Seeing the Futures: Profiling Shared-Memory Parallel Racket

James Swaine
Northwestern University
james.swaine@eecs.northwestern.edu

Burke Fetscher
Northwestern University
burke.fetscher@eecs.northwestern.edu

Vincent St-Amour
Northeastern University
stamourv@ccs.neu.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

This paper presents the latest chapter in our adventures coping with a large, sequentially-tuned, legacy runtime system in today's parallel world. Specifically, this paper introduces our new graphical visualizer that helps programmers understand how to program in parallel with Racket's futures and, to some extent, what performs well in sequential Racket.

Overall, our experience with parallelism in Racket is that we can achieve reasonable parallel performance in Racket without sacrificing the most important property of functional programming language implementations, namely safety. That is, Racket programmers are guaranteed that every Racket primitive (and thus all functions built using Racket primitives) will either behave properly, or it will signal an error explaining what went wrong.

That said, however, it is challenging to understand how to best use futures to achieve interesting speedups, and the visualizer is our attempt to more widely disseminate key performance details of the runtime system in order to help Racket programmers maximize performance.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.1.3 [*Concurrent Programming*]: Parallel Programming

Keywords Racket, parallel functional programming, performance tuning

1. Introduction

Racket (Flatt and PLT 2010)'s futures (Swaine et al. 2010) are a construct for parallelism designed specifically to work around the problem of making Racket's large, legacy, sequentially-tuned runtime system thread-safe. In order to achieve thread safety, we allow futures to execute only a subset of Racket primitives that are known to be free of side-effects with respect to the runtime system's internal state. At a high level, the full set of Racket primitives can be partitioned into three categories:

- *safe* - okay to execute in parallel;
- *barricaded* - halts parallel execution until the future is touched;
- *atomic* - requires synchronization with the runtime thread, but will allow the future to continue to execute in parallel once completed.

The barricaded category is what makes futures safe, ensuring that parallel execution does not corrupt the runtime system's internal state. Specifically, when a future encounters a primitive in the barricaded category, the future simply halts parallel execution. In order to complete the computation that the future began, it is necessary to explicitly touch the future, which ensures that the future completes on the one thread that is allowed to manipulate the shared state of the runtime system.

To a first approximation, safe primitives are the same ones that the JIT compiler has a special fast path for. In Racket, this means that some primitives are safe only when given specific arguments. For example, when `+` receives fixnums (roughly machine-word sized integers) or floating point numbers, then the call to `+` is safe. When it encounters other kinds of numbers (e.g., bignums, rationals, or complex numbers), or encounters non-numbers, then `+` must be barricaded, since those operations may take a long time to complete, may use an unbounded amount of memory, or may involve jumping to an unknown exception handler (possibly one recorded in a portion of the stack that the future shares with the main computation).

The atomic category currently consists of only two operations: memory allocation and JIT compilation. We judged that these two operations are so common and so difficult to eliminate from ordinary Racket code that, rather than barricading them (since they manipulate shared runtime system state), it is better to synchronize with the runtime thread. Accordingly, a Racket programmer can still benefit from parallelism if he/she merely reduces the amount of memory allocation or JIT compilation that happens, instead of eliminating them entirely.

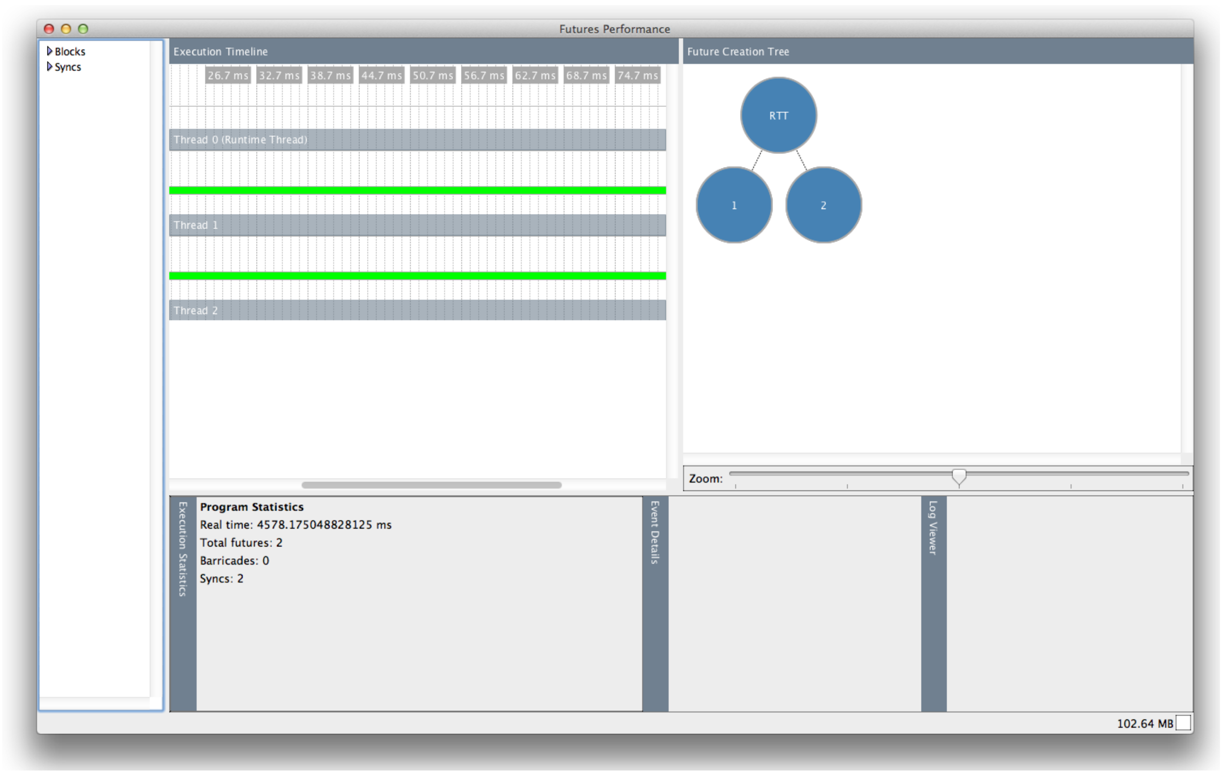
Consider the following sequential program, which adds the summation of two lists:

```
(define a (build-list 1000 (lambda (x) x)))
(define b (build-list 1000 (lambda (x) x)))

(+ (foldl + 0 a) (foldl + 0 b))
```

