pHluid: The Design of a Parallel Functional Language Implementation on Workstations

Cormac Flanagan
Rice University
Department of Computer Science
Houston, Texas 77251-1892, USA
cormac@cs.rice.edu

Rishiyur S. Nikhil
Digital Equipment Corp.
Cambridge Research Laboratory
One Kendall Square, Bldg. 700
Cambridge, Massachusetts 02139, USA
nikhil@crl.dec.com

Abstract

This paper describes the distributed memory implementation of a shared memory parallel functional language. The language is Id, an implicitly parallel, mostly functional language that is currently evolving into a dialect of Haskell. The target is a distributed memory machine, because we expect these to be the most widely available parallel platforms in the future. The difficult problem is to bridge the gap between the shared memory language model and the distributed memory machine model. The language model assumes that all data is uniformly accessible, whereas the machine has a severe memory hierarchy: a processor's access to remote memory (using explicit communication) is orders of magnitude slower than its access to local memory. Thus, avoiding communication is crucial for good performance. The Id language, and its general dataflow-inspired compilation to multithreaded code are described elsewhere. In this paper, we focus on our new parallel runtime system and its features for avoiding communication and for tolerating its latency when necessary: multithreading, scheduling and load balancing; the distributed heap model and distributed coherent cacheing, and parallel garbage collection. We have completed the first implementation, and we present some preliminary performance measurements.

Keywords: parallel and distributed implementations; garbage collection and run-time systems; data flow.

1 Introduction

This paper describes the distributed memory implementation of a shared memory parallel functional language. The language is Id [19], an implicitly parallel, mostly functional language designed in the dataflow group at MIT. Id is semantically similar to Haskell [14] and is in fact currently evolving into pH [1], a dialect of Haskell.

ICFP '96 5/96 PA, USA
© 1996 ACM 0-89791-771-5/96/0005...$3.50

Economics seems to dictate that most scalable parallel platforms in the next five to ten years will be clusters of SMPs (symmetric/shared memory multiprocessors), *i.e.*, machines consisting of a number of nodes that communicate using message passing over a switched interconnection network, where each node may be a small SMP (bus-based, 2-4 processors). Larger shared memory machines are of course possible, as demonstrated by the Stanford DASH multiprocessor [17], and the KSR machines [15], but they are likely to be high-end machines and not widely available. Further, scalable shared memory machines (like the DASH) are also built with physically distributed memories for scalability, and face some similar problems with their memory hierarchies. Even at small numbers of processors (such as 4 to 8), many more people are likely to have access to clusters of uniprocessors of that size than SMPs. Thus, our target for Id is a distributed memory, message passing machine. In our initial implementation, each node is a conventional uniprocessor workstation, not an SMP. We know how to extend this to exploit SMP nodes and believe it will be easy (handling distributed memory is the major hurdle).

The difficult problem is to bridge the gap between the shared memory model of the language and the distributed memory model of the machine. In Id, as in Haskell, SML and Scheme, the language model assumes that all data is uniformly accessible, whereas the machine has a severe memory hierarchy: a processor's access to remote memory (using explicit message passing) is typically orders of magnitude slower than its access to local memory. Thus, minimizing communication, avoiding communication if possible, and tolerating the latency of remote operations, are all crucial for good performance.

The pHluid system is a compiler and runtime system for the Id language that we have been building at Digital's Cambridge Research Laboratory for some years. Based on ideas originating in dataflow architectures[1] [11] the compiler produces multithreaded code for conventional machines.

In this paper, we focus on novel aspects of a new parallel runtime system for pHluid, in particular features that avoid communication and tolerate its latency when necessary: multithreading, scheduling and load balancing; a distributed heap model and distributed coher-

[1] The name pHluid is a play on Id, pH and data*flow*.

ent cacheing, and parallel garbage collection. We also present some preliminary performance data.

2 Background on Id and its compilation to multithreaded code

[The important messages of this section are summarized in its last paragraph. Readers familiar with topics such as Id, Id compilation, dataflow, message driven execution, fine grain multithreading, *etc.* may wish to skip this section and just read the last paragraph.]

Id [19] is an implicitly parallel, mostly functional, language designed in the dataflow group at MIT. It has many features common to other modern functional languages like Haskell and SML– higher-order functions, a Hindley-Milner polymorphic type system with user-defined algebraic types, pattern-matching notation, array and list comprehensions, *etc.* The main novelty of Id is its implicitly parallel evaluation model: everything is evaluated eagerly, except for expressions inside conditionals and inside lambdas. This is described in more detail in [19], but a key behavior relevant to this paper is that most data structures have *I-structure semantics*: given an expression of the form:

```
cons  e1  e2
```

we allocate the data structure and evaluate $e1$ and $e2$ in parallel. The reference to the cons cell is immediately available as the result of the expression. Any consumer of the data structure that attempts to read the head (or the tail) of the cell will automatically block, if necessary, until $e1$ (or $e2$) has completed evaluation and is available in the data structure[2]

The major phases of our pHluid compiler for Id are:

- Parsing, typechecking, simplification, lambda lifting, optimization, *etc.*, eventually producing *P-RISC assembler*, a fine grain multithreaded abstract machine code ("parallel RISC").
- Translation and peephole optimization, converting P-RISC assembler to Gnu C.
- Gnu C compilation, and linking with our new parallel runtime system, written in Gnu C.

We use various C extensions provided by Gnu C, such as first class C labels, and mapping C variables that contain frequently accessed data such as the heap allocation pointer into specific machine registers. First class C labels allow us to represent fine grain, dynamically scheduled threads conveniently. These Gnu C facilities have also been used by other researchers for the same purposes. An important point is that the entire compiler is geared towards producing fine grain multithreaded code for latency tolerance– we do not start with a compiler for sequential machines and add parallelism as an afterthought.

