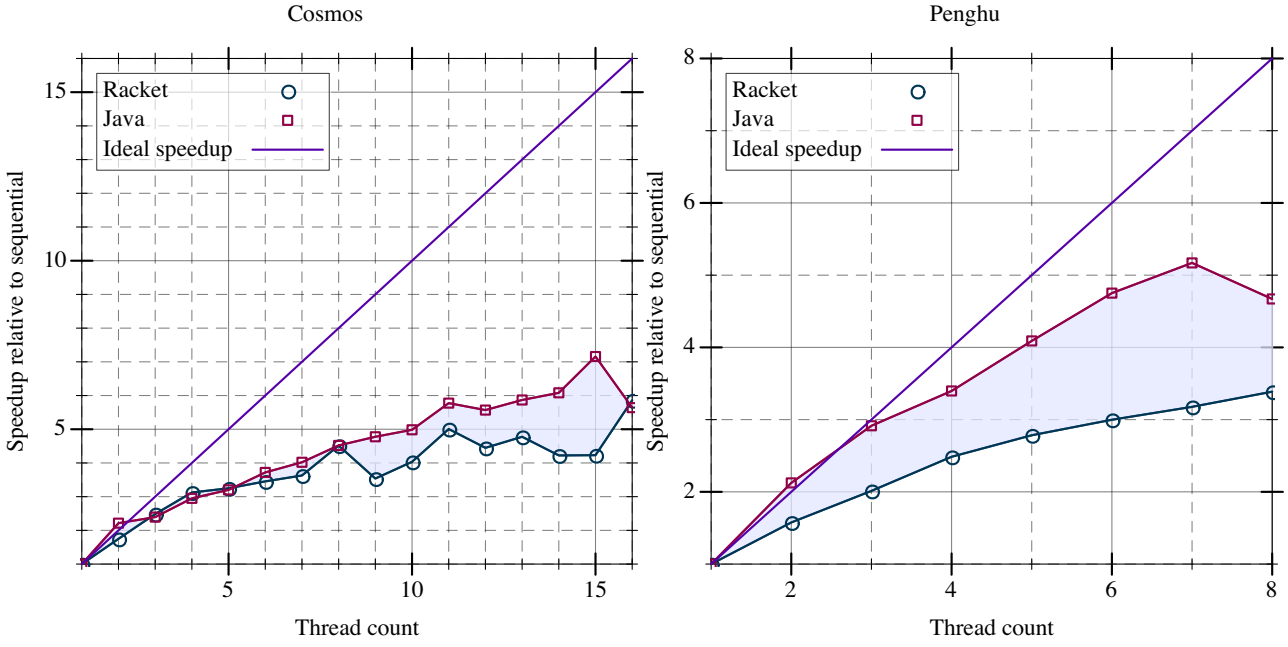

Figure 41: NAS Fourier Transform speedups

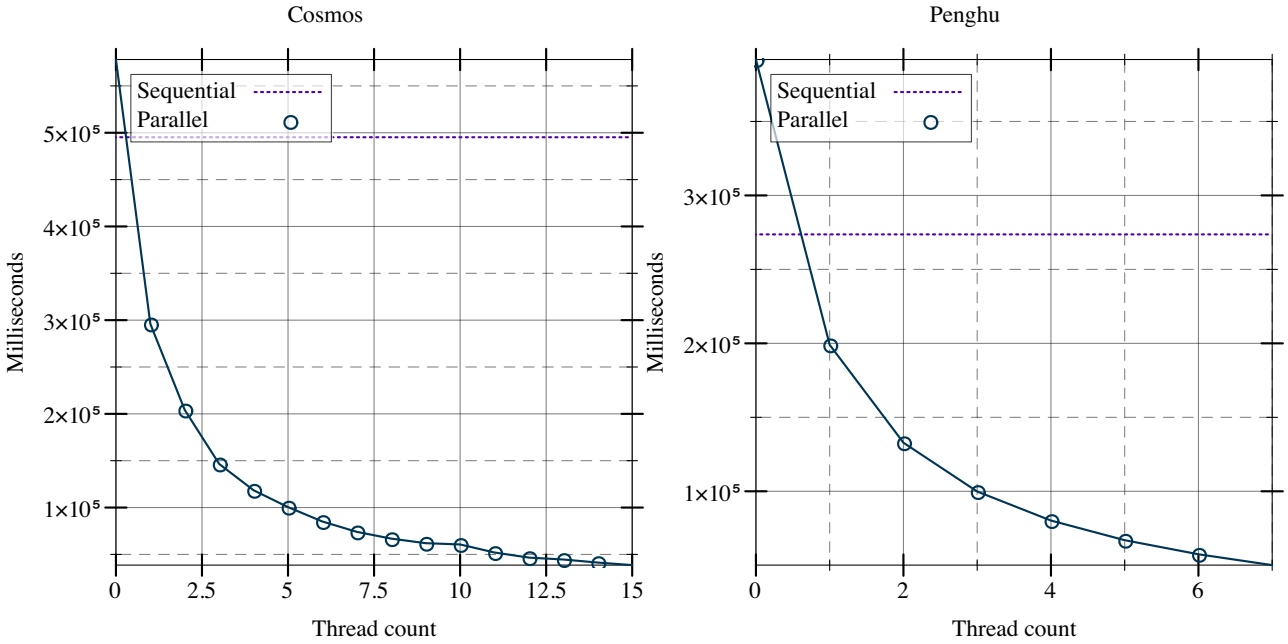

Figure 42: Signal convolution wall-clock times (Parrot)

Racket with futures implementation of NAS FT, however, is able to outperform sequential Java after 3 processors (on the Cosmos machine configuration), and generally exhibits similar scaling characteristics to the reference parallel Java implementation.

As with the self-developed benchmarks (Signal Convolution, Mergesort, and MV-Sparse), the results demonstrate that the approach to incremental parallelization of sequential runtimes can lead to reasonable parallel performance.

Parrot

The prototype implementation of Parrot with futures was only tested using the convolution benchmark. The results, as can be seen in figures 42 and 43, are comparable to those seen with Racket with futures in terms of speedup (compare to figures 32, 34, and 36). This is supportive of the

