Para 2010 – State of the Art in Scientific and Parallel Computing – extended abstract no. 99
University of Iceland, Reykjavik, June 6–9 2010
http://vefir.hi.is/para10/extab/para10-paper-99.pdf

Parallel Programming in Morpho

Snorri Agnarsson[*]

Faculty of Engineering and Natural Sciences, University of Iceland

Abstract Morpho is a multi-paradigm programming language developed at the University of Iceland that supports parallel programming using both fibers (coroutines) and concurrently executing tasks (threads). Communication between both tasks and fibers is through channels. Morpho is open source and an alpha version is available[1].

Keywords programming, programming languages, parallel programming, concurrency, functional programming, threads, tasks, fibers, coroutines, channels, streams

1 Introduction

The Morpho programming language is designed to be a massively scalable scripting language. It runs on top of Java, can be invoked from Java, and Morpho programs have direct access to the functionality of Java and vice versa. Morpho is, however, distinguished from Java in various ways.

- The memory footprint of Morpho tasks is orders of magnitude smaller than that of Java threads, which have corresponding functionality.
- Morpho has fibers (coroutines) which are even more lightweight than tasks. The number of fibers and tasks that a Morpho program can run is orders of magnitude greater than the number of threads that a corresponding Java program can run.
- Morpho supports functional programming as well as imperative programming and object-oriented programming.
  - Morpho has lexical scoping with nested functions and functions as first class values that can be assigned, returned as values and passed between system components.
  - Morpho has full tail-recursion removal of function calls and method invocations.
  - Morpho supports lazy streams for incremental (and parallel) evaluation of sequences.
- Morpho has a module system that supports polymorphic modules. The module system is based on the ideas described in [1]. This corresponds to Java generics, but is more powerful in some respects.
- Morpho uses run-time typing rather than compile-time typing. Variables in Morpho have no type, whereas values have types.
- Morpho has channels as a way of communicating and synchronizing.

2 Channels

Strategies for parallelization are intimately related to strategies for sharing information and passing information safely between system components. Strategies using features such as semaphores, locks and monitors are mainly intended for synchronization between components that share memory. Communication channels have a more general utility. Communication channels were developed by C.A.R. Hoare and others (see [8, 13, 9]) as part of their theory of communicating sequential processes (CSP). They were proposed as a programming language feature to facilitate parallel programming and communication. Pike's Newsqueak programming language implements channels [11]. Channels in CSP and in Newsqueak are synchronized. A receiver and a sender have a rendezvous to pass a value and we have, therefore, a notion of exact concurrency. In Morpho channels are not synchronized in this fashion. We do, however, have a notion of time-ordering since a receiver can only receive a value *after* the sender sends it. The syntax used in Morpho is similar to the one used in Newsqueak.

3 Advantages of Morpho

Morpho can be compared to other languages with message passing features such as Scala and Erlang.

3.1 Morpho and Scala

Morpho shares with Scala the feature of easy interoperability with Java. There are major differences, however, leading to some advantages offered by Morpho.

- Scala is compiled to Java bytecodes whereas Morpho is compiled to indirect threaded code (see [5, 6]) that is interpreted by Java. Thus Morpho completely escapes serious limitations imposed by the architecture of the Java runtime, in particular the assumption that

[*]Email: snorri@hi.is
[1]http://morpho.cs.hi.is

activation records are stored on a stack and that each thread has a separate stack.

- Activation records in Morpho are allocated on the heap, and more importantly in comparison to Scala the whole control chain of a running Morpho task is on the heap. This allows Morpho to support real lightweight thread-like tasks instead of simulated ones as in Scala. Scala offers two kinds of actors (see [7]). Both are comparable to a kind of combination of a Morpho channel and a Morpho task or fiber. The Scala actors are either thread-based or event-based.
  - The thread-based actors consume a lot of memory and the number of such actors is limited to some thousands on 32-bit hardware.
  - The event-based actors, on the other hand, are lightweight and it is possible to use hundreds of thousands or even millions of such actors "concurrently". But the event-based actors do not run separately but are rather implemented as closures that are "piggy-backed" on the threads of those communicating with them. They are therefore of limited use in off-loading work to separate threads or CPU cores.

3.2 Morpho and Erlang

Erlang does offer comparable parallel programming capability to that of Morpho, or even better, since Erlang supports easy parallelization of multiple computers on a network (see [4]) whereas the current implementation of Morpho has no such direct capability yet. Erlangs message passing is currently 3–4 times faster than that of Morpho. However, Morpho has some advantages that can be significant.