Figure 1 shows an example P-RISC assembler translation of the following function that counts the nodes in a binary tree:

```
def leaves Empty          = 1
|   leaves (Node x l r) = leaves l + leaves r;
```

It is displayed here in graphical form as a control-flow graph, but it is trivial to linearize with labels and explicit control transfers. The following features are worth

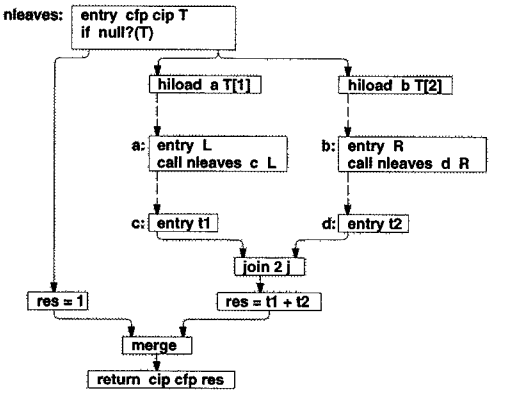

Figure 1. P-RISC assembler: An example

noting:

- The function's code can be viewed as a collection of *threads*, each of which is activated by the arrival of a message (in practice, we optimize away many messages, both statically and dynamically). Messages always arrive at `entry` instructions, whose arguments correspond to the message payload.
- At the top, a *call* message arrives, which allocates a frame for this function, and delivers three arguments to the top `entry` instruction: a continuation frame pointer, `cfp`; a continuation label (instruction pointer) `cip`, and the tree itself, `T`. The `if` tests if the tree is empty; if so, the result is `1`, and the `return` instruction sends a message to the continuation (`cfp`, `cip`) carrying the result.
- If the test fails, we initiate two "heap I-structure loads": the `hiloads` conceptually send messages to the heap locations `T[1]` and `T[2]` requesting their contents, passing the current frame pointer (implicitly) and the labels `a:` and `b:` respectively (explicitly) as their respective continuations. The current function invocation then goes dormant, with no threads active.
- The heap location `T[1]` eventually responds with a message, kicking off the thread at `a:`, placing the value (the left sub-tree) into local variable `L`. The last action of this thread is to initiate a recursive call by sending a `call` message containing the

[2]This non-strict behavior in fact makes Id semantically closer to Haskell than to SML, despite its eager evaluation. Recognizing this semantic similarity, and because of various other syntactic similarities, Id is currently evolving into pH [1], a dialect of Haskell.

continuation (current frame pointer and label `c:`) and argument subtree `L`. This recursive call eventually returns a message that starts the thread at `c:`, loading the count of the left subtree into local variable `t1`.

- Similar actions take place at `b:` and and `d:`. Thus, two threads (from `c:` and `d:`) arrive at the `join` instruction. This instruction counts up the local variable `j` (initialized to 0). Only the last thread proceeds, finding that the count `j` has reached the terminal count (2); earlier threads die at the `join`. Thus, `t1` and `t2` are guaranteed ready when the sum is computed. This final thread computes the sum and returns the result.

- It is undetermined whether thread `a:` executes before `b:` and whether `c:` executes before `d:`. They are scheduled asynchronously, as and when messages arrive. Thus, although these threads execute in some sequential order (because they all execute on the same processor), the two overall load actions (communicating to the heap and back) occur in parallel, and the two recursive function invocations execute in parallel. This kind of fundamental attention to latency is quite unique to Id compilers.

- The round-trip time for the request and response messages of a function call of course depend on how much work the function does, and whether the function call is done locally or remotely. The round-trip time for an `hiload` depends on a number of factors:

  - Whether the heap location is local or remote.
  - The current load on the processor that owns the heap location, which affects how soon it can handle the request.
  - Whether the heap location is "empty" or not when the request message arrives. Due to the non-strict, I-structure semantics, the producer may not yet have delivered the value; in this case, the request message has to be queued on that location until the value arrives.

  However, note that *all* these sources of delay are handled uniformly by couching the code as multiple, fine grain, message-driven threads, each of which *never suspends*.

All this was by way of background, and is discussed in more detail in [18]. *The important points to remember for purposes of this paper are*:

- Parallelism in pHluid is at two levels: the function call is the unit of work distribution across processors. This is real parallelism, in that these processors actually execute simultaneously. Within each function are a number of fine grain threads that are scheduled asynchronously based on message arrival. This multithreading is pseudo-parallelism, in that all threads in a function invocation are multiplexed on the same processor, but it is vital because it permits overlapping communication, synchronization and congestion latencies with useful work, and is a clean model for adaptively responding to the dynamic parallelism of the program.

- Unlike many other parallel languages, threads in pHluid are unrelated to function boundaries. Each function invocation allocates a *frame* that contains, among other things, all the local variables for all the threads of the function. However, because thread invocation does not involve any frame or stack resource allocation, threads are extremely lightweight.

- All functions share the same view of the heap, *i.e.*, the heap is a *shared memory*.

- Threads in pHluid never suspend. All function calls and heap accesses are couched as split-phase transactions: one thread issues a request message, and a response message later initiates a separate continuation thread. For heap accesses, the request message may be queued at the heap location if the value is not yet available. Thus, data access synchronization occurs at heap locations, never in the accessing functions.

Note that checking the *full/empty* state of a heap location is not an issue that is unique to I-structures in Id. Almost exactly the same issue is faced in lazy language implementations where we have to check whether a location has been evaluated yet or still contains a closure. This relationship is not surprising, because they are two different ways of implementing non-strictness (and they are both equally difficult to optimize away).

