

# A Common-Lisp Implementation of Futures for Clusters

Paul-Virak Khuong, Marc Feeley  
{*khuongpv,feeley*}@iro.umontreal.ca

February 26, 2008

## Abstract

Futures are a high-level parallel processing construct which has been popular on shared-memory multiprocessor implementations of Lisp. Futures are often managed using a distributed task queue, local scheduling; load-balancing is achieved with task-stealing. In this paper, we describe an implementation aimed at large clusters of workstations where communication is expensive and tasks are coarse grained. Due to the larger relative cost of computation we use a centralized task management, which allows more complex logic. We also discuss a new higher-order message-passing library which simplified the development of the system and of user code. A performance evaluation is provided.

## 1 Introduction

The `future` construct has a long and successful history [Multilisp, Cilk, ?] as a way to express task-level parallelism on shared-memory machines. The evaluation of futures follows a non-strict strategy. As in call by name, extracting the value of a future may block until the value has been computed. Unlike thunks in call by name, futures may be evaluated eagerly to exploit parallelism.

On shared-memory machines, processors are tightly coupled, and communication and synchronization are inexpensive. Fine grain parallelism is thus an acceptable approach. The task management overhead can be lowered by using lazy task creation [kranz,feeley] which maintains local task queues and performs work stealing to balance the workload. On a cluster, nodes are more loosely coupled, often using off-the-shelf networking technology such as Ethernet. Communication between nodes is considerably more expensive. A simple communication can be as long as executing 10,000 machine instructions. Algorithms with fine grain parallelism must be avoided to achieve good efficiency. A mostly functional, coarse grain dataflow parallelism approach is more suitable for clusters. To support this programming style we have chosen to centralize task management.

<code>create-task</code> <i>creator</i> <i>task-defn</i>	Create a new task, initializing its external reference count to 1, and send the task's identifier to <i>creator</i>
<code>destroy-ref</code> <i>task-id</i>	Decrement the task's external reference count
<code>get-task</code> <i>destination</i>	Send a task that's ready for execution to <i>destination</i>
<code>task-done</code> <i>task-id</i>	Mark task <i>task-id</i> as completely executed
<code>wait-on</code> <i>destination</i> <i>task-id</i>	Send task <i>task-id</i> back to <i>destination</i> as soon as it is completely evaluated
<code>get-value</code> <i>destination</i> <i>task-id</i>	If task <i>task-id</i> is completely evaluated, arrange for a worker to send that task's value to <i>destination</i> . Otherwise send a <code>not-ready</code> message.

Figure 1: Messages processed by the server

## 2 Centrally-Scheduled Futures

We use a client-server architecture. The server is responsible for the management of tasks: scheduling and distributing them to minimize communication costs while exploiting as much parallelism as is available. Centralization offers a global view of the task dependence graph constructed by futures. This global graph simplifies scheduling. It is straightforward to detect “dead” tasks (tasks whose result is no longer relevant), to deal with node failure, and to rewrite the graph itself. Knowledge of the complete set of tasks for distribution makes it possible to allocate tasks to minimize communication, and reduces the importance of nested parallelism as a means of load balancing.

A complex server allows us to have much lighter-weight workers. Workers only have to manage data they receive, and the single task that is currently executing. These processes query the server for work to simplify the handling of (worker-)node failure. Data, the result of tasks evaluation, is distributed to avoid sending all that data to a server, and from the server to workers. Instead, the server sends queries to workers on behalf of other workers, and the data is sent directly between workers.

Clients, responsible for the creation of futures, are also simple. Creating a future consists of sending the task, basically a closure, along with the futures it depends on (or rather, their identifier) to the server, and getting an identifier back. A future, storing that identifier, is then returned and manipulated by the program instead of the future's value itself. When that future is not used anymore, a message is sent to the server to signal that the only references to that task are in the task graph itself. If the value of a future is needed, the client signals the server, and the value is received from a worker node. Every message in the client is synchronous, so no event loop is needed. Since the

<code>run-task <i>task-id</i> <i>thunk</i></code>	Create a new thread to evaluate <i>thunk</i>
<code>nil</code>	“no-op” message, sent in response to <code>get-task</code> when no task is available for the worker
<code>terminate <i>task-id</i></code>	Terminate the evaluation of task <i>task-id</i> if applicable, and remove any reference to that task’s result
<code>send-value-to <i>destination</i> <i>task-id</i></code>	Send the result of task <i>task-id</i> to <i>destination</i>
<code>new-value <i>task-id</i> <i>value</i></code>	Save <i>value</i> as the result of task <i>task-id</i>

Figure 2: Messages processed by workers

client is blocked until that value is received, querying for a future’s value may starve the workers of tasks, in addition to causing additional communications. Starvation may be alleviated by using nested parallelism. However, to minimize communications, it is necessary to pass a continuation as a future, which isn’t always practical in a language without native continuations.

### 3 Application: Parallel Matrix Multiplication

A toy implementation of blocked square matrix multiplication is presented (figure 3). Matrices are represented as quad-trees, with dense rank-2 arrays at the leaves. The code is extremely similar to what one would write to improve locality of access, a fundamental issue on modern architectures, with some annotations for parallelism.

The basic futures construct are `(make-future function argument ...)` and `(value-of future)`.

`make-future` makes the arguments explicit to offer more information and guide scheduling: any argument that is actually a future will have been fully evaluated and replaced with its value when the function is called. `value-of` forces the value of its argument (a future), blocking if needed.