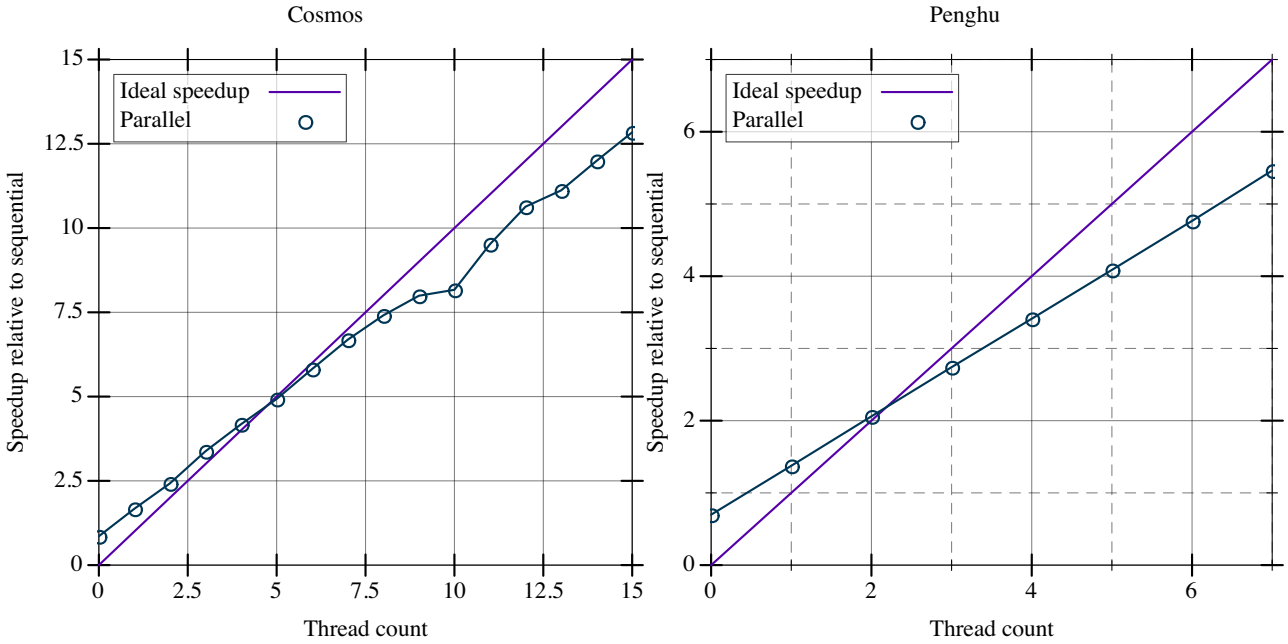

Figure 43: Signal convolution speedup (Parrot)

claim that, overall, the approach leads to reasonable parallel performance (it yields effective parallel programming constructs).

It is important to point out that the raw performance is not comparable, however. This is the result of the implementation being preliminary (contrast figure 28 and figure 29). More specifically, the current implementation is based on a version of Parrot in which the JIT is in a state of flux. Our benchmark results reflect *interpreted* performance without the JIT, and thus should be taken with a grain of salt. The current implementation of futures in Parrot only supports operations on unboxed floating-point numbers.

Caveats notwithstanding, the results for the Parrot with futures proof-of-concept implementation suggest that the approach can be generalized to other high level language implementations.

Part 6

Related Work

Language Support for Shared-Memory Parallelism

This work builds on the ideas of futures from [23], a parallel dialect of Scheme. Parallelism in MultiLisp is also expressed via `future`. However, MultiLisp does not require an explicit `touch` on the part of the programmer. Instead, touches are implicitly performed whenever the value of a future is needed. Also unlike our work, futures in Multilisp always execute in parallel, whereas ours only execute in parallel when it is safe (based on the constraints of the runtime system).

Many language communities face the problem of retrofitting their implementations to support parallelism. The typical approach is to allow arbitrary threads of computation to run in parallel, and to adapt the runtime system as necessary. Some succeed in the transition through substantial re-implementation efforts; threads in the original Java 1.1 virtual machine where implemented as user threads, but many re-implementations of Java now support threads that use hardware concurrency. Others succeed in the transition with the help of their language designs; Erlang and Haskell are prime examples of this category, where the purely functional nature of the language (and much of the language implementation) made a transition to support for parallelism easier, through it required substantial effort [2, 26]. Finally, many continue to struggle with the transition; our own attempts to map Racket-level threads to OS-level threads failed due to the complexity of the runtime system, and frequent attempts to rid the main Python and Ruby implementations of the global interpreter lock (GIL) have generally failed [6, 34]. An attempt to support OS-level threads in OCaml has so far produced an experimental system [10].

Our approach of introducing parallelism through constrained futures is somewhat similar to letting external C code run outside the GIL (and therefore concurrently) in Python or Ruby. Instead of pushing parallelism to foreign libraries, however, our approach draws parallelism into a subset of the language. Our approach is also similar to adding special-purpose parallel operators to a language, as in data-parallel operations for Data Parallel Haskell [11]; instead of constraining

the set of parallel operations a priori, however, our approach allows us to gradually widen the parallelism available through existing constructs.

In adding support for parallelism to Racket, we hope to move toward the kind of support for parallelism that is provided by languages like NESL [8] , X10 [13], Chapel [12], Fortress [1], Manticore [20], and Cilk [9], which were all designed to parallelism from the start. Adding parallelism to a sequential run-time system is a different problem than designing a parallel language from scratch, but we take inspiration from designs that largely avoid viewing parallelism as concurrent threads of arbitrary computation.