3 The new parallel runtime system

This section describes the novel aspects of pHluid's new parallel runtime system. Recall that our target platform is a distributed memory machine: each PE (Processing Element) is a conventional uniprocessor workstation. These PEs communicate by explicit message-passing using *active messages* [21], *i.e.*, each message contains a code pointer (a "handler"), and the message is consumed by simply jumping to the handler with the message itself as an argument. The handler is responsible for extracting items from the message and executing arbitrary code that may free or reuse the message.

3.1 Scheduling and Work Distribution

3.1.1 Multithreading

When a thread is executed, it may enable other threads. For example, consider the following code fragment describing two threads, starting at labels `a:` and `b:`, respectively:

```
a:  ...                                  b: ...
    hiload b T[1]
    ... foo ...
```

the thread at `a:` executes an `hiload` instruction to initiate access to a heap location, and continues (at *foo*). Conceptually, it sends a message to the heap location, which eventually produces a response message that enables the other thread at `b:`.

Each processor maintains a *scheduling stack* containing threads that are ready to run. Each entry in the stack consists of a code pointer, a pointer to the frame of the enclosing function, and a variable number of arguments. A thread is pushed on this stack when a response to a split-phase operation is received. When the current thread terminates, the system pops and executes the next thread on this stack.

We avoid a stack empty check on pops from the scheduling stack by including a *stack empty handler* as the bottom entry in this stack. This handler is invoked whenever the system runs out of executable threads. The handler pushes itself back onto the stack, and then calls the scheduler (described below) to create new threads.

3.1.2 Function calls

In pHluid, the function call is the unit of work distribution across PEs. By default, the system chooses how to do this distribution. However, by means of a source code annotation at a function call site, the programmer can direct that the call must run on a specific PE. Primitive functions are available to discover the PE on which the current function is executing, and the total number of PEs in the current run (this number is decided when starting the executable; Id code is not compiled with a knowledge of the number of PEs, expect for the special case where we know we wish to compile for a uniprocessor).

Each function call is encoded as a split-phase action. The function call itself involves creating a *call record* containing the function pointer (or *codeblock*, described below) and its arguments, including the continuation argument. The programmer can annotate any function call to specify which processor it should run on. When there is no annotation (the usual case), the runtime system chooses where to execute the function. Each processor maintains two scheduling queues containing call records:

- The *fixed queue* contains call records that must be executed on this processor.
- The *stealable queue* contains call records that can be executed on any processor.

For function calls that have a processor annotation, the call record is dispatched to the fixed queue of that processor. For calls without a processor annotation, the call record is placed on the local stealable queue. Call records may migrate between the stealable queues of various processors according to the work stealing algorithm described below. However, a function call does not migrate once it has begun executing.

Each function is described by a *codeblock*, which consists of three function entry points:

- A fast entry point which assumes the call arguments are already in registers.
- A fixed queue entry point which assumes that the arguments need to be retrieved from a call record on the fixed queue.
- A stealable queue entry point which assumes that the arguments need to be retrieved from a call record on the stealable queue.

The scheduler invokes a call record by jumping either to the fixed or to the steabable queue entry point. The fast entry point is used for an optimization described below. The code at the entry point then extracts the arguments from the call record, allocates a frame and starts evaluating the function's body. The function later terminates when one of its threads sends a return message to the function's continuation, deallocates the function's frame, and dies.

3.1.3 Work Scheduling

Naive parallel work scheduling algorithms can result in an exponential growth of the memory requirements of an application as compared to a sequential execution by creating a very large number of parallel threads, each of which simultaneously requires some local storage.

The pHluid work scheduling algorithm is designed to avoid this problem by reducing the number of simultaneously active functions, while still exploiting the available parallelism to keep all the PEs busy.

The scheduler gives work on the scheduling stack a priority higher than either the fixed or the stealable queue, in order to complete existing function activations (if possible) before creating new ones. The scheduler also invokes call records on the fixed queue in preference to those on the stealable queue, since call records on the stealable queue may later be stolen to keep a different PE active.

The scheduler tries to approximate the *depth first traversal* of the function call tree performed by a sequential execution by treating the fixed and stealable queues as LIFO queues, or stacks. When we execute a function call with no processor annotation, or with an explicit processor annotation specifying the current PE, we *push* the call record on the appropriate local queue. Each PE's scheduler always *pops* work from these queues. The net effect of our scheduler is that once all PEs have work, each PE tends to settle into a depth-first traversal of the call tree similar to that performed by a sequential implementation.

3.1.4 Work Stealing

When a PE becomes idle, it sends a `steal` message to a randomly-chosen victim PE. The `steal` message attempts to steal a call record from the victim PE's stealable queue.

Of course, the victim PE may not have any work to steal, in which case the idle PE randomly chooses a new

victim PE. In order to avoid repeatedly bothering active PEs with steal requests they cannot satisfy, we use a linear backoff scheme to decide how long the idle PE must wait before asking that victim PE again, *i.e.*, the wait time increases linearly with the number of times we fail. This scheme successfully adapts to both low-granularity and high-granularity computations.[3]

If the victim PE's stealable queue is nonempty, then the `steal` message returns the call record from the *bottom* of that PE's stealable queue (*i.e.*, away from the stack-like end) to the idle PE. The reason for this approach is that items deeper in the stack are likely to represent fatter chunks of work, being higher in the call tree [12, 16].

The net effect of our work-stealing algorithm is that we do not gratuitously fork functions to other PEs. Instead, work request messages are only sent when some PE is idle, and call records only migrate from busy PEs to idle PEs.

