Distributed Places

Kevin Tew
Brigham Young University
tew@byu.edu

James Swaine
Northwestern University
JamesSwaine2010@u.northwestern.edu

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

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

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Abstract

Distributed Places bring new support for distributed, message-passing parallelism to Racket. This paper gives an overview of the programming model and how we had to modify our existing, runtime-system to support distributed places. We show that the freedom to design the programming model helped us to make the implementation tractable. The paper presents an evaluation of the design, examples of higher-level API's that can be built on top of distributed places, and performance results of standard parallel benchmarks.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors — Run-time environments

General Terms Parallelism, Languages, Design

1. Introduction

Dynamic, functional languages are important as rapid development platforms for solving everyday problems and completing tasks. As programmers embrace parallelism in dynamic programming languages, the need arises to extend multi-core parallelism to multi-node parallelism. Distributed places delivers multi-node parallelism to Racket by building on top of the existing places [18] infrastructure.

The right extensions to dynamic, functional languages enable the introduction of a hierarchy of parallel programming abstractions. Language extension allows these parallel programming abstractions to be concisely mapped to different hardware such as a shared memory node or a distributed memory machine. Distributed places are not an add-on library or a foreign function interface (FFI). Instead, Racket's places and distributed places are language extensions on which higher-level distributed programming frameworks can easily be expressed. An RPC mechanism, map reduce, MPI, and nested-data parallelism are all concisely and easily built on top of distributed places. These higher-level frameworks meld with the Racket language to create extended languages, which describe different types of distributed programming.

The distributed places API allows the user to spawn new execution contexts on remote machines. Distributed places reuse the communication channel API for intra-process parallelism to build a transparent distributed communication system over a underlying sockets layer. Racket's channels for parallel and distributed communication are first-class Racket events. These channels can be waited on concurrently with other Racket event objects such as file ports, sockets, threads, channels, etc. Together, Racket's intra-process and distributed parallelism constructs form a foundation capable of supporting higher-level parallel frameworks.

2. Design

Programming with parallelism should avoid the typical interference problems of threads executing in a single address space. Instead, parallel executions contexts should execute in isolation. Communication between execution contexts should use message-passing instead of shared-memory in a common address space. This isolated, message-passing approach positions the programmer to think about the data-placement and communication needs of a parallel program to enable sustained scalability. Distributed places extend our existing implementation of isolated, message-passing parallelism which, until now, was limited to a single node. As a program moves from multi-core parallelism to multi-node parallelism latency increases and bandwidth decreases; data-placement and communication patterns become even more crucial.

Much of a distributed programming API is littered with system administration tasks that impede programmers from focusing on programming and solving problems. First, programmers have to authenticate and launch their programs on each node in the distributed system. Then they have to establish communication links between the nodes in the system, before they can begin working on the problem itself. The work of the distributed places framework is to provide support for handling the problems of program launch and communication link establishment.

Racket's distributed places API design is centered around machine nodes that do computation in places. The user/programmer configures a new distributed system using declarative syntax and callbacks. By specifying a hostname and port number, a programmer can launch a new place on a remote host. In the simplest distributed-places programs, hostnames and port numbers are hardwired. When programmers need more control, distributed places permits complete programmatic configuration of node launch and communication link parameters.

```
1  #lang racket/base
2  (require racket/place
3           racket/place/distributed)
4
5  (provide hello-world)
6
7  (define (hello-world ch)
8    (printf/f "hello-world received: ~a\n"
9              (place-channel-get ch))
10   (place-channel-put ch "Hello World\n")
11   (printf/f "hello-world sent: Hello World\n"))
12
13 (module+ main
14   (define p (dynamic-place
15               (quote-module-path "..")
16               'hello-world))
17
18   (place-channel-put p "Hello")
19   (printf/f "main received: ~a\n"
20             (place-channel-get p))
21   (place-wait p))
```

Figure 1: Place's Hello World

The hello world example in figure 1 demonstrates the key components of a places program. Appearing first, the `hello-world` procedure is called to create hello-world places. The `main` module follows and contains the code to construct and communicate with a `hello-world` place.

Looking closer at the `main` module, the `hello-world` place is created using `dynamic-place`.

```
(dynamic-place module-path start-
proc) → place?
  module-path : module-path?
  start-proc : symbol?
```

The `dynamic-place` procedure creates a place to run the procedure that is identified by *`module-path`* and *`start-proc`*. The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor is also a place channel to initiate communication between the new place and the creating place.

The module indicated by *`module-path`* must export a function with the name *`start-proc`*. The exported function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place channel that is returned by `dynamic-place`.

The `(quote-module-path "..")` and `'hello-world` arguments on lines 17 and 18 of figure 2 specify the procedure address of the new place to be launched. In this example, the `(quote-module-path "..")` argument provides the module path to the parent module of `main`, where the `'hello-world` procedure is located.

Places communicate over place channels which allow structured data communication between places. Supported structured data includes booleans, numbers, characters, symbols, byte strings, Unicode strings, filesystem paths, pairs, lists, vectors, and "prefab" structures (i.e., structures that are transparent and whose types are universally named). Place channels themselves can be sent in messages across place channels, so that communication is not limited to the creator of a place and its children places; by sending place channels as messages, a program can construct custom message topologies.

```
13 (module+ main
14   (define n (create-place-node
15               "host2"
16               #:listen-port 6344))
17   (define p (dynamic-place
18               #:at n
19               (quote-module-path "..")
20               'hello-world))
21   ...)
```

Figure 2: Distributed Hello World

```
(place-channel-put ch v) → void?
  ch : place-channel?
  v : place-message-allowed?
(place-channel-get ch) → place-message-
allowed?
  ch : place-channel?
```

The `place-channel-put` function asynchronously sends a message *`v`* on channel *`ch`* and returns immediately. The `place-channel-get` function waits until a message is available from the place channel *`ch`*.

```
(place-wait p) → void?
  p : place?
```

Finally the `place-wait` procedure blocks until *`p`* terminates.

