Crystal Scheme
A Language for Massively Parallel Machines

Second Symposium on High Performance Computing, Montpellier (France), October 1991, pp 91-102, North-Holland.

Christian Queinnec*
École Polytechnique & INRIA–Rocquencourt

Abstract

Massively parallel computers are built out of thousands conventional but powerful processors with independent memories. Very simple topologies mainly based on physical neighbourhood link these processors. The paper discusses extensions to the Scheme language in order to master such machines. Allowing arguments of functions to be concurrently evaluated introduces parallelism. Migration across the different processors is achieved through a remote evaluation mechanism. First-class continuations offering some semantical problems with respect to concurrency, we propose a neat semantics for them and then show how to build, in the language itself, advanced concurrent constructs such as futures. Eventually we comment some simulations, with various topologies and migration policies, which enables to appreciate our previous linguistical choices and confirms the viability of the model.

Massively parallel computers [Hewitt 80, Dally *et al.* 89, Germain *et al.* 90] are large ensembles comprising thousands of conventional but powerful processors equipped with independent memories. Their total throughput confers them tremendous computing potential but they still remain to be tamed. Such machines usually have a crystalline structure where a processor is only connected to its neighbours. No direct global addressing is possible within such a machine: informations can only flow from processor to processor on a neighbourhood basis. For the same reason, there is no shared memory. Various topologies exist and mainly hypertorus [Hewitt 80]. Only simple topologies achieve *scalability* i.e. the possibility to make a machine grow by simple addition of processors without logical nor physical discontinuity. To realize a chip which can be directly interconnected in three dimensions seems to be now attainable [Béchennec89] and confers much value to the whole approach.

We follow two ideas, similar to these expressed by [Chien & Dally 89]. First, programming should be relatively easy i.e. the mental model of the computation on so many processors with so many processes must be very simple to be still understandable; second, the language must allow to express sufficient concurrency to utilize the machine. Similarly to Halstead [Halstead 85], we choose to stick to an already familiar algorithmic language: Scheme [Rees & Clinger 86]. We only slightly modify or extend it with the needed capabilities. This approach growing from a small, powerful and semantically clean basis is our preferred way to easily experiment new features. We also try, following the Scheme philosophy, to introduce the minimal capabilities to satisfy our goals.

The modified semantics of Scheme allows to concurrently evaluate the terms composing functional applications. This induces programming style changes as well as a modification of the meaning of continuations. Since concurrency is only introduced on a local basis, a remote evaluation mechanism [Stamos

*Postal Address: Laboratoire d'informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France. Email Address: `queinnec@poly.polytechnique.fr`. This work has been partially funded by Greco de Programmation.

& Gifford 90] makes computations flow over the whole machine. Computations migrate but objects still reside on their birth site.

We retain the full capabilities of Scheme and, in particular, do not forbid side-effects nor first-class continuation. We thus stick to a very simple, even naïve, model: it is our aim to demonstrate that this model is viable through a series of simulations with varying topologies and migration strategies.

The paper is organized in two main parts. The first part (section 1) describe the linguistical side and presents the semantics of our modifications to Scheme. The second part collects and comments results of simulations made on machines presented in section 2. We prove the viability of the previous linguistical choices in section 3. Future (section 4) and related (section 5) works conclude the paper.

1 Concurrent Scheme

We decide to offer a small but complete algorithmic language with simple semantics in order to understand the influence of our proposed extensions. We choose Scheme [Rees & Clinger 86] for that goal since side-effects and continuations are useful reflexive tools to express details of the implementation of Scheme in Scheme itself. This section focuses on two kinds of features: how to introduce concurrency and how computations migrate.

Scheme is a sequential untyped language which, roughly, enriches λ-calculus with assignment and mutable data structures as well as indefinite-extent first-class continuations. We will refer to it as *regular* Scheme and to our modified Scheme as Crystal Scheme.

When a form such as (F A B C) is to be evaluated, the different terms F, A, B and C are concurrently evaluated. Different processes (or tasks) are started yielding values f, a, b and c. When all values are computed, f (presumably a function) is applied on the other values. This behaviour, often nicknamed `pcall` for parallel call [Halstead 84], is an implicit fork-join model. A denotational semantics for such a modified Scheme is fully detailed in [Queinnec 90]. Although these changes cannot be perceived from a purely functional language, side-effects and first-class continuations in Crystal Scheme induce important consequences. Instead one feature of regular Scheme alleviates these consequences. Although sequential, regular Scheme does not specify the order along which terms of a form are evaluated. This leads to a sort of undeterminacy; consider for instance the following program:

```
(car (let ((x 1)) (list x (set! x (+ 5 x)))))
```

This form may return 1 or 6 depending on the order of evaluation of the arguments of `list`.