Though a trivial example, we might attempt to parallelize the program by creating a future for one of the calls to `foldl`. A comparison of the running times of the two versions is surprising: the parallel version's running time is almost the same as the sequential version. Because `+` is not being directly applied, the JIT compiler

FHPC'12 September 15, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1577-7/12/09. . . $10.00



Program Statistics
Real time: 4578.175048828125 ms
Total futures: 2
Barricades: 0
Syncs: 2

Figure 1: The Visualizer

does not inline it and thus each application of + inside `foldl` requires a barricade, thwarting our attempt at parallelism.

Naturally, this state of affairs calls for some help so that a Racket programmer can understand why a program behaves poorly when run in a future. This desire for tool support is exacerbated by the incremental nature of future support in Racket. That is, the futures implementation is designed to enable an incremental approach to runtime parallelization, in which primitives requiring barricades may be manually rewritten one-by-one over time to support more parallel programs.

Although an incremental approach makes adding futures to Racket possible at all, it results in an implementation in which the set of barricaded primitives is constantly shrinking. Thus, even experienced futures programmers cannot always rely on preexisting intuition about which operations are safe for parallelism.

Our experience attempting to write parallel programs that exhibit good scaling shows that it is often very difficult to understand how the underlying implementation affects a given program's potential for parallelism. This paper introduces our graphical profiling tool and demonstrates how it can help Racket programmers refine a program to avoid limitations imposed by the implementation.

The paper proceeds as follows: section 2 introduces the futures API and the visualizer. Section 3 explores how to use the visualizer to build a fast parallel implementation of a fast Fourier transform. Section 4 explains the tracing infrastructure used by Racket to record the information the visualizer requires and finally section 5 discusses related work.

2. Futures and the Visualizer

Programmers can spawn and join on parallel tasks via the following primitives:

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

Creates a future construct encapsulating a thunk, and indicates to the runtime system that the body of the thunk is a good candidate for parallel execution.

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

Forces the evaluation of the thunk encapsulated by *f* (if it hasn't been evaluted already) and blocks until its evaluation is completed.

For example,

```
(let ([f (future (lambda () (+ 1 2)))])
  (list (+ 3 4) (touch f)))
```

asks the system to evaluate `(+ 1 2)` and `(+ 3 4)` in parallel, and returns the two results in a list.

The visualizer window is shown in Figure 1, with the results of running this program:

```
#lang racket
(define (count-down n)
  (unless (zero? n)
    (count-down (- n 1))))
(define (lots-of-nothing)
  (count-down 1000000000))
(define f1 (future lots-of-nothing))
(define f2 (future lots-of-nothing))
(touch f1)
(touch f2)
```

The window consists of four panes. The left-most portion of the window shows the barricaded primitives encountered during the run, as well as the reasons why a future had to synchronize with the main, runtime thread. In this case, there aren't any. The upper middle portion shows what each thread in the system was doing, as a timeline. In this case, each thread was running a separate future, and the green bars indicate the future was doing work (as opposed to waiting for work or waiting for the runtime system). The upper right portion shows a tree indicating which futures created which other futures. In this case, the root node is the runtime system thread, which created the two futures. Finally, the lower portion of the window gives an overview of the program, but when the mouse passes over a portion of the upper middle pane, it gives more information about the operations happening at that point in the timeline.

3. Parallelizing an FFT

To give a sense of how the visualizer can help guide the implementation of a parallel algorithm, this section works through the implementation of a fast Fourier transform (FFT). All of the timings in this section (except as discussed in the last subsection) were measured on an 11" MacBook Air with 2 hyperthreaded 1.8 GHz Core i7's, for a total of four cores from Racket's perspective.