The distributed hello world example in figure 2 shows the two differences between a simple places program and a simple distributed places program. The `create-place-node` procedure uses `ssh` to start a new remote node on `host2` and assumes that ssh is configured correctly. Upon launch, the remote node listens on port `6344` for incoming connections. Once the remote node is launched, a TCP connection to port `6344` on the new node is established. The `create-place-node` returns a node descriptor object, `n`, which allows for administration of the remote node. The remote place is created using `dynamic-place`. The new `#:at` keyword argument specifies the node on which to launch the new place.

Remotely spawned places are private. Only the node that spawned the place can communicate with it through its descriptor object. Named places allow programmers to make a distributed place publicly accessible. Named places are labeled with a name when they are created.

```
(define p (dynamic-place
            #:at n
            #:named 'helloworld1
            (quote-module-path "..")
            'hello-world))
```

Any node can connect to a named place by specifying the destination node and name to connect to. In this example, `node` is a node descriptor object returned from `create-place-node`.

```
(connect-to-named-place node 'helloworld1)
```

3. Higher Level APIs

The distributed places implementation is a foundation that can support a variety of higher-level APIs and parallel processing frameworks such as Remote Procedure Calls (RPC), Message Passing Interface (MPI) [13], MapReduce [4], and Nested Data Parallelism [2]. All of these higher-level APIs and frameworks can be built on top of named places.

```
#lang racket/base
(require racket/place/distributed
         racket/class
         racket/place
         racket/runtime-path
         "tuple.rkt")
(define-runtime-path tuple-path "tuple.rkt")

(module+ main
  (define remote-node (create-place-node
                       "host2"
                       #:listen-port 6344))
  (define tuple-place
    (dynamic-place
      #:at remote-node
      #:named 'tuple-server
      tuple-path
      'make-tuple-server))

  (define c (connect-to-named-place
             remote-node
             'tuple-server))
  (define d (connect-to-named-place
             remote-node
             'tuple-server))
  (tuple-server-hello c)
  (tuple-server-hello d)
  (displayln
    (tuple-server-set c "user0" 100))
  (displayln
    (tuple-server-set d "user2" 200))
  (displayln (tuple-server-get c "user0"))
  (displayln (tuple-server-get d "user2"))
  (displayln (tuple-server-get d "user0"))
  (displayln (tuple-server-get c "user2")))
```

Figure 3: Tuple RPC Example

3.1 RPC via Named Places

Named places make a place's interface public at a well-known address: the host, port, and name of the place. They provide distributed places with a form of computation similar to the actor model [10]. Using named places and the `define-named-remote-server` form, programmers can build distributed places that act as remote procedure call (RPC) servers. The example in figure 3 demonstrates how to launch a remote Racket node instance, launch a remote procedure call (RPC) tuple server on the new remote node instance, and start a local event loop that interacts with the remote tuple server.

The `create-place-node` procedure in figure 3 connects to `"host2"` and starts a distributed place node there that listens on port 6344 for further instructions. The descriptor to the new distributed place node is assigned to the `remote-node` variable. Next, the `dynamic-place` procedure creates a new named place on the `remote-node`. The named place will be identified in the future by its name symbol `'tuple-server`.

The code in figure 4 contains the use of the `define-named-remote-server` form, which defines a RPC server suitable for invocation by `dynamic-place`. The RPC `tuple-server` allows for named tuples to be stored into a server-side hash table and later retrieved. It also demonstrates one-way "cast" procedures, such as `hello`, that do not return a value to the remote caller.

For the purpose of explaining the `tuple-server` implementation, figure 5 shows the macro expansion of the RPC tuple server. Typical users of distributed places do not need to understand the

```
#lang racket/base
(require racket/match
         racket/place/define-remote-server)

(define-named-remote-server tuple-server

  (define-state h (make-hash))
  (define-rpc (set k v)
    (hash-set! h k v)
    v)
  (define-rpc (get k)
    (hash-ref h k #f))
  (define-cast (hello)
    (printf "Hello from define-cast\n")
    (flush-output)))
```

Figure 4: Tuple Server

expanded code to use the `define-named-remote-server` macro. The `define-named-remote-server` form, in figure 5, takes an identifier and a list of custom expressions as its arguments. A place function is created by prepending the `make-` prefix to the identifier `tuple-server`. The `make-tuple-server` identifier is the symbol given to the `dynamic-place` form in figure 3. The `define-state` custom form translates into a simple `define` form, which is closed over by the `define-rpc` forms.

The `define-rpc` form is expanded into two parts. The first part is the client stubs that call the RPC functions. The stubs can be seen at the top of figure 5. The client function name is formed by concatenating the `define-named-remote-server` identifier, `tuple-server`, with the RPC function name, `set`, to form `tuple-server-set`. The RPC client functions take a destination argument which is a `remote-connection%` descriptor followed by the RPC function's arguments. The RPC client function sends the RPC function name, `set`, and the RPC arguments to the destination by calling an internal function `named-place-channel-put`. The RPC client then calls `named-place-channel-get` to wait for the RPC response.

The second part of the expansion part of `define-rpc` is the server implementation of the RPC call. The server is implemented by a match expression inside the `make-tuple-server` function. Messages to named places are placed as the first element of a list where the second element is the source or return channel to respond on. For example, in `(list (list 'set k v) src)` the inner list is the message while `src` is the place-channel to send the reply on. The match clause for `tuple-server-set` matches on messages beginning with the `'set` symbol. The server executes the RPC call with the communicated arguments and sends the result back to the RPC client. The `define-cast` form is similar to the `define-rpc` form except there is no reply message from the server to client

The named place, shown in the tuple server example, follows an actor-like model by receiving messages, modifying state, and sending responses. Racket macros enables the easy construction of RPC functionality on top of named places.