Crystal Scheme does not enforce a particular scheduling strategy. Therefore any value that might be yielded by a form in regular Scheme can also be yielded by the same form in Crystal Scheme. Any sequential strategy is a legal parallel strategy but there are many other possible parallel strategies. This is why Crystal Scheme may return values that no sequential strategies can ever return:

```
(let ((x 1)(y 2))
   (list (begin (set! x (+ x y)) x)
         (begin (set! y (+ x y)) y) ) )
```

This form can only return `(3 5)` or `(4 3)` in regular Scheme. It can also yield `(3 3)` in Crystal Scheme if the two assignments are interleaved. We will see later on more curious behaviours.

1.1 Continuations

Continuations are first-class objects in Scheme and are captured thanks to the `call/cc` function. Since we alter the semantics of applications, the continuation of a term appearing in an application differs from what it was in regular Scheme. We will describe this new behaviour on the following particular example:

```
(list (display 1)
      (call/cc (lambda (k) ...))
      (display 2) )
```

If, in regular Scheme, arguments are evaluated from left to right then the continuation `k` is exactly represented by `(lambda (r) (list 1 r (display 2)))`. Observe that this continuation captures the evaluation of the third argument of `list`. Arguments being concurrently evaluated in Crystal Scheme, this capture must not exist[1]. We therefore propose the following temporary definition for `k`:

1: store the value r as second argument of `list`,
2: if all arguments are now present
  then apply `list` on them and continue the evaluation
  otherwise kills the process yielding r.

More generally, all the processes which compute the various subterms of a form are killed after yielding a value except the last one which performs the application.

Since regular Scheme continuations have an indefinite extent, they may be multiply invoked. An expression may thus return multiple results. Suppose that an extra second result is returned by the second argument of `list` as in:

```
(list (display 1)
      (call/cc (lambda (k) (+ (k 3) (k 5))))
      (display 2) )
```

When the extra second argument occurs, two cases are possible wether the other arguments are all present or not, see figure 1.

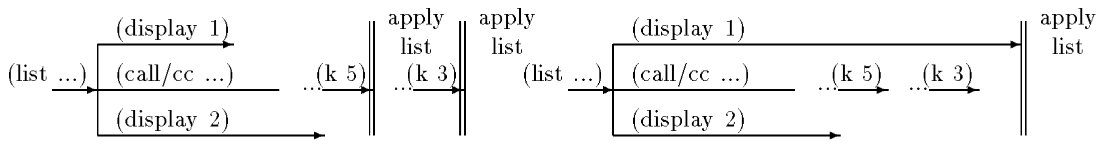

Figure 1: Race between multiple results

In the first (left) case, all other arguments are ready then *two* applications of `list` will be performed. In the (right) other case and if keeping the previous definition of the continuation `k`, only *one* application will be performed with the more recent second argument. We think that this race condition is unfair and thus we modify the meaning of `k` to preserve the number of applications and their associated sets of arguments. `k` now appears as:

1: if the second argument of `list` is still unknown
  then store r in it
  otherwise, record that a new second argument has been computed
2: if some arguments are missing
  then kill the current process
  otherwise
    a: apply `list` and continue the current process
    b: concurrently, for each deferred (recorded) extra arguments
      store it at its right place and create a new process by
      applying `list` on this new set of arguments

We can thus ensure that every invocation to a continuation always create a process even if it is deferred until the application is performed. To return to our previous example:

```
(list (display 1)
      (call/cc (lambda (k) (+ (k 3) (k 5))))
      (display 2) )
```

This form always returns two results: `(1 3 2)` and `(1 5 2)`. We clearly depart from regular Scheme which only returns a single result. Note that to call `+` in the above form is just a trick which allows to compute concurrently `(k 3)` and `(k 5)` without harm since the addition will never be performed. To avoid the use of a binary function, such as `+`, to introduce the parallel computation of two expressions, we will use in the sequel the more elegant `cobegin` macro:

```
(defmacro (cobegin e1 e2)
```

[1] Forms can nevertheless be serialized thanks to the `begin` construct of Scheme.

```
(let ((k (gensym)))
  `(call/cc (lambda (,k) (+ (,k ,e1) (,k ,e2)))) ) )
```

Observe that the two processes that are created above do not have any synchronisation between them, they are completely unrelated and will continue to be run until reaching the initial continuation. The naïve initial continuation in a read-eval-print loop based system [Katz & Weise 90] is to start a new cycle thus leading to two concurrent loops ! A less naïve initial continuation is to kill all returning processes and wait until there is no more running processes at all; when all computations are finished, a new cycle can then be started.

This ability to create independent processes helps us to program interesting features such as generators with multiple simultaneous results [Mitsolides & Harrison 90]. Moereover regular Scheme can also adopt this style of continuations.