3.1 Sequential FFT

To start, a sequential version of the Cooley–Tukey fast Fourier transform, `fft0`, is shown in Figure 2. It accepts a series of samples of a function in the time domain and returns samples of that function from the frequency domain.

The Cooley–Tukey FFT is a divide-and-conquer recursive algorithm. If the input contains a single sample, it is returned unchanged. Otherwise, the function exploits a mathematical identity of the discrete Fourier transform, namely that it can be expressed in terms of two interleaved Fourier transforms of half the size of the original. That is, the input is split into two parts: one containing the samples with even indices and another containing the samples with odd indices. This is the decimation phase, implemented by the `decimate-in-time` function. The Fourier transform is recursively applied to both of the smaller signals. A "twiddle factor" is then applied to the transform of the odd samples, as seen in the `twiddle-factor` function. Finally, the results are combined to form the complete Fourier transform by appending the pointwise sum of the results and their pointwise difference.

The mathematical identity that this algorithm relies on only holds if the two sublists are of equal length and thus our FFT algorithm only works on lists whose length is a power of 2. This is not, generally speaking, an important restriction, since the users of the FFT algorithm typically have the freedom to choose the number of sample points of the function.

The code features a few Racketisms: functions in Racket can return multiple values, which can then be bound separately using the special `define-values` binding construct. In this case, `decimate-in-time` returns the two sublists and the `define-values` on line 4 catches the two values, binding them to `ss` and `ts`. The `for/list` construct on line 11 describes a list comprehension: the variables `b` and `c` are successively bound to each element in the sequences `(in-list bs)` and `(in-list cs)`, respectively. The two lists are traversed in lock-step and the body of the comprehension is evaluated for each element of `bs` and `cs`. Then, the results of the body of the loop are collected into a list. The `for/fold` loop (on line 16) is a generalization of a `for` loop that has accumulators: `evens` and `odds` in our example. They are both initially bound to empty lists. The body of the loop (lines 20–22) is expected to return as many values as there are accumulators. The second set of bindings in `for/fold` specifies the sequences

```
 0 (define (fft0 as)
 1   (cond
 2     [(= (length as) 1) as]
 3     [else
 4      (define-values (ss ts)
 5        (decimate-in-time as))
 6      (define bs (fft0 ss))
 7      (define cs (twiddle-factor (fft0 ts)))
 8      (append (for/list ([b (in-list bs)]
 9                         [c (in-list cs)])
10                (+ b c))
11              (for/list ([b (in-list bs)]
12                         [c (in-list cs)])
13                (- b c)))]))
14
15 (define (decimate-in-time as)
16   (for/fold ([evens '()]
17              [odds  '()])
18       ([i (in-naturals)]
19        [a (in-list (reverse as))])
20     (if (odd? i)
21         (values (cons a evens) odds)
22         (values evens (cons a odds)))))
23
24 (define (twiddle-factor cs)
25   (define n (length cs))
26   (for/list ([k (in-naturals)]
27              [c (in-list cs)])
28     (* c (exp (/ (* pi 0+1i k) n)))))
```

Figure 2: Sequential Cooley–Tukey FFT

over which the loop iterates, as with `for/list` above. In this case, `i` iterates over the infinite sequence of natural numbers, and `a` iterates in reverse over the elements of `as`. All this taken together means that `decimate-in-time` returns the even and odd indexed sublists as described above. For example:

```
> (decimate-in-time '(1 2 3 4))
'(1 3)
'(2 4)
```

3.2 A First-cut Parallel FFT

The sequential FFT implementation has a nice decomposition structure that immediately suggests a parallelization strategy. Specifically, the two recursive calls in `fft0` can be computed in parallel. Figure 3 shows the revised implementation.

The function `fft1` creates two futures: one that simply recurs and another that recurs, and then applies the twiddle factor. Once the two tasks have been spawned, `fft1` touches both of them, waiting for the results.

As has become folklore by now, a first-cut attempt to parallelize an implementation actually slows it down, and this program is no exception. Indeed, this version is about 12 times slower than the sequential implementation.

Opening the visualizer window and looking at the future creation tree (as shown in figure 4), immediately suggests why: when passing a list of 64 elements to `fft1`, our parallel implementation creates 126 futures despite there being only 4 cores available on our machine. As we will see, the implementation can cope with more futures than cores, but with this many more, the time required to