3.2 Racket Message Passing Interface

RMPI is Racket's implementation of the basic MPI operations. A RMPI program begins with the invocation of the `rmpi-launch` procedure, which takes two arguments. The first is a hash from Racket keywords to values of default configuration options. The `rmpi-build-default-config` helper procedure takes a list of Racket keyword arguments and forms the hash of optional con-

```
(module named-place-expanded racket/base
   (require racket/place racket/match)
   (define/provide
    (tuple-server-set dest k v)
     (named-place-channel-put
       dest
       (list 'set k v))
     (named-place-channel-get dest))
   (define/provide
    (tuple-server-get dest k)
     (named-place-channel-put
       dest
       (list 'get k))
     (named-place-channel-get dest))
   (define/provide
    (tuple-server-hello dest)
     (named-place-channel-put
        dest
        (list 'hello)))
   (define/provide
    (make-tuple-server ch)
     (let ()
       (define h (make-hash))
       (let loop ()
         (define msg (place-channel-get ch))
          (match
           msg
           ((list (list 'set k v) src)
            (define result (let ()
                             (hash-set! h k v)
                             v))
            (place-channel-put src result)
            (loop))
           ((list (list 'get k) src)
            (define result
              (let ()
                (hash-ref h k #f)))
            (place-channel-put src result)
            (loop))
           ((list (list 'hello) src)
            (define result
              (let ()
                (printf
                  "Hello from define-cast\n")
                (flush-output)))
            (loop)))
         loop)))
   (void))
```

Figure 5: Macro Expansion of Tuple Server

figuration values. The second argument is a list of configurations, one for each node in the distributed system. A configuration is made up of a hostname, a port, a unique name, a numerical RMPI process id, and an optional hash of additional configuration options. An example of `rmpi-launch` follows.

```
(rmpi-launch
  (rmpi-build-default-config
    #:racket-path "/tmp/mplt/bin/racket"
    #:distributed-launch-path
      (build-distributed-launch-path
        "/tmp/mplt/collects")
    #:rmpi-module "/tmp/mplt/kmeans.rkt"
    #:rmpi-func 'kmeans-place
    #:rmpi-args
      (list "/tmp/mplt/color100.bin"
            #t 100 9 10 0.0000001))

  (list (list "n1.example.com" 6340 'kmeans_0 0)
        (list "n2.example.com" 6340 'kmeans_1 1)
        (list "n3.example.com" 6340 'kmeans_2 2)
        (list "n4.example.com" 6340 'kmeans_3 3
              (rmpi-build-default-config
                #:racket-path "/bin/racket"))))
```

The `rmpi-launch` procedure spawns the remote nodes first and then spawns the remote places named with the unique name from the config structure. After the nodes and places are spawned, `rmpi-launch` sends each spawned place its RMPI process id, the config information for establishing connections to the other RMPI processes, and the initial arguments for the RMPI program. The last function of `rmpi-launch` is to rendezvous with RMPI process 0 when it calls `rmpi-finish` at the end of the RMPI program.

The `rmpi-init` procedure is the first call that should occur inside the `#:rmpi-func` place procedure. The `rmpi-init` procedure takes one argument `ch`, which is the initial `place-channel` passed to the `#:rmpi-func` procedure. The `rmpi-init` procedure communicates with `rmpi-launch` over this channel to receive its RMPI process id and the initial arguments for the RMPI program.

```
(define (kmeans-place ch)
  (define-values (comm args tc) rmpi-init ch)
  ;;; kmeans rmpi computation ...
  (rmpi-finish comm tc))
```

The `rmpi-init` procedure has three return values: an opaque communication structure which is passed to other RMPI calls, the list of initial arguments to the RMPI program, and a typed channel wrapper for the initial place-channel it was given. The typed channel wrapper allows for the out of order reception of messages. Messages are lists and their type is the first item of the list which must be a racket symbol. A typed channel returns the first message received on the wrapped channel that has the type requested. Messages of other types that are received are queued for later requests.

The `rmpi-comm` structure, returned by `rmpi-init`, is the communicator descriptor used by all other RMPI procedures. The RMPI informational functions `rmpi-id` and `rmpi-cnt` return the current RMPI process id and the total count of RMPI processes respectively.

```
> (rmpi-id comm)
3


> (rmpi-cnt comm)
8
```

The `rmpi-send` and `rmpi-recv` procedures provide point-to-point communication between two RMPI processes.

```
> (rmpi-send comm dest-id '(msg-type1 "Hi"))
```

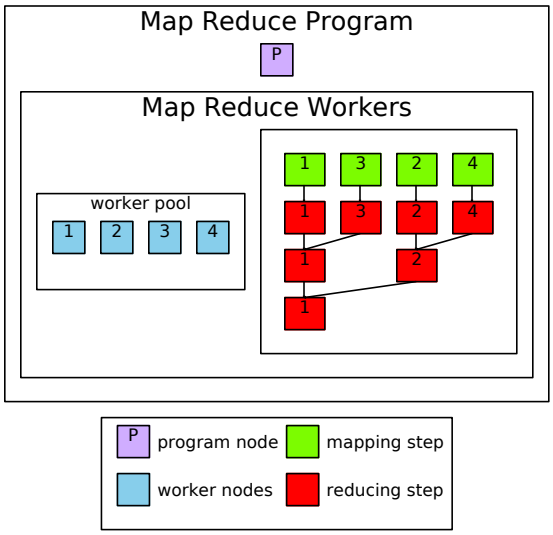

Figure 6: MapReduce Program

```
> (rmpi-recv comm src-id)
'(msg-type1 "Hi")
```

With the `rmpi-comm` structure the programmer can also use any of the RMPI collective procedures: `rmpi-broadcast`, `rmpi-reduce`, `rmpi-allreduce`, or `rmpi-barrier` to communicate values between the nodes in the RMPI system.

The `(rmpi-broadcast comm 1 (list 'a 12 "foo"))` expression broadcasts the list `(list 'a 12 "foo")` from RMPI process `1` to all the other RMPI processes in the `comm` communication group. Processes receiving the broadcast execute `(rmpi-broadcast comm 1)` without specifying the value to send. The `(rmpi-reduce comm 3 + 3.45)` expression does the opposite of broadcast by reducing the local value `3.45` and all the other procesess local values to RMPI process `3` using the `+` procedure to do the reduction. The `rmpi-allreduce` expression is similar to `rmpi-reduce` except that the final reduced value is broadcasted to all processes in the system after the reduction is complete. Synchronization among all the RMPI processes occurs through the use of the `(rmpi-barrier comm)` expression, which is implemented internally using a simple reduction followed by a broadcast.

Distributed places are simply computation resources connected by socket communications. This simple design matches MPI's model and makes RMPI's implementation very natural. The RMPI layer demonstrates how distributed places can provide the foundations of other distributed programming frameworks such as MPI.