3.1.5 Optimizations

The pHluid system is designed using general message-sending mechanisms that work in all cases. For example, the split-phase instruction:

```
hiload b T[0]
```

conceptually involves sending a `hiload` message to the appropriate PE, which accesses the memory location `T[0]`, and then sends a split-phase return message to the original PE.

For cases where the original PE contains the memory at location `T[0]`, we avoid the overhead of sending these messages by immediately performing the `hiload` operation and pushing the enabled thread at `b:` onto the scheduling stack. We also avoid the message sending overhead in other cases by inlining other split-phase operations where possible.

We also optimize split-phase operations that are immediately followed by a `halt` instruction. For example, consider the following code fragment:

```
a:  hiload b T[0]          b: ...
    halt
```

If the `hiload` operation can be satisfied on the current processor, then the pHluid system performs the load and *immediately* jumps to the thread at `b:`, without the overhead of pushing that thread on the scheduling stack and invoking the scheduler.

Similarly, for a function call with no processor annotation followed immediately by a halt instruction, we avoid the overhead of manipulating the stealable queue by simply placing the arguments into appropriate registers and invoking the fast entry point (described above) of that function's codeblock.

[3] Our work-stealing algorithm is a variant of that used in the Cilk system developed at MIT[8]. Our algorithm was developed jointly with Martin Carlisle of Princeton University

3.2 Distributed memory model and distributed coherent cacheing

The memory of each PE is divided into five regions:

- Compiled code and static data: Static data includes data structures that describe codeblocks and the other memory areas, *etc.* These are simply replicated at the same address on all PEs.
- The *heap*: this contains actual Id data structures (constructed data, arrays and closures), and is described in greater detail below.
- The *store*: this area contains objects that are explicitly allocated and deallocated by the runtime system, and which are never accessed remotely. These include frames for function invocations, the fixed scheduling queue, heap access requests that are waiting on empty locations, *etc.* These objects are managed in a freelist organized by object size. Each PE maps its store at a different address, so that we can determine where a frame is located by looking at its address (frames are never accessed remotely, but we do need to know a frame's PE so that we can send a message to it, *e.g.*, for sending a heap access response or for sending a result to a function's continuation).
- The stealable queue: in principle this could be allocated in the store, but because it is manipulated so heavily (since most function calls go through the stealable queue), we allocate a special region for it and treat it as a deque with contiguous entries, allowing cheap pushes and pops from both ends.
- The scheduling stack: this stack is also allocated in a contiguous memory area for performance reasons.

In a previous sequential implementation, and in an initial parallel implementation, we allocated everything in a single heap (as in the SML/NJ implementation [4, 3]). However, we chose to separate out those objects that can be explicitly deallocated and which are not accessed remotely, in order to reduce garbage collection pressure. Although explicit deallocation of system objects incurs a small overhead, it does not require any communication, and significantly reduces the frequency of communication-intensive global garbage collection.

3.2.1 The heap

Heap objects are represented quite conventionally, as contiguous chunks of memory. An object consists of a header word followed by one word for each field (arrays have extra words for index bounds and pre-computed index calculation coefficients). We use the lowest-order bit to tag immediates. Thus, for example, integers lose 1 bit of precision, but this is not a problem on our current platform, 64-bit Alpha workstations. If a field of a heap object does not contain an immediate value, it always contains a pointer, either to another heap object or to a deferred list of continuations waiting for the

field to transition from *empty* to *full*. Since the field contains a pointer in either case, and since our pointers are always aligned to word boundaries, this frees up another low-order bit to use as a full/empty bit and distinguish between these two cases. This works even if the field is of immediate type (*e.g.*, integer), because for the duration that it remains empty, it contains only a pointer and not an immediate, and the full/empty bit is available.

The heap is the only area that requires global access. We use the operating system's virtual memory mapping facilities (`mmap`) so that the heap occupies the same address range on all PEs. The heap consists of a number of fixed-size "pages", each of which is currently 1 KB in size. We partition "ownership" of these pages across the PEs. This, although each PE sees the whole address range, it only owns pages representing only a chunk that is $1/P$ of the total heap size (where P is the number of PEs), and it can only allocate objects into these pages. The remaining pages are treated as a cache for the data owned by other PEs.

The advantage of this approach is that heap addresses do not have to be translated or indirected in going from one PE to another. Further, by examining an object's heap address, a PE can cheaply determine whether it owns the object or only a cached copy. The downside of this approach is that large amounts of address space are required as we increase the number of processors. Although this may somewhat limit the scalability of our system on 32-bit machines, it is not a problem on next-generation 64-bit architectures.

We conjecture that the amount of physical memory required per PE will scale in a reasonable fashion, since only a small portion of the cache pages may actually be used between garbage collection cycles. If this conjecture does not hold in practice, we intend to implement a scheme that limits the number of cache pages in use at any time by explicitly deallocating cache pages.

3.2.2 Heap cacheing

Because Id is a mostly functional language, the vast majority of objects in the heap are I-structure objects. These objects are accessed via the `hiload` operation, therefore we design our heap caching protocol to optimize this operation.

Every heap location is tagged to specify whether it is *full* or *empty* in order to implement I-structure semantics. When a heap object is allocated, all its slots are initially empty, and these slots become full when they are initialized. Once an I-structures is initialized, it can never be changed. This *functional* nature of I-structures allows us to implement a very simple cache-coherence strategy. The invariant maintained by our strategy is that each cache page is always *consistent* with the corresponding "real" page, in the sense that the only allowable difference is that a cache page may be empty at a location that is full (or defined) in the corresponding real page.

We associate a *valid* bit with every cache page. If a cache page is marked invalid, it implies that each I-structure in the page should be interpreted as being empty. Initially, every cache page is marked invalid, thus ensuring that the cache invariant holds at the start of program execution.