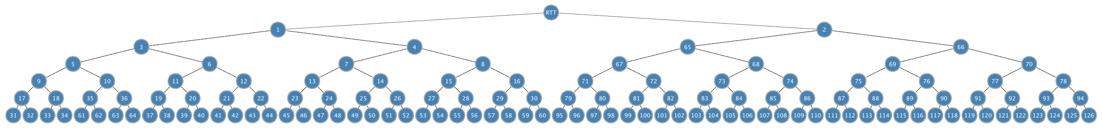


Figure 4: Future Creation Tree

```
29 (define (fft1 as)
30   (cond
31     [(= (length as) 1) as]
32     [else
33      (define-values (ss ts)
34        (decimate-in-time as))
35      (define bs
36        (future
37         (λ () (fft1 ss))))
38      (define cs
39        (future
40         (λ () (twiddle-factor (fft1 ts)))))
41      (append
42       (for/list ([b (in-list (touch bs))]
43                  [c (in-list (touch cs))])
44         (+ b c))
45       (for/list ([b (in-list (touch bs))]
46                  [c (in-list (touch cs))])
47         (- b c)))]))
```

Figure 3: FFT with Too Much Parallelism

```
48 (define (fft2 as)
49   (fft2/depth as (init-d)))
50
51 (define (init-d)
52   (define pc (processor-count))
53   (fl->fx (ceiling (/ (log pc) (log 2)))))
54
55 (define (fft2/depth as d)
56   (cond
57     [(= (length as) 1) as]
58     [(= d 0) (fft0 as)]
59     [else
60      (define-values (ss ts)
61        (decimate-in-time as))
62      (define bs
63        (future
64         (λ ()
65           (fft2/depth ss (- d 1)))))
66      (define cs
67        (future
68         (λ ()
69           (twiddle-factor
70            (fft2/depth ts (- d 1))))))
71      (append
72       (for/list ([b (in-list (touch bs))]
73                  [c (in-list (touch cs))])
74         (+ b c))
75       (for/list ([b (in-list (touch bs))]
76                  [c (in-list (touch cs))])
77         (- b c)))]))
```

Figure 5: Depth-limited FFT

create and manage the futures themselves dominates the time required to transform the list.[1]

3.3 Limiting the Amount of Parallelism

Figure 5 shows a revision of our FFT implementation that adds a depth parameter (`d`) to control the number of futures.

The `fft2/depth` function (line 55) is now the main workhorse; in addition to accepting a list to transform, it also accepts the depth parameter `d`; if `d` is `0`, it stops spawning new futures and falls back to the original, sequential algorithm, `fft0`.

The initial value of the depth is computed by taking the base 2 logarithm of the number of processors. Since the algorithm works only on lists with length that is a power of two, we know exactly how much parallelism we can exploit. The `init-d` function, on line 51, shows how this is computed in Racket (the `fl->fx` function converts a floating point integer to an exact, fixnum integer). Otherwise, the code in figure 5 is the same as the code in figure 3, except that the depth parameter is now being passed around.

Timing this version shows that it is still slower than the original, sequential FFT implementation, but only by about 10%. To see why, we turn to the timeline portion of the visualizer. Figure 6's upper half shows a segment of the timeline, about 3.8 milliseconds into the trace of the parallel FFT. At a first glance, it looks almost like interesting things are happening in parallel, albeit not very much work. Mousing over any of those events in the window, as shown in the bottom half of figure 6, reveals the problem.

When moving the mouse over an event, the visualizer draws purple lines that connect all of the events on that future and these lines tell us that even those apparently parallel operations are really all happening on the same future, which is itself moving between different threads in the runtime system. In other words, our parallel program is effectively sequential.

If we turn to the "blocks" section of the visualizer window, we see that the functions `+`, `*`, `/`, `exp`, `-`, `append`, and `touch` are all barricaded. Accordingly, as soon as a future uses one of those operations it simply pauses to wait for its `touch` and thus does not execute in parallel.

[1] As you probably already noticed, figure 4 is not a screengrab from the futures visualizer; instead it is a high-resolution direct rendering of the future creation tree. This is because our future visualizer internally constructs `pict` objects (from a functional picture library included with Racket) when drawing them in the visualizer window, but these picts can also be rendered as `pdf` and included in papers, which is what we have done here and will do with the rest of the pictures in this paper. In addition to improving the print quality of this paper, this approach has the advantage that we can script the creation of the pictures and thus be confident that they are unlikely to be out of sync with the prose and the code figures.