Some syntactic sugar is also offered. The macro

```
(spawn ((var1 value-form1) ...)
  body-form ...)
```

expands to

```
(make-future (lambda (var1 ...) body-form ...)
  value-form1 ...)
```

and is thus to `make-future` as `let` is to `lambda`. Additional code is also generated to simplify the use of nested parallelism.

```
(defgeneric mat-mult (a b inc)
  (:documentation
   "returns a x b      if inc is nil,
    a x b + inc otherwise.

   It is assumed that a and b (and inc, if non-nil)
   are of the same size.")
```

Figure 3: Definition of the matrix multiplication function

```
(defmethod mat-mult (a b inc)
  "Assume that a, b and inc will evaluate to trivial matrices."
  (spawn ((a a) (b b) (inc inc))
        (if inc
            (base-mult-add a b inc)
            (base-mult a b))))
```

Figure 4: Base case of matrix multiplication

The base case (figure 4) spawns a future that executes a serial matrix multiplication. The arguments are not forced immediately, instead letting the scheduling system pass their values directly to the worker node.

The recursive case (figure 5) creates futures in the 4 subcomputations (2 serial half-sized multiplications each) and gathers the 4 quadrants of the output matrix together. It is identical to the recursive case of serial blocked matrix multiplication, except that the leaves of the resulting tree contain futures instead of dense matrices.

## 4 Message-Passing Library

The message passing library used to build the futures system abstracts away the difference between shared-memory and inter-process communications. Thus, it must be able to serialize ground data and functions across processes. Since parallel algorithms usually work with large data sets, often matrices and vectors, it is also desirable that the serialization of such objects be particularly efficient.

The object graph is traversed in post-order, and a stack used to resolve circular references during deserialization. The serialized representation is outputted sequentially, making it possible to write it to a stream on the fly. This is especially useful for large arrays of unboxed data, for which copying can be completely avoided.

Deserialization can then read the serialized representation sequentially. A serialized object may only refer to objects that have already been serialized or use explicit forward references. Normal references are resolved by indexing in an

```

(defmethod mat-mult ((a blocked-matrix) (b blocked-matrix) inc)
  (with-indices ((a (children-of a) 4) ; a: (children-of a), a.i: (aref a i)
                (b (children-of b) 4)
                (i (if inc
                      (children-of inc)
                      #(nil nil nil nil))
                 4))
    (let ((o0 (mat-mult a.0 b.0 i.0)) ; spawn all the subcomputations
          (o1 (mat-mult a.0 b.1 i.1)) ; before their dependents
          (o2 (mat-mult a.2 b.0 i.2))
          (o3 (mat-mult a.2 b.1 i.3)))
      (make-blocked-matrix (mat-mult a.1 b.2 o0)
                           (mat-mult a.1 b.3 o1)
                           (mat-mult a.3 b.2 o2)
                           (mat-mult a.3 b.3 o3))))))

```

Figure 5: Recursive case of matrix multiplication

array of serialized objects; forward references to an object, if any, are resolved once it is completely deserialized.

Although very little data consists of functions, completely forgoing them in messages, e.g. by doing explicit closure conversion, unnecessarily increases coupling between communicators. This is particularly an issue when the messages are futures. To avoid this problem, a restricted form of function serialization is implemented. Functions are assumed to have been compiled and given a global identifier in a common “core” (a similar approach is used for class and structure definitions). Closures can then be serialized as tuples containing the correct identifier and environment. We use an implementation which represents lexical environments as a flat vector of values or value cells, so the sharing of bindings is implicitly preserved.

The library currently uses TCP connections for communications, but can easily be adapted to any communication backend that provides atomic non-blocking point-to-point messages, such as MPI. The serializer is also not fundamentally tied to the host implementation. Even closures can be represented portably, using a codewalking-based approach. A generic implementation could thus be developed. Unfortunately, performance of serialized code would likely be affected; the current approach does not require any change to the compiled program.

## 5 Results

Numerical results and comparisons (figures 6, 7 and 8) are not yet available.

Figure 6: Time VS number of processors for size k

Figure 7: Overhead for single-process case VS size of matrix

Figure 8: Overhead compared to direct call to BLAS VS size of matrix

## 6 Future Work

We do not fully take advantage of the centralized dependence graph exposed by our client-server futures. That information could be used to guide the scheduling of futures according to priorities, to detect circular dependencies or to be able to survive node failure. It would also be interesting to allow users to define local transformations to optimize the graph, for example to fuse maps and folds or to address other cross-cutting concerns.

It was pointed out that continuations could be exploited to minimize communications, by migrating code instead of data. While we could use a language implementation with native serializable continuations, it is also possible to develop restricted portable continuations above serializable closures.

## 7 Conclusion

We presented and developed a centralized scheme for the distributed evaluation of futures that does not require special runtime support. The performance and scaling properties of this scheme were evaluated. We also developed a higher-order message passing library. Extending these codes to be portable, especially between processes of different implementations, could be helpful in the development of distributed applications in heterogeneous environments. The application of futures to cluster computing is hoped to simplify the development of distributed applications, compared to typical approaches in the style of MPI, OpenMP or Map/Reduce. It still remains to be seen whether coarse-grained futures are well-suited to the parallelization of a diverse class of algorithms, both iterative and recursive.