1.2 Atomic Exchange

In regular Scheme, the value returned by an assignment is unspecified, we choose to return the former value of the binding. The form `(set!` *name expression*`)` first computes the value of *expression* then atomically exchange this value with the content of the location bound to *name* and to return the latter. Denotational details appear in [Queinnec 90]. To have this atomic exchange facility allows us to program, at the user level, busy-waiting loops and semaphores similarly to [Halstead 85]. Non preemptive schedulers can also be defined in a manner reminiscent of [Wand 80]. Eventually, we can describe futures [Halstead 84] in Crystal Scheme itself without resorting to magic functions such as **`make-future`** or **`determine-future`**. To make the description simpler, we omit all critical sections and hygienic renamings [Kohlbecker *et al.* 86]. To simplify even more, we take the interface of the usual **`delay`** and **`force`** of regular Scheme i.e. a delay must be explicitely **`force`**-d to deliver its value.

```
(defmacro (delay e)
  `(let ((computed? #f)
         (queue '())
         (value 'wait) )
     (cobegin       ;;concurrently,
        (lambda ()  ;;return the delay
            (if computed?
                value
                ;wait until value is computed
                (call/cc (lambda (proc)
                           (set! queue (cons proc queue))
                           (suicide) )) ) )
        (begin    ;;and compute its value
              (set! value ,e)
              (set! computed? #t)
              (foreach (lambda (proc) (proc value)) queue)
              (set! queue '())
              (suicide) ) ) ) )
```

The difference with a real Scheme delay is that its computation is done concurrently with its use. This behaviour raises very subtle issues discussed in [Katz & Weise 90] that may again be programmed in Crystal Scheme. The **`suicide`** function kills the current process; it can be, very inefficiently, simulated by an endless computation.

1.3 Migration

Concurrency is introduced on a local basis thanks to application. A function named **`remote-funcall`** allows to perform an application on a remote site. On a regular topology a remote site is one of the neighbour of the current processor. **`remote-funcall`** is a regular function and thus does not involve a modification of the Scheme compiler.

`(remote-funcall` *function arguments...*`)`

The subforms *function* and *arguments* are first evaluated on the current site then **`remote-funcall`** is applied and the value of *function* is applied on the values of *arguments* on a remote site. Memories are distributed so a remote call involves the function, its arguments and its continuation[2] to migrate. We decide to offer the illusion of a global space virtually accessible from everywhere. Entities (pairs, closures, continuations ...) do not move, they reside on the site where they were created. When an entity o migrate to the processor in direction d (where d is north or south-west or up etc.) a remote pointer $< \overline{d}, o >$ is created on this remote processor, $\overline{d}$ being the opposite direction to d. A remote pointer $< d_1, < d_2, o >>$ on processor P designates the entity remotely pointed by $< d_2, o >$ from the processor P' which is the d_1-neighbour of P. A remote pointer is thus a chain of relocations which ends with the entity itself. A single simplifying rule exists for the migration of remote pointers:

$$\forall o, \forall d, < \overline{d}, < d, o >>= o$$

Of course there exists some "immediate" entities that may be directly exchanged such as small integers, booleans, characters, or even universally known simple primitives such as `+` or `cons`.

Computations on entities are handled through the "geographical commutation rule" :

> To compute the geographically strict function c on $< d, o >$ with continuation k on processor P is just computing c on o with continuation $< \overline{d}, k >$ on processor P' which is the d-neighbour of P.

Examples of computation c include looking up an identifier in a remote environment, sending a value to a remote continuation, store a value in a remote pair ... All Scheme computations follow this rule except comparison primitives such as **`eq?`**, **`eqv?`** ... which need a special implementation when comparing two remote entities coming from different directions.

Let us consider the following computation:

$_k$`(remote-funcall cdr (cons 'foo 'bar))`

The various terms are computed and respectively yield the primitive *cdr* and the dotted pair $\boxed{\text{foo}}\boxed{\text{bar}}$. When applied on these two values, **`remote-funcall`** decide to send them as well as its continuation k towards the south neighbour. The objects that are received by the south neighbour are <north,*cdr*>, <north,$\boxed{\text{foo}}\boxed{\text{bar}}$> and <north,$k$>. The <*direction*,*object*> notation represents a reference to a distant *object* located to the *direction* of the current site. We can expect any primitive such as *cdr* to be known everywhere therefore <north,*cdr*> is simply *cdr*. The *cdr* primitive is geographically strict: it reads a precise location only on the site where resides this very location. Therefore to apply *cdr* on <north,$\boxed{\text{foo}}\boxed{\text{bar}}$> giving the result to <north,k> is just to transfer the computation towards the site which holds the dotted pair i.e. the north neighbour. The computation is therefore sent again, this time to the north. When sent to the north <north,$\boxed{\text{foo}}\boxed{\text{bar}}$> is just <south,<north,$\boxed{\text{foo}}\boxed{\text{bar}}$>> which simplifies to $\boxed{\text{foo}}\boxed{\text{bar}}$. The final computation is therefore the application of *cdr* on $\boxed{\text{foo}}\boxed{\text{bar}}$ and the result is given to k which is precisely the (simpler) computation $_k$**`(cdr (cons 'foo 'bar))`**.

Observe that computations carry their continuation and therefore are not bound to the sites through which they pass. This is a sort of "geographical tail elimination".