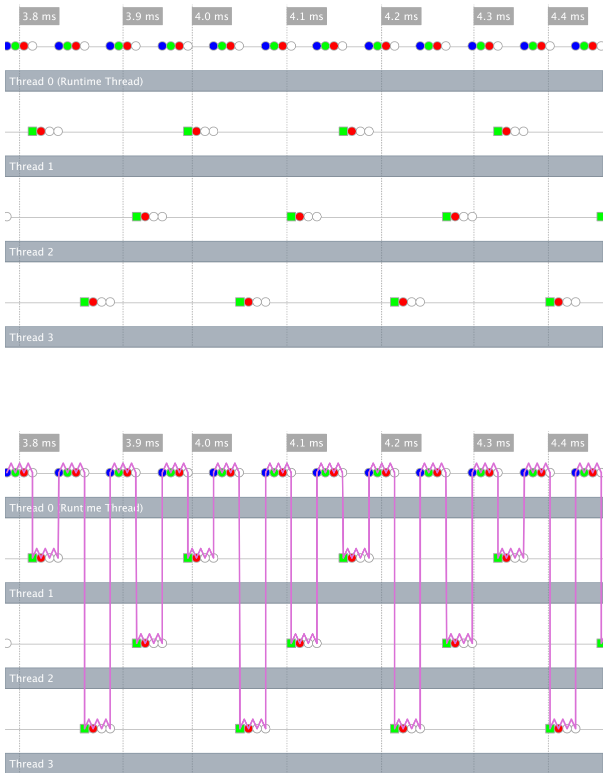


Figure 6: Future Timeline for Depth-limited FFT

3.4 Introducing Typed Racket

There are two problems with the arithmetic operations: they occasionally operate on mixed arguments (e.g., floats and fixnums, or floats and complexes) and they often operate on complex numbers. Neither of those operations work in parallel futures.

Happily, porting our program to Typed Racket (Tobin-Hochstadt and Felleisen 2008) helps with both problems (and improves the sequential performance, to boot). For the former, the type system can tell us when we use mixed-mode arithmetic;[2] for the latter, Typed Racket will automatically break up complex numbers into their real and imaginary components and pass them to and return them from functions separately, replacing arithmetic on complex numbers by arithmetic on floats.

Figure 7 shows the revised code for the main FFT function. The main difference is the addition of type specifications to the top-level functions. In addition, the `for/list` expressions must be changed into `for/list:` expressions so they can be annotated with types. Converting `decimate-in-time` and `twiddle-factor` requires similar revisions.

These changes eliminate the arithmetic operation barricades, leaving `append` and `touch`. Eliminating the `touch` barricade is not possible, due to the nature of futures, but `append` is more interesting. Indeed, why should we expect `append` to be barricaded at all? The reason is wrapped up in the history of Racket and the incremental nature of the implementation of futures. In short, `append` is implemented as part of the runtime system and thus has access to state that it could, in theory, corrupt. All such functions, by default, are considered unsafe and barricades are automatically put in place.

[2] In general, Typed Racket is happy to type-check such mixed-mode arithmetic (St-Amour et al. 2012), but we can add type annotations that force consistent types for all the arguments.

```
(: fft3 ((Listof Float-Complex)
         ->
         (Listof Float-Complex)))
(define (fft3 as)
  (fft3/depth as (init-d)))

(: fft3/depth ((Listof Float-Complex)
               Integer
               ->
               (Listof Float-Complex)))
(define (fft3/depth as d)
  (cond
    [(= (length as) 1) as]
    [(= d 0) (fft3/seq as)]
    [else
     (define-values (ss ts)
       (decimate-in-time as))
     (define bs
       (future
        (λ ()
          (fft3/depth ss (- d 1)))))
     (define cs
       (future
        (λ ()
          (twiddle-factor
           (fft3/depth ts (- d 1))))))
     (append
      (for/list: : (Listof Float-Complex)
        ([b (in-list (touch bs))]
         [c (in-list (touch cs))])
        (+ b c))
      (for/list: : (Listof Float-Complex)
        ([b (in-list (touch bs))]
         [c (in-list (touch cs))])
        (- b c)))]))
```

Figure 7: Typed FFT

In the past, core functions like `append` were implemented in the runtime system for performance reasons. These days, functions like `append` can be implemented in Racket and still have competitive performance with those implemented in C, partly because the Racket compiler has improved to the point that it generates reasonable code and partly because implementing the function in Racket avoids the cost of the context switch from JIT-generated code to the C-based runtime system.

At this point, if the authors of the FFT implementation also contributed to the runtime system, the natural choice would simply be to port `append` to Racket and adjust the existing runtime system to use the Racket version instead. To determine how beneficial this improvement would be to our FFT, we implement a basic version of `append` in Racket and measure the performance of our algorithm.[3]

[3] A general theme in the recent history of the Racket runtime system has been moving primitives from the C-based runtime system to Racket proper, and the existence of futures has encouraged this trend. While it is somewhat annoying that seemingly simple primitives like `append` have not yet been migrated, over 750 such primitives exist and, as we shall see, this program turns out not to be a concrete reason to migrate `append`.