- Morpho interoperates directly with Java and Morpho programs can leverage functionality available in Java (and vice versa).
- Morpho supports imperative and object-oriented programming (as well as functional programming).
- Morpho automatically supports any platform that has Java support.

4 The Implementation of Morpho

Morpho uses indirect threaded code (see [5, 6]). Each operation in the Morpho virtual machine is a Java object which has a method, `execute()` that executes the operation. A running Morpho task has a corresponding Java object that runs operations in an array of operations. The Morpho interpreter in its simplest form is as shown in figure 1, where `code` is the array of operations being executed, `pc` is the current program counter, and `this` is the current task.

The run-time overhead of interpreting the Morpho operations is comparatively small, but Morpho is slowed down a little by the fact that it is intended to be a high-level scripting language and therefore uses run-time typing rather than static or compile-time typing. Nevertheless, the performance of Morpho as measured by counting the number of messages passed through channels per second, is about the same as that of compiled Java code. And Morpho supports orders of magnitude more tasks and fibers than Java supports threads.

```
for (;;)
{
  code[pc++].execute(this);
}
```

Figure 1: Morpho interpreter loop

4.1 Morpho Tasks and Fibers

Morpho supports both tasks and fibers. Tasks are comparable in funtionality to Java threads in that they run concurrently and independently of each other. Each task contains a set of fibers (also called coroutines). Only one fiber in each task is running at any one time. This makes synchronization of fibers inside the same task quite easy. Because activation records in Morpho are stored on the heap it is trivial to suspend fibers by simply storing a pointer to their current activation records and related state. The state that needs to be saved is quite small, on the order of 20 bytes, but of course it mostly consists of pointers to existing data. Similarly, creating a new fiber or a new task in Morpho is a quite trivial and fast operation.

A set of tasks in a Morpho machine share a set of Java threads that take turns in executing the tasks. Typically the number of Morpho tasks is much larger than the number of Java threads servicing them. A task whose fibers are all blocked, for example by waiting on a channel (see below), releases its Java thread until it is ready again. This leads to a natural and efficient load-balancing between cores in multi-core machines. Tasks also regularly yield their threads even if not blocked.

One might think that storing activation records on the heap would slow down the run-time considerably. However, there is good analytical and empirical evidence that this strategy should not significanty reduce performance (see [2, 3]). As memory sizes of computers become larger there is even reason to believe that this strategy could lead to faster systems than those relying on stack allocation for activation records.

5 Using Channels

- The expression `makeChannel()` creates a new channel and returns it.

- The expression `closeChannel(c)` closes a channel `c`, after which reads from the channel will eventually return a channel EOF.

- The expression `channelEOF()` returns the channel EOF value, which is a special value in Morpho.
- The expression `c<-e` writes the value `e` to the channel `c`. If the channel was in a writable state then this operation may succeed immediately and the executing fiber then immediately continues executing. Otherwise the executing fiber will wait until the channel becomes writable.
- The expression `<-c` reads a value from the channel `c` and returns it. If the channel is in a readable state then this may succeed immediately, otherwise there is a wait, as in the above write operation.

A Morpho channel can be shared by multiple Morpho fibers and Java threads. A single fiber or Java thread can listen simultaneously on multiple channels.

The Morpho function in figure 2 reads all the values from a channel and returns their sum.

```
;;; Use:   s = sumChannel(c);
;;; Pre:   c is a channel that generates
;;;        numbers.
;;; Post:  All the numbers have been read
;;;        from the channel and s is
;;;        their sum.
sumChannel =
  fun(c) {
    val eof = channelEOF();
    var sum = 0, x = <-c;
    while x != eof {
      sum = sum + x;
      x = <-c;
    };
    return sum;
  };
```

Figure 2: Summing a channel

6 Streaming Channels

Sometimes the imperative programming of channels becomes inconvenient and even confusing, for example when we need to navigate back and forth in the communication stream or when we need to use the same stream of values multiple times or in multiple places in the system. It then often is better to use a functional programming style to package a channel into a stream. Streams are functional programming constructs that stand for sequences of values. Streams may be infinite, if needed, and have the nature that a value in the stream is not evaluated until it is first used, whereupon it is memoized and not evaluated again upon further accesses.

- The expression `#[]` stands for the empty stream.
- The expression `#[ head $ tail ]` creates a stream whose first value (the head) is `head`, and whose tail (the rest of the values in the sequence) is the result from evaluating `tail`, which may be any expression. As with lazy streams in functional languages (see for example the pure functional Haskell programming language [10]), the tail expression is only evaluated if and when the tail is needed, so there is nothing to prevent us from creating infinite streams.
- The expression `streamHead(s)` returns the head of the stream `s`.
- The expression `streamTail(s)` returns the tail of the stream `s`.