3.3 Map Reduce

Our MapReduce implementation is patterned after the Hadoop [1] framework. Key value pairs are the core data structures that pass through the map and reduce stages of the computation. In the following example the number of word occurrences is counted across a list of text files. The files have been preprocessed so that there is only one word per line.

Figure 6 shows the different actors in the MapReduce paradigm. The program node `P` creates the MapReduce workers group. When a `map-reduce` call is made the program node serves as the controller of the worker group. It dispatches mapper tasks to each node and waits for them to respond as finished with the mapping task. Once a node has finished its mapping task it runs the reduce operation on its local data. Given two nodes in the reduced state, one node can reduce to the other; freeing one node to return to the worker pool for allocation to future tasks. Once all the nodes have reduced to a single node, the `map-reduce` call returns the final list of reduced key values.

The first step in using distributed place's MapReduce implementation is to create a list of worker nodes. This is done by calling the `make-map-reduce-workers` procedure with a list of hostnames and ports to launch nodes at.

```
(define config (list (list "host2" 6430)
                     (list "host3" 6430)))
(define workers (make-map-reduce-workers config))
```

Once a list of worker nodes has been spawned, the programmer can call `map-reduce` supplying the list of worker nodes, the config list, the procedure address of the mapper, the procedure address of the reducer, and a procedure address of an optional result output procedure. Procedure addresses are lists consisting of the quoted-module-path and the symbol name of the procedure being addressed.

```
(map-reduce
  workers
  config
  tasks
  (list (quote-module-path "..") 'mapper)
  (list (quote-module-path "..") 'reducer)
  #:outputer (list (quote-module-path "..")
                   'outputer))
```

Tasks can be any list of key value pairs. In this example the keys are the task numbers and the values are the input files the mappers should process.

```
(define tasks (list (list (cons 0 "/tmp/w0"))
                    (list (cons 1 "/tmp/w1"))
                    ...))
```

The mapper procedure takes a list of key value pairs as its argument and returns the result of the map operation as a new list of key value pairs. The input to the mapper, in this example, is a list of a single pair containing the task number and the text file to process, `(list (cons 1 "w0.txt"))`. The output of the mapper is a list of each word in the file paired with 1, its initial count. Repeated words in the text are repeated in the mappers output list. Reduction happens in the next step.

```
;;(->
;;  (listof (cons any any))
;;  (listof (cons any any)))
(define/provide (mapper kvs)
  (for/first ([kv kvs])
    (match kv
      [(cons k v)
       (with-input-from-file
         v
         (lambda ()
           (let loop ([result null])
             (define l (read-line))
             (if (eof-object? l)
                 result
                 (loop (cons (cons l 1)
                             result))))))])))
```

After a task has been mapped, the MapReduce framework sorts the output key value pairs by key. The framework also coalesces pairs of key values with the same key into a single pair of the key and the list of values. As an example, the framework transforms the output of the mapper `'(("house" 1) ("car"`

1) ("house" 1)) into '(("car" (1)) ("house" (1 1)))

The reducer procedure takes, as input, this list of pairs, where each pair consists of a key and a list of values. For each key, the reducer reduces the list of values to a list of a single value. In the word count example, An input pair, (cons "house" '(1 1 1)) will be transformed to (cons "house" '(4)) by the reduction step.

```
;;(->
;; (listof (cons any (listof any)))
;; (listof (cons any (listof any))))
(define/provide (reducer kvs)
  (for/list ([kv kvs])
    (match kv
      [(cons k v)
       (cons k (list (for/fold ([sum 0])
                               ([x v])
                       (+ sum x))))])))
```

Once each mapped task has been reduced the outputs of the reduce steps are further reduced until a single list of word counts remains. Finally an optional output procedure is called which prints out a list of words and their occurrence count and returns the total count of all words.

```
(define/provide (outputer kvs)
  (displayln
    (for/fold ([sum 0]) ([kv kvs])
      (printf "~a - ~a\n" (car kv) (cadr kv))
      (+ sum (cadr kv)))))
```

3.4 Nested Data Parallelism

The last parallel processing paradigm implemented on top of distributed places is nested data parallelism [9]. In this paradigm recursive procedure calls create subproblems that can be parallelized. An implementation of parallel quicksort demonstrates nested data parallelism built on top of distributed places.

The distributed places, nested data parallelism API – `ndp-get-node`, `ndp-sendwork`, `ndp-get-result`, and `ndp-return-node` – is built on top of the RMPI layer. The main program node, depicted as `P` in figure 7, creates the `ndp-group`. The `ndp-group` consists of a coordinating node, `0`, and a pool of worker nodes `1, 2, 3, 4`. The coordinating node receives a sort request from `ndp-sort` and forwards the request to the first available worker node, node `1`. Node `1` divides the input list in half and requests a new node from the coordinator to process the second half of the input. The yellow bars on the right side of figure 7 show the progression as the sort input is subdivided and new nodes are requested from the coordinator node. Once the sort is complete, the result is returned to the coordinator node, which returns the result to the calling program `P`.

Like the previous two examples, the nested data parallel quicksort example begins by spawning a group of worker processes.

```
(define config
  (list (list "host2" 6340)
        (list "host3" 6340)
        (list "host4" 6340)
        (list "host5" 6340)
        (list "host6" 6340)))

(define ndp-group (make-ndp-group config))
```

Next the sort is performed by calling `ndp-qsort`.

```
(displayln (ndp-qsort (list 9 1 2 8 3 7 4 6 5 10)
                      ndp-config))
```

The `ndp-qsort` procedure is a stub that sends the procedure address for the `ndp-parallel-qsort` procedure and the list