On a local `hiload`, *i.e.*, an `hiload` to a heap address that is owned by the current PE, we check the full/empty tag; if full, we return the value (more accurately, as described before, push the value and enabled thread on the local scheduling stack). If empty, we queue the `hiload`'s continuation on the heap location, allocating the queue entries themselves in the store, not the heap.

On an `hiload` to a location owned by some other PE, we check our cache at that location. If we're lucky, the location is full, and the value can be read immediately. Otherwise, we need to retrieve the location's value from the owner PE of that page. Since we need to communicate with the owner PE anyway, it makes sense to request an up-to-date copy of that page at the same time, in the expectation that future `hiloads` to the same page can be satisfied immediately using the new copy of the page. Provided there is no outstanding request for that page, we simply send a `hiload` message to the owner of the page, and also request an up-to-date copy of the page. However, if there is already an outstanding request for that page, then we queue the `hiload` locally, and reprocess it as soon as the new copy is received, since the new copy may allow us to satisfy the `hiload` locally, without performing any additional communication.

An `histore` ("heap I-store") operation always writes through the cache to the owner PE (there, it may cause waiting continuations to be released). However, most `histores` are to locally-owned locations. Consider a typical Id constructor expression of the form `(e1,e2)`: the heap allocation for the pair, and the two `histores` initializing the pair are part of the same function, and so the `histores` are guaranteed to be local, since memory allocation is always performed using the local memory of each PE.

Because the `histore` operation only initializes a previously empty location, it does not affect the consistency of existing cache copies of the page containing that location. Thus, the PE that owns a page never has to send any "invalidation" messages to other PEs that have previously obtained copies of that page. In fact, the owner does not have to keep track of which PEs obtained cached copies, and it is perfectly safe for the owner to send multiple copies of the same page at different times. A new copy of the page is guaranteed to be consistent with a previous copy— the only differences will be that some previously empty locations may now be full.

A general observation here is that our distributed cache-coherence protocol relies heavily on the mostly-functional nature of the source language; the protocol is trivial compared to those required for imperative languages [2].

We mentioned that Id is not a pure functional language – it has side-effecting constructs that operate on "M-structures". However, as in SML and unlike Scheme, these side-effects are constrained by the type system to specific objects and operations, and this allows the compiler to isolate them into special P-RISC assembler

instructions: `hmload` (for "heap M-structure load") and `hmstore` (for "heap M-structure store"). These never interfere with the "functional" `hiload` and `histore` operations, *i.e.*, the same heap location cannot be accessed by both I-structure and M-structure operations. Thus, the consistency of I-structure cacheing is not compromised. Our preliminary implementation of these side-effecting instructions does not involve caching – they always read and write through to the owner. We expect the overhead to be acceptable on many programs because, as in SML, side effecting operations have a much lower frequency compared to normal functional operations. However, programs that use M-structures extensively will require a more sophisticated implementation of these operations.

3.3 Parallel distributed garbage collection

The pHluid system uses a "stop-and-copy" approach for performing parallel garbage collection. Whenever any PE runs out of heap space, all the processors suspend their current activities to participate in a global garbage collection cycle. The garbage collector is a parallel extension of the conventional Cheney two-space copying collector [9].

The first action taken by each PE during garbage collection is to deallocate the cache pages (using `munmap`), since these pages will become inconsistent during garbage collection. Special care must be taken with `hiload` requests queued on a cache page. These `hiload` requests are dispatched to the owner PE, where they are queued on the specific locations that they refer to. This communication can amortized by bundling it with other communication that is necessary during GC anyway.

Next, each PE allocates (with `mmap`) a to-space into which local reachable objects will be copied. We expect that physical memory pages previously used for the cache will be reused for the to-space.

Each PE maintains two pointers into its to-space in the usual manner: a *scan* pointer and a *free* pointer. Objects in to-space below the scan pointer only contain pointers to other objects in to-space, whereas objects above the scan pointer may refer to objects in from-space.

During GC, each PE then calls the function `move_object` on each of its local roots. This function copies an object from the from-space into the to-space. For local objects, this function behaves exactly as in a conventional Cheney collector – it copies the object to the free pointer, incrementes the free pointer and returns the new address of that object.

For non-local objects, the function sends a `move_object` message to the owner PE of that object. That PE then copies the object into to-space, and returns an active message that, when invoked, updates the appropriate location in the memory of the original PE with the new address in to-space of the object. Since this operation is implemented in a split-phase manner, the original PE can continue copying other objects while the operation is in progress.

To avoid deadlock, the pHluid system uses separate queues for regular computation messages and GC messages. Each processor treats its incoming computation message queue as part of its root set, and updates all messages (*via* the function `move_object`) to refer only to to-space objects. We ensure that all outstanding computation messages are received, and hence updated, by their destination PE during GC by sending a `flush` message along the FIFO pipe connecting each pair of PEs.

Detecting termination of a global GC cycle is difficult because each PE only has direct knowledge of its own status, and messages characterizing the status of other PEs may be out-of-date by the time they are received.

We characterize the status of a PE as *inactive* once that PE has:

1. Copied all known live objects into its to-space,
2. Updated all entries in the computation message queue,
3. Received all incoming `flush` messages, and
4. Received a response to each `move_object` split-phase operation.

A PE that is inactive may become active again on receipt of a `move_object` message. The global garbage collection cycle can only terminate when all PEs are inactive simultaneously. We detect this situation using the following protocol:

Conceptually, the PEs are organized as a ring. At any time a particular PE (the "terminator-PE") is responsible for detecting GC termination. When this PE becomes inactive, it tries to send a `PEs-inactive?` message around the ring of PEs. If this message encounters an active PE, then that PE becomes the terminator-PE. Otherwise, the message is successfully forwarded around the ring and returned to the terminator-PE.