The main cost of **`remote-funcall`** is due to the migration of objects. Primitives, small integers are simple to transfer. Compound data such as dotted pairs or closures do not migrate and are only remotely pointed thus inducing the commutation rule to create more and more processes on different sites when these compound date must be read or written. Some sharing analyses [Deutsch90] can be made to determine objects that could gain if copied onto remote sites rather than being remotely pointed. When transfering a closure, the code part of it can be cached i.e. only transfered the first time, since it cannot be written and therefore can be safely duplicated; the other part i.e. the environment, is remotely pointed unless a mutability analysis is done which allows to duplicate the locations of read-only variables.

The **`remote-funcall`** facility is the minimal imperative feature that forces computations to flow over the whole machine but we do not want to bother the user with it. We therefore embed a weaker form of this facility in the semantics of Crystal Scheme application: any application might be remotely computed

[2]In effect the result of an application (even remote) has to be returned to its continuation.

according to the compiler or the run-time. We thus alter the semantics of application to handle all the problems of concurrency and migration but we do not enforce it. A sequential implementation of regular Scheme is therefore an almost legal implementation of Crystal Scheme let alone continuations.

1.4 Comparing Regular Scheme and Crystal Scheme

Crystal Scheme is very close to regular Scheme and can be summed up in four points:

- we constrain assignment to atomically return the old value,
- an application might compute concurrently its arguments and might be remotely applied,
- each call to an argument continuation ensures that an application will eventually be performed even if deferred,
- some functions such as `suicide` or imperative `remote-funcall` are added to the standard library.

This extensions confer Crystal Scheme surprising abilities while making continuations harmoniously co-exist with concurrency.

Does Crystal Scheme computes what regular Scheme computes ? For any sequential strategy there is an equivalent parallel strategy, thus any observed behaviour (a final result or an endless loop) in sequential mode may as well be observed in parallel mode. Many other *bizarreries* can be observed. A form may have multiple results, some of which cannot be computed sequentially: an example was previously given. A form may loop under some parallel scheduling strategies even if it yields a result under any sequential mode.

We use hereafter the `until` macro which loops evaluating an expression until its value is true:

```
(defmacro (until expression)
  `(letrec ((loop (lambda ()
                    (if ,expression #t (loop)) )))
     (loop) ) )

(let ((x #f))
  (until (not (eq? (begin (set! x (not x)) x)
                   (begin (set! x (not x)) x) ))) )
```

Whatever the evaluation order is, this form finishes in regular Scheme. A breadth-first parallel strategy performing the two assignments before the two final references to `x` would loop forever.

A form may loop under any sequential mode, but it may yield some (or even no) values under particular parallel strategies:

```
(let ((b1 #f) (b2 #f))  ;returns (#t #t)
  (list (begin (set! b1 #t) (until (and b1 b2)))
        (begin (set! b2 #t) (until (and b1 b2))) ) )

(let ((k1 'wait))       ;terminates without value (another suicide) !
  (cons (begin (until (procedure? k1))
               (k1 'foo) )
        (call/cc (lambda (k) (set! k1 k) 'bar)) ) )
```

With any fair strategy, where a runable process is never indefinitely blocked, the first form returns a result while the second terminates without any result. The first argument of `cons` will never return a value since it only provides an extra result to the second argument to `cons` !

One might think that living in such a world is difficult, the work of Halstead and his followers have proved that such problems are rare. The possibility of abstraction can hide a large majority of them without forbiding the use of safe side-effects.

2 Megamachines, Topology and Migration

We only consider massively parallel machines. We assume that, to be built, these machines must have a crystalline structure where a common pattern is repeated again and again. Such scalability on the hardware side must have its counterpart on the software side: linguistical means to express concurrency must be simple to understand and use.

We made some simulations with some topologies. We only consider isotropic topologies in order to avoid problems with the processors located on the edges [Hewitt 80]. Note that it is possible to imagine and build on a computer topologies that are different from the seven chemical crystalline structures. We restrict our study to the three following topologies:

linear-torus: Every processor is linked to two neighbours located on its east and west side. The last one has the first one as its east neighbour and reciprocally.

planar-hypertorus: Every processor has four neighbours located at north, east, south and west, see figure 2.

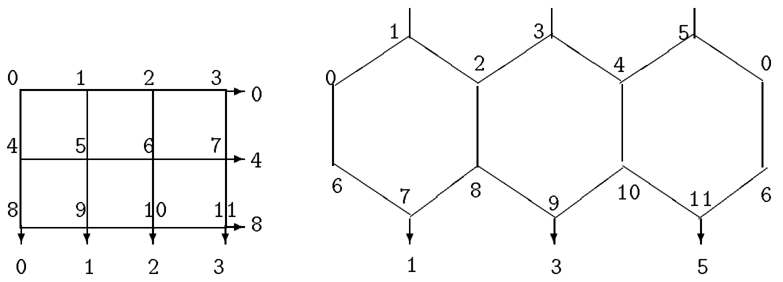

Figure 2: Planar Hypertorus (hypertorus4*3) Planar Beehive Hypertorus (behive2*6)

planar-beehive-hypertorus: (in honour of the Apiary Project) where every processor has three neighbours as shown in figure 2

Each of these topologies have 3D extensions increasing the number of neighbours but exceeding the capabilities of our simulator. Besides the physical topology is the migration policy i.e. when and where are remotely applied applications. We mainly inquire two policies:

flow: The crystal is considered as a vector field where exists preferred directions along which are transferred remote applications. Computations starts from the initial processor and explodes [Germain & Giavitto & Sansonnet 91] from it. The vector field is set-up to favour the quest for idle processors. For example, in planar hypertorus remote applications are alternatively done towards east or south.

dilute: Remote applications are transferred on each neighbour on a round-robin basis. There is thus no preferred directions and sometimes processes are sent to the direction where they come from.