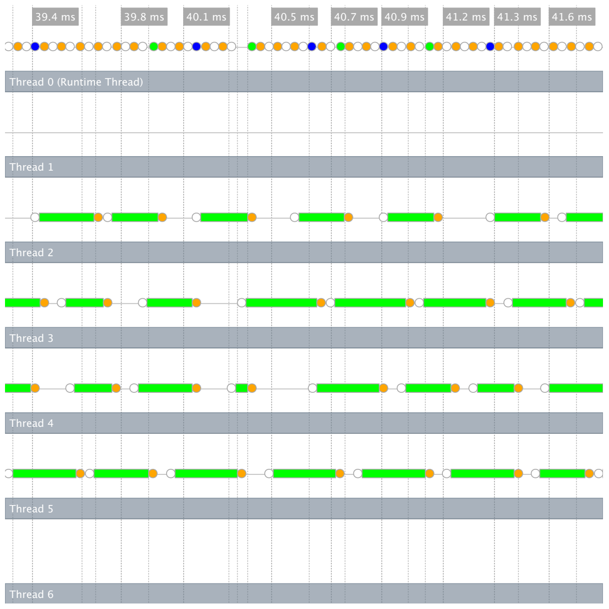


Figure 8: Future Timeline for Barricadeless FFT

3.5 Allocation

At this point, on 65536 element lists, the parallel FFT is about 2.4 times as fast as the original sequential one on our four-core machine. Unfortunately, however, this speedup is not due to parallelism. We can see this by comparing the wall-clock time and the cpu time that Racket reports for a call to `fft3`. For the 65536 element list case, Racket reports only about 20% more cpu time than real time.

Looking more closely at the trace reveals why. Specifically, consider figure 8. It shows a section of the timeline of `fft3`. As you can see, there are four futures running in parallel in this section of the trace. Each little green region (where parallel work occurs) is preceded by an orange circle and followed by a white one. Mousing over them reveals that they are synchronization points where the future is allocating additional memory. So, each future is only able to do a little bit of work before it must again synchronize to allocate more memory. This is the current bottleneck that stops us from getting parallel speedup.

Looking at `fft3` with allocation in mind, it is easy to see that it creates several intermediate lists that are not really necessary. One easy list to eliminate is the one created by `twiddle-factor`. Instead of creating a new list that holds the adjusted values, we can adjust the values in the loops at the end of our FFT algorithm, when the elements of the list are used. Figure 9 shows `fft4`, which is the result of performing this list fusion.

Timing `fft4` reveals a 20% speedup as compared to `fft3`, but still no significant parallelism. The visualizer confirms that allocation-induced synchronization is still to blame.

3.6 In-place FFT

To rectify this problem, we must adjust our FFT implementation to be in-place. To do so, we first copy the list into an array, perform the in-place FFT, and then extract the elements of the array, returning them as a list. To further reduce allocation, the algorithm keeps separate arrays for the real and imaginary components, avoiding allocation due to complex number boxing in the process. Figure 12 and figure 13 show the implementation in Racket.

```
(: fft4/depth ((Listof Float-Complex)
               Integer
               ->
               (Listof Float-Complex)))
(define (fft4/depth as d)
  (cond
    [(= (length as) 1) as]
    [(= d 0) (fft4/seq as)]
    [else
     (define-values (ss ts)
       (decimate-in-time as))
     (define bs
       (future
        (λ ()
          (fft4/depth ss (- d 1)))))
     (define cs
       (future
        (λ ()
          (fft4/depth ts (- d 1)))))
     (define n/2 (->fl (length (touch cs))))
     (define-values (l r)
       (for/lists:
         ([l : (Listof Float-Complex)]
          [r : (Listof Float-Complex)])
         ([b (in-list (touch bs))]
          [c (in-list (touch cs))]
          [k (in-naturals)])
         (define twiddle-c
           (* c (exp (/ (* pi 0.0+1.0i
                           (->fl k))
                        n/2))))
         (values (+ b twiddle-c)
                 (- b twiddle-c))))
     (append l r)]))
```

Figure 9: Fused FFT

While this code has certainly taken a turn for the worse in its low-level nature, and it does look like C with a few extra parentheses, the programming model is decidedly still not C's: errors are still caught as soon as they happen, and there is no explicit memory management required.

Ultimately, we were led here, informed by what we learned from the visualizer about the current implementation of Racket. And indeed, figure 10 shows that we are now getting good parallelism with this version. Specifically, the large contiguous green bars indicate that significant work is occurring in parallel.

3.7 How Well Did we Do?

Now that we've improved the parallel performance on our small machine, let us see how well this performs on a more significant machine and to compare it to an FFT implementation in C.

We used a machine with 4 2.1 GHz, 16 core, 64 bit AMD Opteron 6272s (for a total of 64 cores) running Red Hat Enterprise Linux Server release 6.2. The C code implements precisely the algorithm given in figure 12 and figure 13. It was compiled with gcc version 4.4.6, with optimization level `-O2`. Additionally, we called directly into the imperative portion of the Racket implementation, both to match the C code's behavior, but also to focus on timing the core of the algorithm, instead of the conversion from a list to a vector (and back).