At this stage, the terminator-PE needs to ensure that all PEs are still inactive. It does this by sending a `PEs-still-inactive?` message around the ring of PEs. If this message encounters a PE which has been active since receiving the `PEs-inactive?` message, then that PE becomes the terminator-PE. Otherwise, the `PEs-still-inactive?` message is successfully returned to the terminator PE. At this stage we can guarantee that all PEs are inactive. The terminator-PE then broadcasts a message announcing the termination of the GC cycle. On receipt of this message, each PE returns an acknowledgement and resumes computation.

To ensure that distinct GC cycles do not overlap, a PE that exhausts its heap space sends a request for a new GC cycle to the terminator-PE of the previous cycle. That terminator-PE waits, if necessary, until all PEs acknowledge termination of the last GC cycle, before broadcasting a message initiating the new cycle.

4 Preliminary Performance Measurements

We have just completed a first implementation of the system as described above. We have taken some preliminary performance measurements, but we have yet

to perform a detailed analysis of these timings in order to understand exactly where the time goes.

Our implementation (both the compiler and the compiled code) runs on Digital Alpha workstations under Digital Unix, using `gcc` to compile the C "object code" and the runtime system. The parallel system runs on any network of Alpha workstations. Currently, it uses UDP sockets as the basic packet delivery mechanism. UDP sockets are unreliable (packet delivery and packet order are not guaranteed), so we implemented a thin reliability layer above UDP sockets called RPP, for Reliable Packet Protocol; this is still advantageous over TCP, which is a reliable protocol, for several reasons: it is packet based, not stream based, and so does not need "packet parsing" by the receiver, and it automatically multiplexes incoming packets from all peers into a single receive queue, instead of having to wait on N receive queues, one per peer. With RPP, It will be trivial for us in the future to exploit faster packet delivery mechanisms, even if they are unreliable.

Our parallelism measurements were taken on a particular configuration with eight 225 MHz Alpha workstations with an ATM interconnection network, running Digital Unix. Communication is still rather expensive relative to computation speed. The following table shows the round trip latency from a UDP socket send to a UDP socket receive, for various packet sizes:

Packet size (Bytes)	Round trip latency (μsecs)
64	550
128	580
256	680
1024	908
4096	1820

The maximum bandwidth is about 12 MBytes/sec (close to the ATM maximum). Thus, it is not an easy platform on which to get any parallel speedup.

For a sequential, uniprocessor baseline, we compared an Id program compiled for a single processor and executed on a single processor, with a conventional (sequential) C program that implements the same algorithm but uses conventional C idioms. The pHluid compiler was given a flag to produce code for a uniprocessor, so that it could optimize away all message-passing code. For the C compilation phase of the Id program, and for the C program, we used the same `gcc` compiler with the same `-O2` optimization level. The program we used does a lot of heap allocation: `paraffins(18)` enumerates all paraffin molecules containing up to 18 carbon atoms, unique up to certain symmetry conditions– the Id program is described in detail in [6]. The C version did not have any garbage collection (it simply had a large enough heap). One version of the C code used `malloc()` for each heap allocation, and another version did its own allocation out of a large chunk of memory allocated once using `sbrk()` at the start of the program.

`paraffins(18)`	Time (secs)	Relative speed
pHluid	0.53	1x
C (malloc)	0.40	1.3x
C (sbrk)	0.20	2.6x

We believe that these indicate that pHluid's uniprocessor performance for Id programs is approaching competitive performance for functional language implementations (the study in [13] measures the performance of another program, the pseudoknot benchmark. It compares a functional version compiled with over 25 functional language compilers, and a C version. The best functional versions were about 1.4x, 1.5x, 1.9x and 2.0x slower than C).

The following table shows the elapsed time to call and return from a trivial function that takes a `void` argument and simply returns a `void` result (in Id, `void` is the trivial type with just one value of type `void`). We measured this for two cases: a local call, where the function executes on the same PE as the caller, and for a directed remote call, where the function is directed, by means of an annotation, to execute on a different PE from the caller (this does not involve any work stealing):

Local function call	9 μsecs
Remote function call	1987 μsecs

Recall that the remote function call involves allocating a call record containing the function pointer, normal arguments and continuation arguments; sending it to the remote PE; at the remote PE, reading the message and executing its handler which, in this case, places it on the fixed queue; executing the scheduler which takes the call record off this queue, allocates a frame, and invokes the function at the fixed queue entry point, which unmarshalls the normal and continuation arguments; and, follows a similar process for the return. The messages in each direction are of the order of 100 Bytes, whose raw round-trip latency is about 580 μsecs (from the previous table); thus, the overhead is quite high, about 1400 μsecs, and is clearly an area with much room for improvement.

The following table shows the speed of the `hiload` and `histore` heap access operations. All accesses are to full I-structure locations (but they still perform the tests for emptiness, *etc.*).