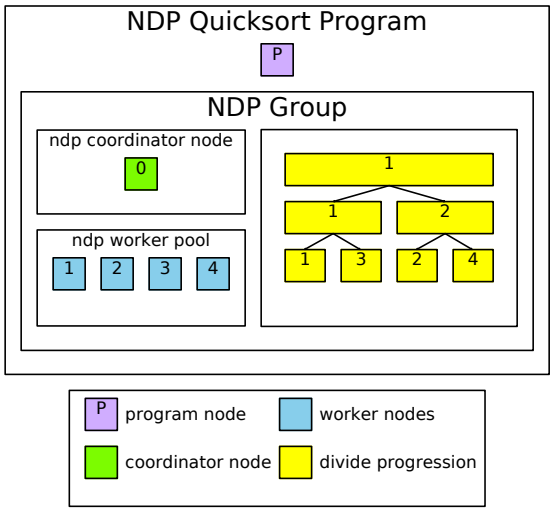

Figure 7: NDP Program

to sort to the `ndp-group`. The work of the parallel sort occurs in the `ndp-parallel-sort` procedure in figure 8. First the `partit` procedure picks a pivot and partitions the input list into three segments: less than the pivot, equal to the pivot, and greater than the pivot. If a worker node can be obtained from the `ndp-group` by calling `ndp-get-node`, the `gt` partition is sent to the newly obtained worker node to be recursively sorted. If all the worker nodes are taken, the `gt` partition is sorted locally using the `ndp-serial-qsort` procedure. Once the `lt` partition is sorted recursively on the current node, the `gt-part` is check to see if it was computed locally or dispatched to a remote node. If the part was dispatched to a remote node, its results are retrieved from the remote node by calling `ndp-get-result`. After the results are obtained the remote node `node` can be returned to the `ndp-group` for later use. Finally the sorted parts are appended to form the final sorted list result.

4. Implementation

A key part of the distributed place implementation is that distributed places is a layer over places, and parts of the places layer are exposed through the distributed places layer. In particular, each node, in figure 9, begins life with one initial place, the message router. The message router listens on a TCP port for incoming connections from other nodes in the distributed system. The message router serves two primary purposes: it multiplexes place messages and events on TCP connections between nodes and it services remote spawn requests for new places.

There are a variety of distributed places commands which spawn remote nodes and places. These command procedures return descriptor objects for the nodes and places they create. The descriptor objects allow commands and messages to be communicated to the remote controlled objects. In Figure 10, when node A spawns a new node B, A is given a `remote-node%` object with which to control B. Consequently, B is created with a `node%` object that is connected to A's `remote-node%` descriptor via a TCP socket connection. B's `node%` object is the message router for the new node B. A can then use its `remote-node%` descriptor to spawn a new place on node B. Upon successful spawning of the new place on B, A is returned a `remote-place%` descriptor object. On node B, a `place%` object representing the

```
(define (ndp-parallel-qsort l ndp-group)
  (cond
    [(< (length l) 2) l]
    [else
      (define-values (lt eq gt) (partit l))

      ;; spawn off gt partition
      (define gt-ref
         (define node (ndp-get-node ndp-group))
         (cond
           [node
             (cons #t (ndp-send-work
                        ndp-group
                        node
                        (list
                          (quote-module-path)
                          'ndp-parallel-qsort)
                        gt))]
           [else
             (cons #f (ndp-serial-qsort gt))]))

      ;; compute lt partition locally
      (define lt-part
        (ndp-parallel-qsort lt ndp-group))

      ;; retrieve remote results
      (define gt-part
        (match gt-ref
          [(cons #t node-id)
            (begin0
              (ndp-get-result ndp-group node-id)
              (ndp-return-node
                ndp-group
                node-id))]
          [(cons #f part) part]))

      (append lt-part eq gt-part)]))
```

Figure 8: NDP Parallel Sort

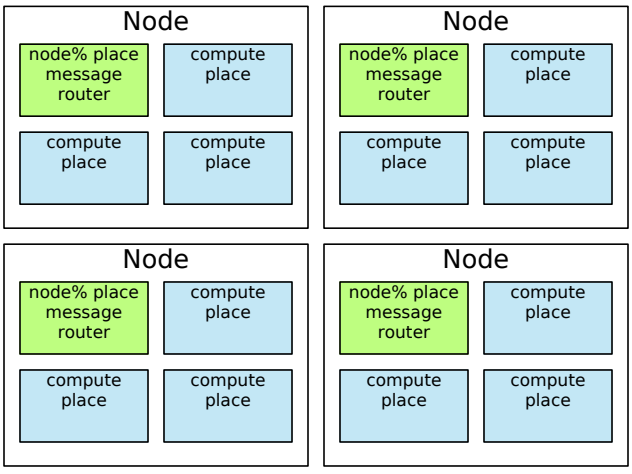

Figure 9: Distributed Places Nodes

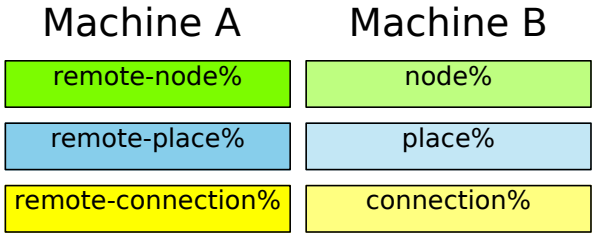

Figure 10: Descriptor (Controller) - Controlled Pairs

newly spawned place is attached to B's `node%` message-router. The `remote-connection%` descriptor object represents a connection to a named place. At the remote node, B, a `connection%` object intermediates between the `remote-connection%` and its destination named-place.

To communicate with remote nodes, a place message must be serializable. As a message-passing implementation, places send a copy of the original message when communicating with other places. Thus the content of a place message is inherently serializable and transportable between nodes of a distributed system.

To make place channels distributed, `place-socket-bridge%` proxies need to be created under the hood. The `place-socket-bridge%`s listen on local place channels and forward place messages over TCP sockets to remote place channels. Each node in a Racket distributed system must either explicitly pump distributed messages by registering each proxy with `sync` or bulk register the proxies, via the `remote-node%` descriptor, with a message router which can handle the pumping in a background thread.

Figure 11 shows the layout of the internal objects in a simple three node distributed system. The node at the top of the figure is the original node spawned by the user. Early in the instantiation of the top node, two additional nodes are spawned, node 1 and node 2. Then two places are spawned on each of node 1 and node 2. The instantiation code of the top node ends with a call to the `message-router` form. The `message-router` contains the `remote-node%` instances and the `after-seconds` and `every-seconds` event responders. Event responders execute when specific events occur, such as a timer event, or when messages arrive from remote nodes. The message router de-multiplexes events and place messages from remote nodes and dispatches them to the correct event responder.