We choose five machines for our simulations:

single: the one-processor sequential machine without concurrency nor topology,

linear4: four processors linked in a torus,

linear16: sixteen processors linked in a torus,

hypertorus4*4: a planar hypertorus with sixteen processors,

beehive2*8: a planar beehive hypertorus machine with sixteen processors.

3 Benchmarks

We first derived a simulator from the denotational semantics of the concurrent Scheme afore presented. Then we added the description of the machines, their topologies and migration policies in order to implement remote applications. We then instrumented the resulting programs to keep track of the number and the nature of steps of the computations. Although these steps cannot be accurately compared since they do not last equal times, they provide good indications of the trends.

The benchmark suite is given in appendix and defines the well-known Fibonacci, Quick-Sort and Queens functions. The benchmark is not written in pure Crystal Scheme since we wanted to have greater control, we explicitely insert `pcall` and `remote-funcall` where these effects must imperatively be done. These are simple annotations that could be synthesized by any non-dumb compiler. A `remote-funcall` nests any non-terminal call to a recursive function. A `pcall` nests any form which function is not an integrable primitive with at least two non trivial subforms.

We perform the tests[3] of table 1. Fibonacci is a good test since it creates a lot of processes and only needs to transfer small integers. Quicksort is a bad one since it handles dotted pairs which must be remotely pointed and, worst of all, offers a bottleneck in `separate` and `merge`. To sort an already sorted list is a bad case of quicksort but can be compared against sorting a random list. The result of `qsort` is a list that may be constructed on different sites, to sort it again leads to a distributed quicksort. The penultimate test combines two different and unrelated computations. The last one finds all the solutions to the four-queens problem.

```
(fib 10)                                 ;;fib(10)
(qsort '(1 2 3 4 5 6 7 8 9 10))          ;;qsort(1..10)
(qsort '(5 3 9 7 1 8 10 2 6 4))          ;;qsort(random)
(qsort (qsort '(5 3 9 7 1 8 10 2 6 4))   ;;qsort(qsort)
(pcall cons                              ;;fib//qsort
       (fib 10)
       (qsort '(5 3 9 7 1 8 10 2 6 4)) )
(queens 4)                               ;;queens4
```

Table 1: Benchmark Suite: Tests

3.1 Raw Results

Table 2 contains the raw timings of the above benchmarks. The *Ticks* number is the time the machine took to evaluate one particular benchmark form. Ticks start when the zero-th processor is requested to evaluate the expression and end when processor zero receive the answer. At both time all other processors are idle. We will later examine times taken by individual processors.

Machine	*Migration*	Ticks					
		fib(10)	qsort(1..10)	qsort(random)	qsort(qsort)	fib//qsort	queens4
single		3638	5029	3450	8456	7092	13038
linear4	flow	3812	7712	3345	11021	6893	11732
linear16	flow	2495	6987	3298	10249	4902	10251
hypertorus4*4	flow	1908	7281	2516	9761	4334	5549
hypertorus4*4	dilute	1945	6290	2412	8783	4226	7250
beehive2*8	dilute	1903	5483	2522	9692	4173	6789

Table 2: benchmark raw results

The Fibonacci benchmark shows speedups of nearly 50%. Since non trivial computations of `fib` results in two new calls to `fib`, to have at least two neighbours is winning. Since only small integers are transfered, the migration policy does not have a real impact.

[3] The final paper will contain a test using `call/cc`.

The quasi-sequentiality of our version of `qsort` does not allow to exhibit enthusiastic speedups. When sorting an already sorted list, the speedup is merely a speed-down ! The speedup is better with a random list but does not exceed 25%. The figures for “qsort(qsort)” are close to the sum of the figures for “qsort(random)” and “qsort(1..10)” except for the beehive topology which entangled itself !

When combining two unrelated computations, the total time for “fib//qsort” is lesser than the sum of the times for “fib(10)” and “qsort(random)”. Speedups are not so good (except for the linear16 machine) due to a lot of contention on the initial processor and its direct neighbours. This is normal since the first computation is always introduced on the zero-th processor. It would be probably more effective to enter the machines with various entry points: for instance and for these planar machines, each user might have a different corner and a personal migration policy.