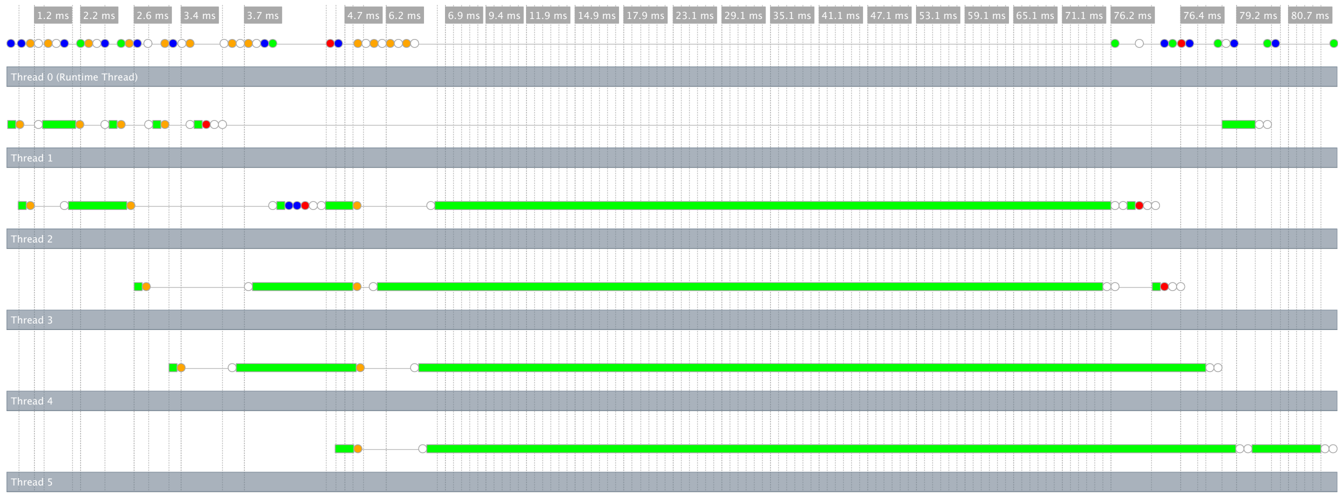


Figure 10: FFT 5's Complete Timeline with Input Size 131072

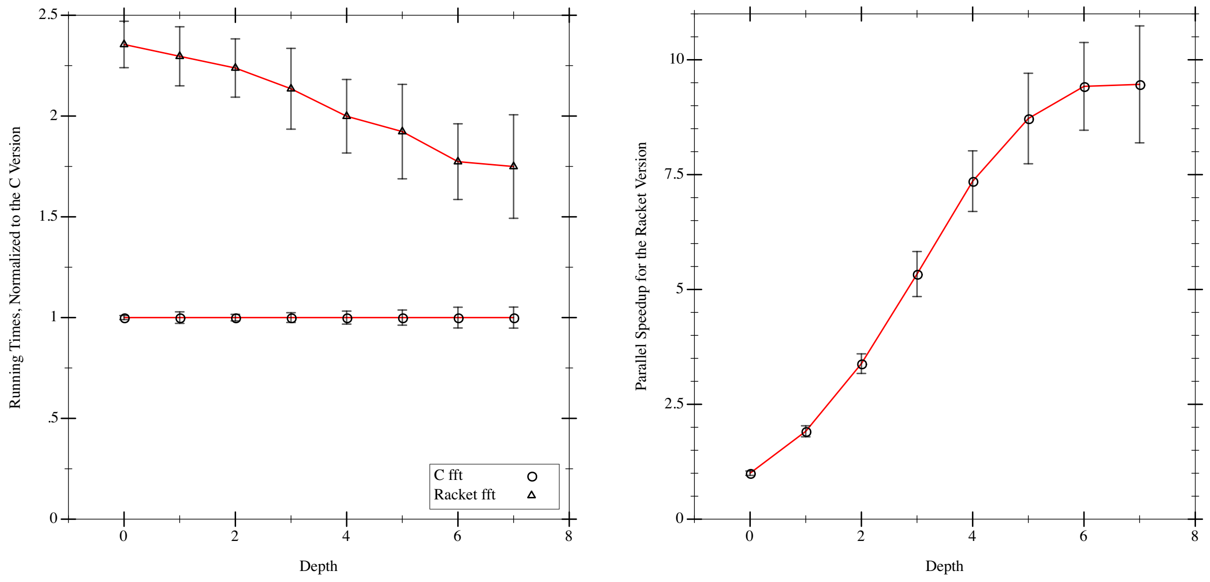


Figure 11: Comparison Between Racket and C and Parallel Speedup for Racket

These experiments were conducted with revision 55b11bf3 of the Racket git repository; details regarding setup are available here: `www.eecs.northwestern.edu/~jes947/fv/`.

Figure 11 shows the results. The graph on the left shows the Racket and C times at each depth, normalized to the C time. We measured times for depths up to one more than what `init-d` computes for our benchmarking machine. Each result is the average of 125 runs. The error bars extend one standard deviation in each direction. Generally speaking, Racket scales slightly better than the C code, but is about twice as slow. This scaling, however, is probably because the Racket version does more work than the C version in the first place.

The graph on the right of figure 11 shows the parallel speedup for Racket compared to sequential Racket. We used the same data and methodology as for the graph on the left. Note that the horizontal axis is depth, not number of cores used (the number of cores doubles at each new depth).