Finally, function overloading is used to allow `place-` functions, such as `place-channel-get`, `place-channel-put`, and `place-wait`, to operate transparently on both place and distributed place instances. To accomplish this, distributed place descriptor objects are tagged as implementing the `place<%>` interface using a Racket structure property. Then `place-` functions dynamically dispatch to the distributed place version of the function for distributed place instances or execute the original function body for place instances.

5. Distributed Places Performance

Two of the NAS Parallel Benchmarks, IS and CG, are used to test the performance of the Racket distributed places implementation. The Fortran/C MPI version of the benchmarks were ported to Racket's distributed places. Performance testing occurred on 8 quad-core Intel i7 920 machines. Each machine was equipped with at least 4 gigabytes of memory and a 1 gigabit Ethernet connection.

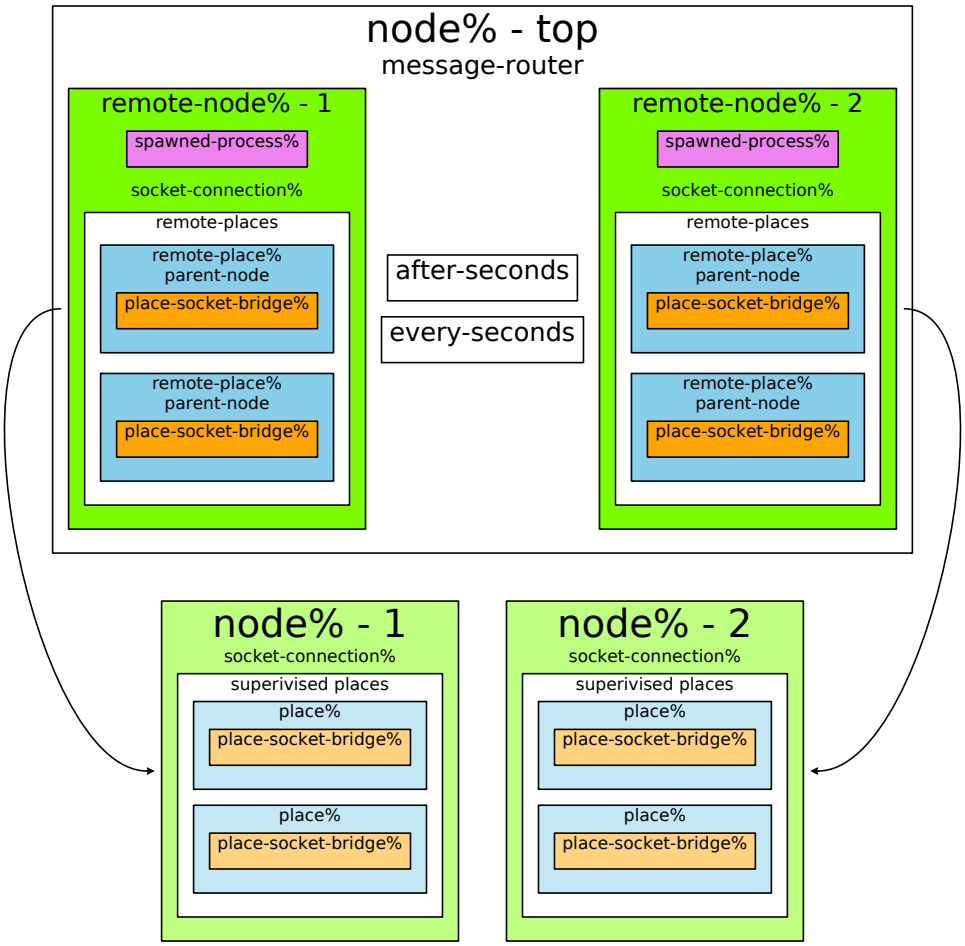

Figure 11: Three Node Distributed System

Performance numbers are reported for both Racket and Fortran/C versions of the benchmarks in figure 12. Racket's computational times scaled appropriately as additional nodes were added to the distributed system. Computational times are broken out and graphed in isolation to make computational scaling easier to see.

Racket communication times were larger than expected. There are several factors, stacked on top of one another, that explain the large communication numbers. First, five copies of the message occur during transit from source to destination. In a typical operation, a segment of a large flonum vector needs to be copied to a destination distributed place. The segment is copied out of the large flonum vector into a new flonum vector message. The message vector's length is the length of the segment to be sent. Next, the newly constructed vector message is copied over a place channel from the computational place to the main thread which serializes the message out a TCP socket to its destination. When the message arrives at its destination node the message is deserialized and copied a fourth time over a place channel to the destination computational place. Finally the elements of the message vector are copied into the mutable destination vector.

Racket's MPI implementation, RMPI, is not as sophisticated as the standard MPICH [14] implementation. MPICH has nonblocking sends and receives that allow messages to flow both directions simultaneously. Both the NAS Parallel Benchmarks used, IS and CG, use non-blocking MPI receives. RMPI on the other hand, always follows the typical protocol design of sending data in one direction and then receiving data from the opposite direction.

The largest contributor to Racket's excessive communication times is the serialization costs of the Racket primitive `write`. On Linux, serialization times are two orders of magnitude larger than the time to write raw buffers. One solution would be to replace distributed place's communication subsystem with FFI calls to an external MPI library. This solution would bypass the expensive `write` calls currently used in distributed places. Another viable solution would be to recognize messages that are vectors of flonums and use a restricted-form of write that could write flonum vectors as efficiently as raw buffers. Finally, it should be noted that using Racket's `write` is advantageous in cases where the message to be sent is a complex object graph instead of a simple raw buffer.