3.2 Analysis of Remote Applications and Migration

We instrumented our simulator to keep track of the nature of the various events it handles. We distinguish:

evaluation: a form, a constant or a reference is to be evaluated,

resumption: a value (or a list of values) is returned to the continuation,

lookup: an identifier is looked up in the current lexical environment,

remote-application: `remote-funcall` is applied,

remote-computation: one of the above computation has to be transfered to another site.

The number of remote-computations represents the number of computations remote applications induced.

The figures obtained from “fib(10)” appear in table 3. The number of evaluation is of course constant since the work to be done is always the same. The other figures tremendously increase as `remote-funcall` is called. Some of them can be optimized and mainly the number of lookup. We took the hypothesis that a processor ignores the global environment (except the initial processor) therefore, to know the value of `+`, `-` or `>=`, the lookup process induces many remote-computations to reach the initial processor, get the bound value: a primitive, and then come back with that primitive. The way back is just a value transfer and thus belongs to the resumption class. With this hypothesis our figures represent a sort of worst case which may only be improved.

Machine	*Migration*	Taxonomy of computations for “fib(10)”				
		evaluation	resumption	lookup	remote-application	remote-computation
single		1467	1249	705	0	0
linear4	flow	1467	3472	2820	108	4338
linear16	flow	1467	3472	2820	108	4338
hypertorus4*4	flow	1467	3472	2820	108	4338
hypertorus4*4	dilute	1467	3200	2548	108	3794
beehive2*8	dilute	1467	2632	1980	108	2658

Table 3: Taxonomy of computations for “fib(10)”

Topology and migration policy affect the speed. The “dilute” policy has no preferred migration directions, therefore some computations go back to the initial processor, shorten the lookup distance and decrease the number of remote computations and resumptions. Interestingly enough, the beehive topology only offers three neighbours and looks more localized than the hypertorus4*4 but still provides the same speedup level.

The situation is somewhat different with “qsort(random)”. The figures appear in table 4. The sequential bottleneck of `separate` and `merge` is visible since, for the same number of `remote-funcall`, there is much less induced remote computations. That is why the tighter topology, beehive2*8, performs well. Observe that the difference between the “dilute” and “flow” migration policy is still visible.

Three points are worth to be observed.

Machine	*Migration*	taxonomy of computations for “qsort(random)”				
		evaluation	resumption	lookup	remote-application	remote-computation
single		1385	1198	826	0	0
linear4	flow	1385	2053	1575	106	1690
linear16	flow	1385	2053	1575	106	1690
hypertorus4*4	flow	1385	2053	1575	106	1690
hypertorus4*4	dilute	1385	1751	1273	106	1086
beehive2*8	dilute	1385	1637	1159	106	858

Table 4: Taxonomy of computations in “qsort(random)”

- The total number of computational steps performed by all the processors of a machine is immense compared to the sequential number of steps for a single machine. But when these steps are disseminated on a large number of processors, the machine still offers some speedup. This validate a posteriori the fact that objects can be distantly pointed as well as the geographical commutation rule.
- The cost is mainly due to distant lookup and remote computations which must be improved in a real compiler. Mutability analyses as well as sharing analyses would be benefitful.
- Very naïve strategies such as “dilute” perform well and seem sufficient to spread computations all over the machine processors.

3.3 Linguistical Choices Revisited

We presented Crystal Scheme with implicit `pcall` and implicit `remote-funcall`. We made some simulations where we change the semantics of the application to always be an effective remote application. There is a neat decrease in performance since all immediate computations like addition, comparison or slot reading take more time to be transferred than to be locally performed. Table 5 makes appreciate the vast speed-down that may be incurred if badly annotating code.

Machine	*Migration*	“fib//qsort”	
		ticks	speed-down
single		11137	57 %
linear4	flow	23960	248 %
linear16	flow	10967	124 %
hypertorus4*4	flow	12965	199 %
beehive2*8	dilute	8631	107 %

Table 5: Systematic remote applications

It thus seems natural to limit migration in order to improve the overall behaviour. Since the computation of `fib` and `merge` both wait for two important subcomputations, one of them can be done locally. We thus run the benchmark suite after removing one of the two imperative `remote-funcall` appearing in `fib` and `merge`. If we exclude the linear4 machine which clearly does not belong to the class of massively parallel machines, it apparently seems uninteresting to remove one of the two `remote-funcall`. Half of the time this new version runs slower than the previous one, see table 6. The reason is that the critical computation path is not shortened even if less remote computations are induced. But there is a neat advantage which is to drastically decrease the load average of the machine as can be seen in table 7. The decrease is mostly apparent with “fib(10)” where it is near 50%. The conclusion is that the compiler must not starve a processor just to wait remote applications.