Concurrent Caml Light [15] relies on a compile-time distinction between mutable and immutable objects to enable thread-local collection. Concurrent Caml Light gives its threads their own nurseries, but the threads all share a global heap. Concurrent Caml Light is more restrictive than Racket places. In Concurrent Caml Light, only immutable objects can be allocated from thread-local nurseries; mutable objects must be allocated directly from the shared heap. Concurrent Caml Light presumes allocation of mutable objects is infrequent and mutable objects have longer life spans. Racket's garbage collector performs the same regardless of mutable object allocation frequency or life span.

Erlang [37] is a functional language without destructive update. The Erlang implementation uses a memory management system similar to Racket's master and place-local GCs. All Erlang message contents must be allocated from the shared heap; this constraint allows O(1) message passing, assuming message contents are correctly allocated from the shared heap, and not from the Erlang process's local nursery. The Erlang implementation employs static analysis to try to determine which allocations will eventually flow to a message send and therefore should be allocated in the shared heap. Since messages are always allocated to the shared heap, Erlang must collect the share heap more often then Racket, which always allocates messages into the destination place's

local heap. Erlang's typical programming model has many more processes than CPU cores and extensive message exchange, while places are designed to be used one place per CPU core and with less message-passing traffic.

Haskell [31, 32] is a pure functional language with support for concurrency. Currently, Haskell garbage collection is global; all threads must synchronize in order to garbage collect. The Haskell implementors plan to develop local collection on private heaps, exploiting the predominance of immutable objects similarly to Concurrent Caml Light's implementation. In contrast to pure functional languages, Racket programs often include mutable objects, so isolation of local heaps, not inherent immutability, enables a place in Racket to independently garbage-collect a private heap.

Manticore [19] is designed for parallelism from the start. Like Erlang and Haskell, Manticore has no mutable datatypes. In contrast, places add parallelism to an existing language with mutable datatypes. As the implementation of places matures, we hope to add multi-level parallelism similar to Manticore.

Matlab provides programmers with several parallelism strategies. First, compute intensive functions, such as BLAS matrix operations, are implemented using multi-threaded libraries. Simple Matlab loops can be automatically parallelized by replacing `for` with `parfor`. Matlab's automatic parallelization can handle reductions such as min, max and sum, but it does not parallelize loop dependence. Matlab also provides task execution on remote Matlab instances and MPI functionality. Rather than adding parallelism through libraries and extensions, places integrate parallelism into the core of the Racket runtime.

Python's multiprocessing library [35] provides parallelism by forking new processes, each of which has a copy of the parent's state at the time of the fork. In contrast, a Racket place is conceptually a pristine instance of the virtual machine, where the only state a place receives from its creator is its starting module and a communication channel. More generally, however, Python's

multiprocessing library and Racket's places both add parallelism to a dynamic language without retrofitting the language with threads and locks.

Communication between Python processes occurs primarily through OS pipes. The multiprocessing library includes a shared-queue implementation, which is implemented by using a worker thread to send messages over pipes to the recipient process. Any "picklable" (serializable) python object can be sent through a multiprocessing pipe or queue. Python's multiprocessing library also provides shared-memory regions implemented via `mmap()`. Python's pipes, queues and shared-memory regions must be allocated prior to forking children, which need to use them. Racket's approach offers more flexibility in communication; channels and shared-memory vectors can be created and sent over channels to already-created places; and channels can communicate immutable data without the need for serialization.

Python and Ruby implementors, like Racket implementors, have tried and abandoned attempts to support OS-scheduled threads with shared data [6, 34, 38]. All of these languages were implemented on the assumption of a single OS thread—which was a sensible choice for simplicity and performance throughout the 1990s and early 2000s—and adding all of the locks needed to support OS-thread concurrency seems prohibitively difficult. A design like futures could be the right approach for those languages, too.