6. Related Work

Erlang [16] Erlang's distributed capabilities are built upon its process concurrency model. Remote Erlang nodes are identified by `name@host` identifiers. New Erlang processes can be started using the `slave:start` procedure or at the command line. Erlang uses a feature called links to implement fault notification. Two pro-

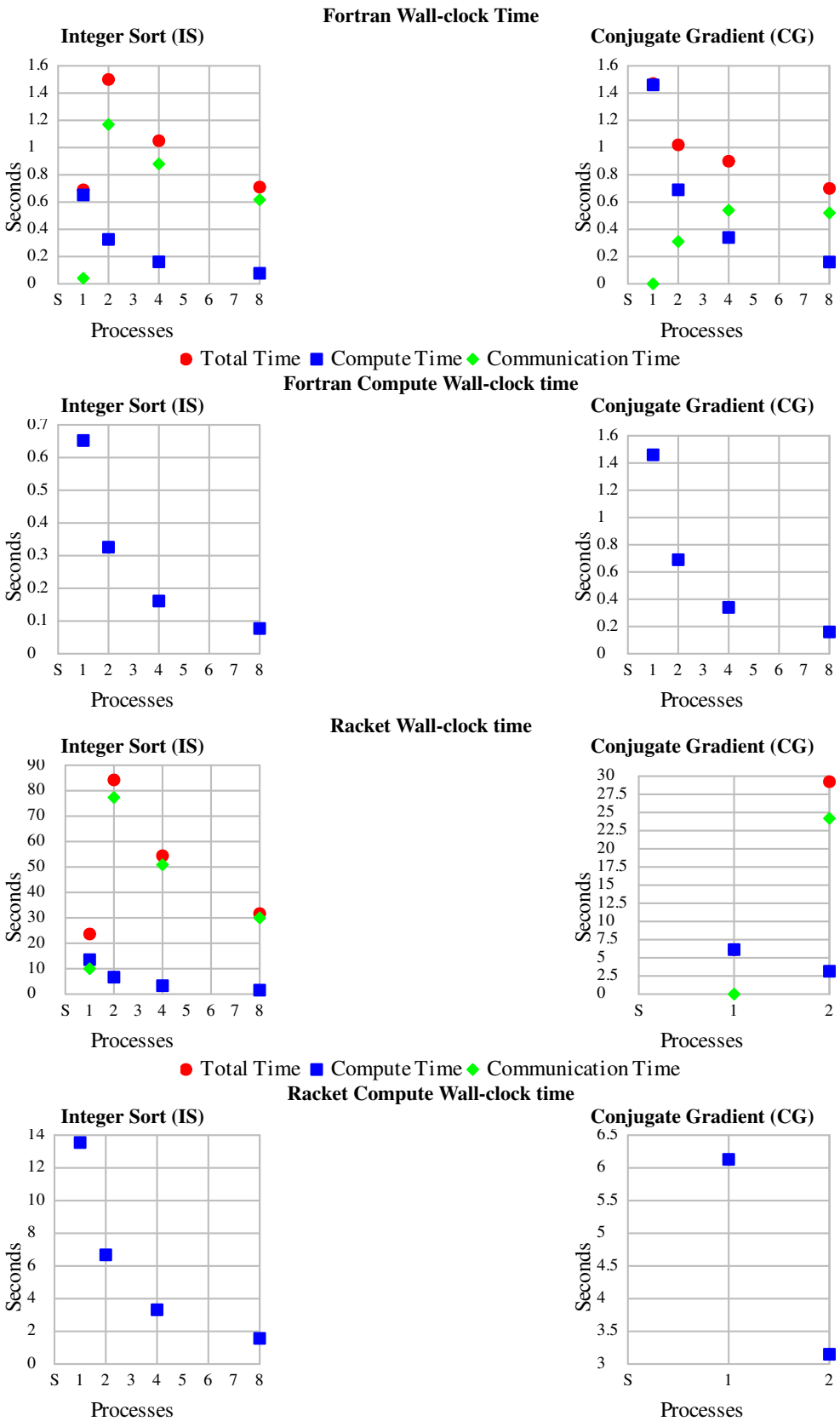

Figure 12: IS, CG, and MG class A results

cesses establish a link between themselves. Links are bidirectional; if either process fails the other process dies also. Erlang also provides monitors which are unidirectional notifications of a process exiting. Distributed Places and Erlang share a lot of similar features. While Erlang's distributed processes are an extension of its process concurrency model, Distributed Places are an extension of Racket's places parallelism strategy. Erlang provides a distributed message passing capability that integrates transparently with its inter-process message passing capability. The Disco project implements map reduce on top of a Erlang core. User level Disco programs, however, are written in Python, not Erlang. In contrast, the implementation and user code of distributed places' map reduce are both expressed as Racket code. Erlang has a good foundation for building higher-level distributed computing frameworks, but instead Erlang programmers seem to build customized distributed solutions for each application.

MapReduce [4] is a specialized functional programming model, where tasks are automatically parallelized and distributed across a large cluster of commodity machines. MapReduce programmers supply a set of input files, a map function and a reduce function. The map function transforms input key/value pairs into a set of intermediate key/value pairs. The reduce function merges all intermediate values with the same key. The framework does all the rest of the work. Google's MapReduce implementation handles partitioning of the input data, scheduling tasks across distributed computers, restarting tasks due to node failure, and transporting intermediate results between compute nodes. The MapReduce model can be applied to problems such as word occurance counting, distributed grep, inverted index creation, and distributed sort.

[8] Termite is a distributed concurrent scheme built on top of Gambit-C Scheme. Direct mutation of variables and data structures is forbidden in Termite. Instead mutation is simulated using messages and suspended, lightweight processes. Lookup in Termite's global environment is a node relative operation and resolves to the value bound to the global variable on the current node. Termite supports process migration via serializable closures and continuations. Termite follows Erlang's style of failing hard and fast. Where Erlang has bidirectional links, Termite has directional links that communicate process failure from one process to another. Failure detection only occurs in one direction from the process being monitored to the monitoring process. Termite also has supervisors which like supervisors in Erlang, restart child processes which have failed. Distributed Places could benefit from Termites superior serialization support, where nearly all Termite VM objects are serializable.
Akka [19] is a concurrency and distributed processing framework for Scala and Java. Like Erlang, Akka is patterned after the Actor model. Akka supports Erlang like supervisors and monitors for failure and exit detection. Like Erlang, Akka leaves the creation of higher-level distributed frameworks to custom application developers.