	Operation	(μsecs)
1	Local `hiload`	7
2	Remote `hiload`, uncached, unmapped	1466
3	Remote `hiload`, uncached, mapped	1196
4	Remote `hiload`, cached	4
5	Local `histore`	5
6	Remote `histore`	1047

The first line describes an `hiload` to a heap object that is local, *i.e.*, on the same PE. The next three lines describe an `hiload` to a field of a remote object, *i.e.*, allocated on a different PE. Line 2 is for the case when the remote object has not yet been locally cached, and the page containing that object has not yet been mapped into the local address space. In line 3, the object is not yet locally cached, but it is on a page that has been locally mapped (due to an earlier remote access to some other location on that page). Line 5 is for the case when it has been locally cached. For lines 2 and 3, we send

a relatively small request message, and we get back an entire heap page (1 KB in size). The raw UDP communication latencies for this are about 225 + 454 μsecs. Thus, there is still about 517 μsecs overhead beyond the raw communication cost to fetch and install a cache copy of a heap page, plus about 270 μsecs if we have to map the page in. Line 4 just shows that `hiloads` to cached data proceed much faster (we do not know why it is faster than the the local `hiload` in line 1). Similarly, lines 5 and 6 are for a local and remote `histore`, respectively. A remote `histore` involves a round-trip communication (the return message is treated as an acknowledgement that the `histore` has completed).

We have been able to measure parallel speedups only for some small programs because of a current problem in an early phase of the pHluid compiler (which was implemented before the parallel runtime system) wherein list comprehensions are translated into sequential loops, thus eliminating parallelism (for example, our `paraffins` program uses list comprehensions heavily, and so we have not yet been able to measure its parallel performance). Figure 2 shows speedups for three small programs. The `trees` program is a synthetic benchmark that creates a balanced binary tree to a given depth, copies the tree several times, counts the leaves and returns the count. `nqueens(12)` is the standard N-queens problem with a board size of 12. `matrix-multiply(500)` creates two 500x500 floating point matrices, multiplies them, and sums the resulting matrix. The last program uses explicit distribution of work, directing a sub-matrix computation to be performed on a specific PE.

These figures were measured on a testbed system with "only" eight processors. Eight processors is certainly well within the range of SMPs, and one may argue that there is no need to go to a distributed memory implementation. However, we believe that it is important to look at distributed memory systems even at this size, for several reasons. First, eight-processor workstation clusters are a lot more common, and cheaper, than eight-processor SMPs. Further, there is a growing trend away from massively parallel multicomputers towards achieving the same power by assembling small-degree clusters of small-degree SMPs.

Although these figures are for relatively small programs and do not supply any diagnostic detail about how and where time is spent, we find them encouraging considering that we are using such an expensive communication layer. It would not have been surprising to see very little speedup at all, or even a slowdown. It appears that our runtime system mechanisms for avoiding, eliminating and tolerating communication are effective. We hope to do much more detailed performance studies in the near future.

5 Related Work

We are not aware of any other implementations of conventional shared memory functional languages for distributed memory machines, that address the cost of remote communications as aggressively as pHluid does. There have of course been several implementations on shared memory machines (such as [16] and [7]), and implementations of message-passing functional languages on distributed memory machines (such as Erlang [5]). However, implementing a shared memory language on a distributed memory machine appears to require a substantially different approach, with latency-tolerance a high priority throughout the design.

The Berkeley Id/TAM compiler [10] shares the same dataflow heritage as our pHluid compiler and, not surprisingly, has many similarities (although they share no code). TAM itself (Threaded Abstract Machine) is somewhat similar to our P-RISC Assembler, but our scheduling discipline for threads is quite different (these two scheduling disciplines have been compared in detail by Ellen Spertus at MIT, and is reported in [20]). The Berkeley Id-on-TAM system has been implemented on distributed memory machines with relatively fast communication facilities, such as the Connection Machine CM-5 and the MIT J-Machine, but not to our knowledge on workstation farms. The Berkeley Id/TAM system does not use distributed cacheing, nor does it have a garbage collector. We do not know what mechanism is used for work and data distribution, and if it has any automatic load balancing mechanism.

We have become aware of other implementations of functional languages on distributed memory machines— on Transputer networks some years ago, a parallel Haskell at Glasgow over PVM more recently, and a distributed memory implementation of Sisal currently in progress— but we do not know what, if any, latency-tolerance features they use. All these systems start with a compiler optimized for sequential execution, and incrementally modify them for parallel execution. In contrast, the pHluid and Berkeley TAM compilers have latency-tolerance at the forefront throughout, and consequently have a very different abstract machine model based on fine grain, message driven multithreading.

6 Conclusion

In this paper we have described our implementation of a modern shared memory parallel functional language; the novelty is in the way we target distributed memory machines, including workstations clusters with relatively poor communication facilities. Here, aggressive latency-tolerance is a central preoccupation, and is achieved by a combination of the compilation strategy (producing fine grain threads), a work-stealing mechanism that avoids distributing work unnecessarily, a distributed coherent cacheing mechanism for the heap that exploits the functional nature of the language, a memory organization that reduces pressure for global garbage collection by managing several dynamically allocated objects purely locally, and a parallel garbage collector. We also presented some preliminary performance measurements.

Our pHluid implementation is new and still immature, and there is a lot of work ahead: making it sturdy enough to handle more substantial programs; implementing it on distributed memory platforms with faster interconnects (such as Digital's Memory Channel, which achieves communication latencies of less than 5 μsecs),

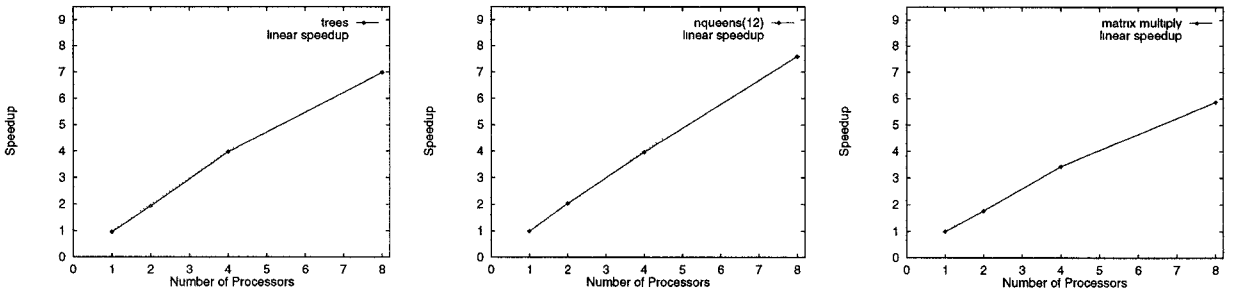

Figure 2. Parallel speedups of some small Id programs

taking detailed performance measurements and characterizing the system to understand the bottlenecks; improving the heap cacheing model to be more economical in memory use (to avoid each PE having to map the full heap); exploiting SMPs as cluster nodes (*i.e.,* using shared memory when available); adding support for large distributed arrays (currently no object, including arrays, may be larger than one PE's heap); *etc.*