4. Extracting Traces

The visualizer extracts information about a program run via Racket's logging system. 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

```
(: fft5 ((Listof Float-Complex)
         ->
         (Listof Float-Complex)))
(define (fft5 as)
  (define as-r
    (apply flvector (map real-part as)))
  (define as-i
    (apply flvector (map imag-part as)))
  (define n (flvector-length as-r))
  (define xs-r (make-flvector n 0.0))
  (define xs-i (make-flvector n 0.0))
  (define d (init-d))
  (fft5/depth as-r as-i xs-r xs-i n 0 d)
  (for/list ([r (in-flvector as-r)]
             [i (in-flvector as-i)])
    (make-rectangular r i)))

(: decimate-in-time
   (FlVector FlVector
    FlVector FlVector
    Integer Integer
    -> Void))
(define (decimate-in-time as-r as-i
                          xs-r xs-i
                          n/2 start)
  (for ([i (in-range n/2)])
    (define si (+ start i))
    (define si2 (+ si i))
    (define si21 (+ si2 1))
    (define sin2 (+ si n/2))
    (flvector-set!
     xs-r si (flvector-ref as-r si2))
    (flvector-set!
     xs-i si (flvector-ref as-i si2))
    (flvector-set!
     xs-r sin2 (flvector-ref as-r si21))
    (flvector-set!
     xs-i sin2 (flvector-ref as-i si21))))

(: twiddle-factor
   (FlVector FlVector
    Integer Integer -> Void))
(define (twiddle-factor cs-r cs-i
                        n/2 start)
  (define c (/ (* pi 0.0+1.0i) (->fl n/2)))
  (for ([k (in-range n/2)])
    (define k-start (+ k start))
    (define res
      (* (make-rectangular
          (flvector-ref cs-r k-start)
          (flvector-ref cs-i k-start))
         (exp (* c (->fl k)))))
    (flvector-set! cs-r k-start
                   (real-part res))
    (flvector-set! cs-i k-start
                   (imag-part res))))
```

Figure 12: In-place FFT, part i

```
(: fft5/depth
   (FlVector FlVector FlVector FlVector
    Integer Integer Integer
    -> Void))
(define (fft5/depth as-r as-i xs-r xs-i
                    n start d)
  (unless (= n 1)
    (define n/2 (quotient n 2))
    (decimate-in-time as-r as-i xs-r
                      xs-i n/2 start)
    (cond
      [(= d 0)
       (fft5/depth xs-r xs-i as-r as-i
                   n/2 start 0)
       (fft5/depth xs-r xs-i as-r as-i
                   n/2 (+ start n/2) 0)
       (twiddle-factor xs-r xs-i n/2
                       (+ start n/2))]
      [else
       (define bs
         (future
          (λ ()
            (fft5/depth xs-r xs-i as-r
                        as-i n/2 start
                        (- d 1)))))
       (define cs
         (future
          (λ ()
            (fft5/depth xs-r xs-i as-r
                        as-i n/2
                        (+ start n/2) (- d 1))
            (twiddle-factor xs-r xs-i n/2
                            (+ start n/2)))))
       (touch bs)
       (touch cs)])
    (for ([k (in-range n/2)])
      (define sk (+ start k))
      (define sk2 (+ sk n/2))
      (define br (flvector-ref xs-r sk))
      (define bi (flvector-ref xs-i sk))
      (define cr (flvector-ref xs-r sk2))
      (define ci (flvector-ref xs-i sk2))
      (flvector-set! as-r sk2 (- br cr))
      (flvector-set! as-i sk2 (- bi ci))
      (flvector-set! as-r sk (+ br cr))
      (flvector-set! as-i sk (+ bi ci)))))
```

Figure 13: In-place FFT, part ii

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.

5. Related Work

Futures are one of two forms of parallelism available in Racket, the other being *places* (Tew et al. 2011). The two approaches share the common problem of coping with Racket’s large, legacy runtime system. Unlike futures, places circumvent this problem by cloning most of the runtime system state and thus allow communication between parallel tasks only via explicit message passing. This approach has the advantage that any Racket program can be made to run in parallel (unlike futures, where programs must avoid barricades), but with the disadvantages that creating a place is more expensive and in most cases communication requires copying data.

Futures are based on similar constructs in Multilisp (Halstead 1985), 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.

Threadscope (Jones Jr. et al. 2009) 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 (2007) 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 (1993) 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 (Zaki et al. 1999) 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 (Knupfer et al. 2008) 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 (Knupfer et al. 2006), 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. (2000) 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 (He et al. 2010) 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 (Bacon et al. 2007) 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.

Acknowledgments Thanks to Peter Dinda for letting us use his 64 core machine for the results in section 3.7. Thanks to Neil Toronto for his input regarding the timing analysis. Thanks to Sam Tobin-Hochstadt and the anonymous FHPC 2012 reviewers for their feedback on the paper. This work was partially supported by the NSF.

Bibliography

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

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.

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

Robert H. Halstead Jr. A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 1985.

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.

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

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.

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.

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.

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

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the Numeric Tower. In *Proc. Intl. Symp. Practical Aspects of Declarative Languages*, 2012.

James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. Back to the Futures: Incremental Parallelization of Existing Sequential Runtimes. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2010.

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.

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

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