For the same reason it does not seem useful to explicit `pcall`. The compiler must carry the burden of correctly annotating code. We therefore decide to evaluate expressions sequentially skipping remote

Machine	*Migration*	"fib(10)"		"qsort(random)"		"fib//qsort"	
		ticks	speedup	ticks	speedup	ticks	speedup
single		3368	7 %	3400	1%	6772	5 %
linear4	flow	2489	35%	2728	18%	4461	35 %
linear16	flow	2489	0 %	2674	19%	4156	15 %
hypertorus4*4	flow	1731	9 %	2658	-6%	4230	2 %
hypertorus4*4	dilute	1827	6 %	2487	-3%	4338	-3 %
beehive2*8	dilute	1968	-3%	2582	-2%	4405	-6 %

Table 6: Benchmarks with only one `remote-funcall`

Machine	*Migration*	"fib(10)"		"qsort(random)"		"fib//qsort"	
		two	one	two	one	two	one
single		1.00/1	1.00/1	1,00/1	1,00/1	1,00/1	1,00/1
linear4	flow	3.23/4	2,46/4	2,04/4	1,92/4	2,79/4	2,55/4
linear16	flow	4,93/16	2,46/16	2,07/16	1,96/16	3,90/16	2,74/16
hypertorus4*4	flow	6,45/16	3,54/16	2,71/16	1,98/16	4,41/16	2,69/16
hypertorus4*4	dilute	5,77/16	3,21/16	2,33/16	2,11/16	3,93/16	2,45/16
beehive2*8	dilute	4,70/16	2,73/16	2,05/16	1,97/16	3,47/16	2,29/16

Table 7: Load Average with two or one `remote-funcall` (active/total processors)

applications. When an application is sent to another site, simulations show that it is better to perform it immediately (lifo) rather than just enqueueing it in the list of active processes (fifo). As shown in table 8, the only acknowledgeable trend is that 'lifo' is winning. Observe also that pure 'lifo' is roughly similar to sequential mode. The advantage of performing immediately remote computations is to shorten the critical evaluation path. To delay, for instance, a remote lookup may block its originator but induces the same amount of work on the current processor.

Machine	*Migration*	"fib(10)"		"fib//qsort"	
		ticks	speed-down	ticks	speed-down
single		3638	0 %	7092	0%
linear4	flow	5114	34 %	7807	13%
linear16	flow	3522	41 %	5485	12%
hypertorus4*4	flow	2061	8 %	4648	7%
hypertorus4*4	dilute	2134	10 %	4589	9%
beehive2*8	dilute	1929	1 %	4580	10%

Table 8: FIFO Scheduling of remote computations

To sum up, both `pcall` and `remote-funcall` gain to be implicit but the compiler must carefully annotate the generated code to limit superfluous migrations.

3.4 Load Analysis

One of our prior requirement was that the language must allow to express sufficient parallelism to utilize the machine. Only "fib(10)" can meet this requirement since it spawns 177 processes while the various `qsort` do not exceed 10 of them. We already gave some load average figures in table 7. But the load average of individual processor is far from uniform. Since the global environment is only present on the zero-th processor one should expect that the bottleneck is there and on its immediate neighbours. Table 9 shows the individual load average when computing "fib(10)" on hypertorus4*4[dilute]. The initial processor works nearly all the time (98%) due to the number of lookup it has to satisfy.

Processor Ids			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Busy percentage			
98%	81%	50%	30%
16%	11%	10%	8%
18%	25%	26%	9%
60%	64%	44%	27%

Table 9: Individual Load on "fib(10)" with hypertorus4*4[dilute]

Machine	*Migration*	Ticks		
		fib(10)	qsort(random)	fib//qsort
single		1831	1826	3606
linear4	flow	1977	2028	3737
linear16	flow	1324	2004	2762
hypertorus4*4	flow	1065	1601	2410
hypertorus4*4	dilute	992	1542	2484
beehive2*8	dilute	950	1638	2479

Table 10: Benchmark raw results with two-processor nodes

To alleviate this important load, one may suggest to consider that each processor is in fact a two-processor-node which share a same memory. Table 10 contains the result of running again the benchmarks with two-processor-nodes. Times are approximatively divided by 50%. Results are even better if we use three-processor nodes ... We can conclude to the superiority of the shared memory model which offers greater efficiency. Nonetheless distributed machines can handle much more wider problems and you can observe on "fib//qsort" that the speedup between the single machine and hypertorus or beehive naturally decreases. With sixteen-processor nodes, "fib//qsort" will take nearly the same time on every machines we consider here.

4 Future Works

One of the main virtue of Scheme is its automatic memory management. Massively parallel machines need to have a Garbage Collector. Two kinds of entities can be scavenged: simple data and processes. Distributed GCs exist for data but they generally require to stop the whole machine which is probably not a good thing for such a machine. At first sight, it seems useless to collect processes since the basic model of concurrency adheres to the fork-join model: any process is useful. But the semantics we gave to continuations, exemplified by the ability to program futures, may lead to useless processes. There again exist distributed GCs which also require to stop the whole machine.

To offer implicit parallelism helps programmers in a way but forces them to think more about race conditions. It would be interesting to warn, at least, the user in case some variables might concurrently be read or written. This study belongs to the abstract interpretation domain but does not seem (at least for our knowledge) to be already done for the Scheme framework.