Acknowledgements: The "P-RISC to Gnu C" phase of the compiler was originally designed and implemented on Sparcstations by Derek Chiou of MIT's Lab for Computer Science. We ported this implementation to our Digital Alpha workstations. The work-stealing algorithm with linear backoff was jointly developed with Martin Carlisle of Princeton University; he implemented it in Cid, another parallel language implementation at Digital CRL.

References

[1] ADITYA, S., ARVIND, AUGUSTSSON, L., MAESSEN, J.-W., AND NIKHIL, R. S. Semantics of pH: A Parallel Dialect of Haskell. In *in Proc. Haskell Workshop (at FPCA 95), La Jolla, CA* (June 1995).

[2] AGARWAL, A., SIMONI, R., HENNESSY, J., AND HOROWITZ, M. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. 15th. Ann. Intl. Symp. on Computer Architecture, Hawaii* (May 1988).

[3] APPEL, A. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters 25*, 4 (1987), 275–279.

[4] APPEL, A., AND MACQUEEN, D. B. A Standard ML Compiler. In *Proc. Conf. on Functional Programming and Computer Architecture, Portland, Oregon* (September 1987). Springer-Verlag LNCS 274.

[5] ARMSTRONG, J., VIRDING, R., AND WILLIAMS, M. *Concurrent Programming in Erlang.* Prentice Hall, 1993. ISBN: 0-13-285792-8.

[6] ARVIND, HELLER, S., AND NIKHIL, R. S. *Programming Generality and Parallel Computers.* ESCOM Science Publishers, P.O.Box 214, 2300 AE Leiden, The Netherlands, 1988, pp. 255–286. Proc. 4th Intl. Symp. on Biological and Artificial Intelligence Systems, Trento, Italy, September 1988.

[7] AUGUSTSSON, L., AND JOHNSSON, T. Parallel Graph Reduction with the <nu,G>-machine. In *Proc. Fourth Intl. Conf. on Functional Programming Languages and Computer Architecture, London* (September 1989), pp. 202–213.

[8] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, CA* (July 19-21 1995), pp. 207–216.

[9] CHENEY, C. J. A Nonrecursive List Compacting Algorithm. *Communications of the ACM 13*, 11 (November 1970), 677–678.

[10] CULLER, D. E., GOLDSTEIN, S. C., SCHAUSER, K. E., AND VON EICKEN, T. v. TAM – A Compiler Controlled Threaded Abstract Machine. *J. Parallel and Distributed Computing, Special Issue on Dataflow 18* (June 1993).

[11] GAUDIOT, J.-L., AND BIC (EDITORS), L. *Advanced Topics in Data-flow Computing.* Prentice Hall, 1991.

[12] HALSTEAD, R. Multilisp: A language for concurrent symbolic computataion. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[13] HARTEL, PIETER, H., FEELEY, M., ALT, M., AUGUSTSSON, L., BAUMANN, P., BEEMSTER, M., CHAILLOUX, E., FLOOD, C. H., GRIESKAMP, W., VAN GRONINGEN, J. H. G., HAMMOND, K., HAUSMAN, B., IVORY, M. Y., JONES, R. E., LEE, P., LEROY, X., LINS, R. D., LOOSEMORE, S., ROJEMO, N., SERRANO, M., TALPIN, J.-P., THACKRAY, J., THOMAS, S., WEIS, P., AND WENTWORTH, E. P. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. In *Workshop on Implementation of Functional Languages, J. R. W. Glauert (editor), School of Information Systems, Univ. of East Anglia* (September 1994).

[14] HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMAN,

M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIEBURTZ, R., NIKHIL, R., PARTAIN, W., AND PETERSON, J. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices 27*, 5 (May 1992).

[15] KENDALL SQUARE RESEARCH. Kendall Square Research Technical Summary, 1992.

[16] KRANZ, D. A., HALSTEAD JR., R. H., AND MOHR, E. Mul-T: A High Performance Parallel Lisp. In *Proc. ACM Symp. on Programming Language Design and Implementation, Portland, Oregon* (June 1989).

[17] LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., AND LAM, M. S. A. The Stanford DASH Multiprocessor. *IEEE Computer* (March 1992), 63–79.

[18] NIKHIL, R. S. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. 6th. Ann. Wkshp. on Languages and Compilers for Parallel Computing, Portland, Oregon, Springer-Verlag LNCS 768* (August 12-14 1993), pp. 390–405.

[19] NIKHIL, R. S. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Tech. Rep. Draft, Digital Equipment Corp., Cambridge Research Laboratory, September 23 1993.

[20] SPERTUS, E., AND DALLY, W. J. Evaluating the Locality Benefits of Active Messages. In *Proc. 5th. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, CA* (July 19-21 1995), pp. 189–198.

[21] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th. Ann. Intl. Symp. on Computer Architecture, Gold Coast, Australia* (May 1992), pp. 256–266.