Figure 3 shows how to transform a channel into a stream.

```
;;; Use:   s = streamOfChannel(c);
;;; Pre:   c is a channel that generates
;;;        some finite or infinite
;;;        sequence of values.
;;; Post:  s is a stream of the values
;;;        generated by c.
streamOfChannel =
  fun(c) {
    val head = <-c,
    eof = channelEOF();
    if head == eof {
      return #[];
    } else {
      return
        #[head $ streamOfChannel(c)];
    };
  };
```

Figure 3: Transforming a channel into a stream

We can re-implement the `sumChannel` function as shown in figure 4.

```
;;; Use:   s = sumChannel2(c);
;;; Pre:   c is a channel that generates
;;;        numbers.
;;; Post:  All the numbers have been read
;;;        from the channel and s is
;;;        their sum.
sumChannel2 =
  fun(c) {
    var s = streamOfChannel(c),
    sum = 0;
    while s != #[] {
      sum = sum + streamHead(s);
      s = streamTail(s);
    };
    return sum;
  };
```

Figure 4: Summing a channel again

We can also transform streams into channels as shown in figure 5.

```
;;; Use:  c = channelOfStream(s);
;;; Pre:  s is a stream of values.
;;;       s need not be finite.
;;; Post: c is a new channel that
;;;       generates all the values
;;;       in s.
;;; Note: s is unchanged.
channelOfStream =
  fun(s) {
    val c = makeChannel();
    startFiber(
      fun() {
        while s!=#[] {
          c <- streamHead(s);
          s = streamTail(s);
        };
        closeChannel(c);
      }
    );
    return c;
  };
```

Figure 5: Transforming a channel into a stream

7 Scalability

Morpho tasks and fibers offer one to three orders of magnitude better parallel scalability than Java threads. A Morpho program can run well over a million concurrent tasks and fibers on a 32-bit personal computer, which is likely to be sufficient for the scalability needs of most current systems. On current 64-bit hardware this number has been verified to go up to tens of millions. One reason for this improved scalability is that separate fibers and tasks in Morpho do not need separate run-time stacks. Activation records (sometimes called stack frames) in Morpho are not allocated on stacks but rather on the heap. This causes the memory footprints of fibers and tasks to be in proportion to the average length of the control chains instead of being in proportion to the maximum length of the control chains or worse. Also, because of Morpho's automatic elimination of tail-recursion, the control chains of running fibers are likely to stay quite short, further reducing the memory footprint.

Morpho programs interface in a natural fashion, through channels, with the new event-driven IO features of Java (Java.nio, see [12]), which gives us the best of both worlds, the scalability of event-driven IO as well as the convenient imperative and functional programming styles of Morpho channels and streams. Java.nio offers one to two orders of magnitude better scalability than regular socket programming in Java, in regard to possible throughput and in regard to the number of concurrent connections that can be served.

8 Conclusion

Morpho can be used with Java to improve scalability and programming convenience in massively parallel system development. The improvements are largely due to a huge reduction in the memory footprint of Morpho tasks and fibers relative to Java threads.

References

[1] S. Agnarsson and M. S. Krishnamoorthy. Towards a Theory of Packages. *SIGPLAN Notices*, 20:117–130, June 1985.

[2] A. W. Appel. Garbage Collection Can Be Faster Than Stack Allocation. *Inf. Process. Lett.*, 25(4):275–279, 1987.

[3] A. W. Appel and Z. Shao. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming*, 6, 1994.

[4] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[5] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[6] R. B. K. Dewar. Indirect threaded code. *Commun. ACM*, 18(6):330–331, 1975.

[7] P. Haller and M. Odersky. Actors that Unify Threads and Events. In *Proc. COORDINATION 2007*, 2007.

[8] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 26(1):100–106, 1983.

[9] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[10] S. Peyton-Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, pages 47–96. Press, 2002.

[11] R. Pike. The Implementation of Newsqueak. *Softw., Pract. Exper.*, 20(7):649–659, 1990.

[12] W. Pugh and J. Spacco. MPJava: High-Performance Message Passing in Java using Java.nio. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 323–339, 2003.

[13] A. W. Roscoe, S. Brookes, and C. A. R. Hoare. A Theory of Communicating Sequential Processes. *Journal of the ACM*, (3):560–599, July 1984.