Futures are based on similar constructs in Multilisp [23], but differ in that Racket's futures require an explicit touch and may not exhibit parallelism because of the legacy code in the runtime system.

Profiling and Visualization

Threadscope [25] is a visualization tool for Parallel Haskell. It organizes trace information into a timeline displaying work done by individual Haskell Execution Contexts, which roughly correspond to operating system threads. The tool uses a non-allocating, buffered per-thread logging

scheme that incurs minimal overhead (similar to Racket's future logging). Berthold and Loogen [7] developed the Eden Trace Viewer, a similar tool, for Eden, a parallel extension to Haskell that supports distributed computation. The Eden viewer is designed to assist in performance tuning and provides visual displays of state and communication data at the machine, thread, and process level.

Runciman and Wakeling [36] demonstrated the use of a "parallelism profile graph" to aid in the subtle problem of the optimal placement of parallel annotations. Instead of processing logs from actual parallel execution, they used a compiler extension to produce programs that simulate parallelism, which generated logs used to construct their profile graphs. Racket has a similar construct: `would-be-future`, that runs sequentially, but produces the same logging that `future` would. This helps the Racket programmer determine the extent to which his/her program encounters barricades.

Various tools have been developed to aid in optimizing parallel programs designed for distributed environments. Jumpshot [42] is a free visualizer for MPI programs that displays both communication patterns and timelines of process state on a per-node and global basis.

VAMPIR [30] is a commercial visualization tool which can be used to profile programs in both distributed and shared-memory environments. The tool is capable of using multiple methods of instrumentation depending on program environment and libraries used (MPI, OpenMP), and offers graphical displays for a large array of metrics. VAMPIR uses an open trace format [29], which is shared with a number of other tools. While reusing that trace format would give us interoperability with these existing tools, our traces record Racket-specific information that cannot be accomodated by OTF; restricting logging to such a format would greatly limit the usefulness of our visualizer.

Kergommeaux et al. [28] developed Paje for ATHAPASCAN, a system leveraging two levels of parallelism using a distributed network of shared-memory multi-processor nodes. Paje reads log information into a simulator which is used to produce interactive timeline visualizations at the network, node, and processor levels.

Cilkview [24] is an analysis tool targeting the Cilk++ extensions to C++. It is able to give upper and lower bounds on the parallelism available in a specific program by running it sequentially and recording some information about its behavior. It also provides a harness to run the program at different levels of parallelism to test the results of its analysis.

IBM's Tuning Fork [3] is a trace-based visualization tool targeting real-time systems. It provides both real-time and replayable information display in an extensible framework with some innovative default visualizations, notably an "oscilloscope" view that can show behavior across a wide range of time scales.

Bibliography

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, and G. Steele. The Fortress Language Specification. 2011. `https://projectfortress.java.net/`

[2] Joe Armstrong. A history of Erlang. Hopl Iii: Proceedings Of The Third Acm Sigplan Conference On The History Of Programming Languages edition, 2007.

[3] David F. Bacon, Perry Cheng, Daniel Frampton, and David Grove. TuningFork: Visualization, Analysis and Debugging of Complex Real-time Systems. IBM Research, RC24162, 2007.

[4] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, 1991.

[5] Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. In *Proc. Symposium on Artificial Intelligence Programming Languages*, 1977.

[6] David Beazley. Understanding the Python GIL. PyCon 2010, 2010.

[7] Jost Berthold and Rita Loogen. Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In *Proc. Intl. Conf. on Parallel Computing: Architectures, Algorithms, and Applications*, 2007.

[8] Guy Blelloch. Implementation of a Portable Nested Data-Parallel Language. *Proc. of the Symposium on Principles and Practice of Parallel Programming*, pp. 102–111, 1993.

[9] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *ACM SIGPLAN Notices*, pp. 207–216, 1995.