Kali [3] is a distributed version of Scheme 48 that efficiently communicates procedures and continuations from one compute node to another. Kali's implementation lazily faults continuation frames across the network as they are needed. Kali's proxies are really just address space relative variables. Proxies are identified by a globally unique id. Sending a proxy involves sending only its globally unique id. Retrieving a proxies value returns the value for the current address space. Kali allow for retrieval of the proxy's source node and spawning of new computations at the proxy's source.

Distributed Functional Programming in Scheme (DFPS) [17] uses futures semantics to build a distributed programming platform. DFPS employs the Web Server collection's `serial-lambda` form to serialize closures between machines. Unlike Racket futures, DFPS' `touch` form blocks until remote execution of the future completes. DFPS has a distributed variable construct called a `dbox`. For consistency, a dbox should only be written to once or a reduction function for writes to the `dbox` should be provided. Once a `dbox` has be set, the DFPS implementation propagates the `dbox` value other nodes that reference the `dbox`,

Cloud Haskell [5, 6] is a distributed programming platform built in Haskell. Cloud Haskell has two layers of abstraction. The lowest layer is the process layer, which is a message-passing distributed programming API. Next comes the tasks layer which provides a framework for failure recovery and data locality. Communication of serialized closures requires explicit specification from the user of what parts of environment will be serialized and sent with the code object.

On top of its message-passing process layer, Cloud Haskell implements typed channels that allow only messages of a specific type to be sent down the channel. A Cloud Haskell channel has a SendPort and a ReceivePort. ReceivePorts are not serializable and cannot be shared, which simplifies routing. SendPorts, however, are serializable and can be sent to multiple processes, allowing many to one style communication.

High-level Distributed-Memory Parallel Haskell (HdpH) [12] builds upon Cloud Haskell's work by adding support for polymorphic closures and lazy work stealing. HdpH does not require a special language kernel or any modifications to the vanilla GHC runtime. It simply uses GHC's Concurrent Haskell as a systems language for building a distributed memory Haskell.

Dryad [11] is an infrastructure for writing coarse-grain data-parallel distributed programs on the Microsoft platform. Distributed programs are structured as a directed graph. Sequential programs are the graph vertices and one-way channels are the graph edges. Unlike Distributed Places, Dryad is not a programming language. Instead it provides a execution engine for running sequential programs on partitioned data at computational vertices. Although Dryad is not a parallel database, the relational algebra can be mapped on top of a Dryad distributed compute graph. Unlike distributed places which is language centric, Dryad is infrastructure piece, which doesn't extend the expressiveness of any particular programming language.

Jade [15] is a implicitly parallel language. Implemented as a extension to C, Jade is intended to exploit task-level concurrency. Like OpenMP, Jade consists of annotations that programmers add to their sequential code. Jade uses data access and task granularity annotations to automatically extract concurrency and parallelize the program. A Jade front end then compiles the annotated code and outputs C. Programs parallelized with Jade continue to execute deterministically after parallelization. Jade's data model can interact badly with the programs that write to disjoint portions of a single aggregate data structure. In contrast, Distributed Places is an explicitly parallel language where the programmer must explicitly spawn tasks and explicitly handle communication between tasks.

Dreme [7] is a distributed Scheme. All first-class language objects in Dreme are mobile in the network. Dreme describes the communication network between nodes using lexical scope and first class closures. Dreme has a network-wide distributed memory and a distributed garbage collector. By default, Dreme sends objects by reference across the network, which can lead to large quantities of hidden remote operations. In contrast, distributed places copies all objects sent across the network and leaves the programmer responsible for communication invocations and their associated costs.

7. Conclusion

Building distributed places as a language extension allows the compact and clean construction of higher level abstractions such as RPC, MPI, map reduce and nested data parallelism. Distributed

places programs are more compact and easier to write than traditional C MPI programs. A Racket MPI implementation of parallel k-means was written with distributed places using less than half the lines of code of the original C and MPI version. With distributed places, messages can be heterogeneous and serialization is handled automatically by the language.

In addition to distributed parallel computing, Racket has many features that make it a great coordination and control language. Rackets provides a rich FFI (foreign function interface) for invoking legacy C code. Racket also includes extensive process exec capabilities for launching external programs and communicating with them over standard IO pipes. Racket's FFI, process exec capabilities, and distributed places gives programmers a powerful distributed coordination and workflow language.

With distributed places, programmers can quickly develop parallel and distributed solutions to everyday problems. Developers can also build new distributed computing frameworks using distributed places as common foundation. The distributed extension of places augments the Racket programmer's toolbox and provides a road map for other language implementers to follow.

Bibliography

[1] Apache Software Foundation. Hadoop. , 2012. `http://hadoop.apache.org`

[2] Guy E. Blelloch. Programming Parallel Algorithms. Communications of the ACM, 1996.

[3] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 1995.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.

[5] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Haskell for the Cloud. In Proceedings of the 4th ACM symposium on Haskell (Haskell '11), 2011.

[6] Jeffrey Epstein. Functional programming for the data centre. MS thesis, University of Cambridge, 2011.

[7] Matthew Fuchs. Dreme: for Life in the Net. PhD dissertation, New York University, 1995.

[8] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency Oriented Programming in Termite Scheme. In *Proc. Scheme and Functional Programming*, 2006.

[9] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel lang. In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '93), 1993.

[10] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73), 1973.

[11] Michael Isard, Mihai Budiur, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. European Conference on Computer Systems (EuroSys), 2007.

[12] Patrick Maier, Phil Trinder, and Has-Wolgang Loidl. High-level Distributed-Memory Parallel Haskell in Haskell. Symposium on Implementation and Application of Functional Languages, 2011.

[13] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi2-report.pdf, 2003. `http://www.mpi-forum.org/docs/mpi2-report.pdf`

[14] MPICH. MPICH. http://www.mcs.anl.gov/mpich2, 2013.

[15] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems* 20(1), pp. 1–63, 1998.

[16] 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.

[17] Alex Schwendner. Distributed Functional Programming in Scheme. MS thesis, Massachusetts Institute of Technology, 2010. `http://groups.csail.mit.edu/commit/papers/2010/alexrs-meng-thesis.pdf`

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

[19] Typesafe Inc. Akka. http://akka.io, 2012.