[10] Mathias Bourgoin, Adrien Jonquet, Emmanuel Chailloux, Benjamin Canou, and Philippe Wang. OCaml4Multicore. 2007. `http://www.algo-prog.info/ocmc/web/index.php`

[11] Manuel T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *Proc. Workshop on Declarative Aspects of Multicore Programming*, pp. 10–18, 2007.

[12] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programming and the Chapel Language. *Intl. Journal of High Performance Computing Applications*, pp. 291–312, 2007.

[13] Phillippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarker. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[14] Oracle Corporation. Java Threads in the Solaris Environment. 2010. `http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/`

[15] Damien Doligez and Xavier Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proc. ACM Symp. Principles of Programming Languages*, 1993.

[16] Robert Bruce Findler and Matthew Flatt. Slideshow: Functional Presentations. In *Proc. ACM Intl. Conf. Functional Programming*, 2006.

[17] Matthew Flatt, Eli Barzilay, and Robby Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM Intl. Conf. Functional Programming*, 2009.

[18] Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

[19] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proc. ACM Intl. Conf. Functional Programming*, 2008.

[20] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yinggi Xiao. Manticore: A Heterogeneous Parallel Language. In *Proc. Workshop on Declarative Aspects of Multicore Programming*, pp. 37–44, 2007.

[21] Michael A. Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Implementation of NAS Parallel Benchmarks in Java. NAS Technical Report NAS-02-009, 2002.

[22] Martin Gardner. Mathematical games: the fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, pp. 120–123, 1970.

[23] Robert H. Halstead Jr. A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 1985.

[24] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview Scalability Analyzer. In *Proc. ACM Symp. Parallelism in Algorithms and Architectures*, pp. 145–156, 2010.

[25] Don Jones Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proc. ACM Symp. Haskell*, pp. 81–92, 2009.

[26] S.P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. ACM Symp. Principles of Programming Languages*, 1996.

[27] Cezary Juszczak. Fast mergesort implementation based on half-copying merge algorithm. 2007. http://kicia.ift.uni.wroc.pl/algorytmy/mergesortpaper.pdf

[28] J. Chassin de Kergommeaux, B. Stein, and P.E. Bernard. Paje, an interactive tool for tuning multi-threaded parallel applications. *Parallel Computing* 26, pp. 1253–1274, 2000.

[29] Andreas Knupfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Proc. Intl. Conf. on Computational Science*, pp. 526–533, 2006.

[30] Andreas Knupfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Muller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. *Tools for High Performance Computing* 4, pp. 139–155, 2008.

[31] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proc. Intl. Symp. on Memory Management*, 2008.

[32] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *Proc. ACM Intl. Conf. Functional Programming*, 2009.

[33] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory. In *Proc. ACM Symp. Principles and Practice of Parallel Programming*, 2014.

[34] Python Software Foundation. Python design note on threads. 2008. http://www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock

[35] Python Software Foundation. multiprocessing — Process-based "threading" interface. 2011. http://docs.python.org/release/2.6.6/library/multiprocessing.html#module-multiprocessing

[36] Colin Runciman and David Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *Proc. Glasgow WORKSHOP on Functional Programming*, pp. 236–251, 1993.

[37] Konstantinos Sagonas and Jesper Wilhelmsson. Efficient Memory Management for Concurrent Programs that use Message Passing. *Science of Computer Programming* 62(2), pp. 98–121, 2006.

[38] Werner Schuster. Future of Threading and Garbage Collection in Ruby - Interview with Koichi Sasada. 2009. `http://www.infoq.com/news/2009/07/future-ruby-gc-gvl-gil`

[39] Kevin Tew. Places: Parallelism for Racket. PhD dissertation, University of Utah, 2012.

[40] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda. Places: Adding Message-Passing Parallelism to Racket. In *Proc. Dynamic Languages Symposium*, 2011.

[41] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symp. Principles of Programming Languages*, 2008.

[42] Omer Zaki, Ewing Lusk, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications* 13, pp. 277–288, 1999.