In order for Crystal Scheme to become a real language, two other problems must be addressed: exception handling and task control. The lack of the latter prevents us to write the classical *parallel-or* construct which, when a disjunct returns true, suspends the computation of the useless others. It is our thought that task control should be made orthogonal to continuations maybe with a sponsor-like model [Osborne 90].

We also intend to develop a real compiler for Crystal Scheme and to inquire code generation techniques taking into account that neighbour processors having direct links can probably be tightly synchronized with low overhead. Afterthat, we will wait for a machine to come.

5 Related Works

The pionneer work of Halstead [Halstead 84] popularized the `future` concept. Since that, all implementations of `future` have been done on shared memory processors. A priori futures introduce more parallelism than `pcall` since they immediately return a place-holder that can be handled but for its content. Futures raise semantical problems when multiply determined [Katz & Weise 90], moreover the repairing machinery seems to slow down processes. Futures also offer some efficiency problem since any strict primitive has to touch its arguments before execution. All these problems may be alleviated by clever compiling techniques.

We show how to achieve futures[4] with our simpler assumptions and explicitely address the problem of distributed memories and data migration. Moreover our approach is semantically simpler.

A description of a Concurrent Scheme for distributed memories appears in [Kessler & Swanson 90]. They introduce a `future`-like mechanism called `make-thread`. Mutual exclusion is handled through domains: any computation is performed in a domain and blocks all computations of the same domain until it finishes. Domains define disconnected data subsets: no data sharing is possible between different domains. Data are thus copied before being passed from one domain to another. Domains are good candidates to be associated to processors and `make-domain` is a kind of `remote-funcall`.

Our approach differs on a variety of points. First we do not introduce so many concepts and prefer to stick to the global value space of Scheme and thus avoid copying between processors. This raises race condition but it is unclear if copying semantics is cleaner.

A parallel abstract machine was proposed in [Giorgi & Le Métayer 90] where a `remote-funcall` facility was already present. Since their language is purely functional they do not handle assignment, nor data mutation nor first-class continuations that we have shown to be valuable tools. The current version of this machine only handles booleans and small integers while we handle any kind of Scheme objects. Like us the semantics of the application might involve a remote computation.

A study of a massively parallel machine appears in [Germain & Giavitto & Sansonnet 91]. They cover a number of points of the hardware side. They also give very accurate figures since they define an assembly language for the machines they simulate. Parallelism is explicit. When a function is defined, the user must indicate its execution mode. A run-time choice between concurrent (local) or parallel (remote) allows to load balance the processes. They do not address semantical issues on continuations nor they handle compound data like pairs or vectors.

6 Conclusion

We presented an extension of the Scheme programming language oriented to massively parallel machines with independent memories. We follow the philosophy of Scheme and looked for very simple means to benefit from such machines. We finally retain that each application might have its terms concurrently evaluated and might be remotely performed. We also choose an exchange semantics for the assignment. Finally we do not change the syntax of Scheme since we do not add special forms but only alter its semantics to introduce concurrency and migration. We then compare the sequential and parallel modes of computations and exhibit some examples of very peculiar behaviour. Yet we think that these behaviours are understandable in the sole terms of concurrency rather than depending on the hidden implementation of high level constructs.

The meaning of continuations have been changed to accomodate concurrency and later on revealed their interest when programming high level constructs such as futures. It is an amazing result that such a little kernel can express such complex constructs.

The simulations we made show that the behaviour of Crystal Scheme is not ridiculous but still remains to be improved. Distributed memories induce tremendous migrations but remote computations can benefit from the numerous idle processors and still offer global speedup. The main problem is that the kind of parallelism we offer is very sensitive to the length of the critical evaluation path and therefore

[4] Since futures can be programmed in Crystal Scheme, we plan to experiment them.

often presents some bottlenecks. Nonetheless it is surprising that so much concurrency can be observed even when handling symbolic data structures such as lists.

7 Acknowledgment

We are thankful to Jean-Paul Sansonnet whose views on massively parallel machines and his magic words on *computational explosion* triggered this work.

Bibliography

[Béchennec89] J-L Béchennec, *MegaPack: a 3D Packaging for Massively Parallel Computers*, LRI-Archi TR 89-07, 1989.

[Chien & Dally 89] Andrew A. Chien, William J. Dally, *Concurrent Aggregates*, Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming PPOPP, Seattle, March 1990, pp 187–196.

[Dally *et al.* 89] William J. Dally et al., *The J-Machine: A Fine-Grain Concurrent Computer*, Proceedings of the IFIPS Conference 1989.

[Deutsch90] Alain Deutsch, *On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications*, 16th ACM Symposium on Principles of Programming Languages, San Francisco, January 1990.

[Germain & Giavitto & Sansonnet 91] Cécile Germain, Jean-Louis Giavitto, jean-Paul Sansonnet, *Implémentation d'un Paradigme de Programmation Fonctionnelle sur une Machine Massivement Parallèle*, Journées Francophones des Langages Applicatifs, Grès-en-Vercors, France, Janvier 1991.

[Germain *et al.* 90] C. Germain, J-L. Béchennec, D. Etiemble, J-P. Sansonnet, *An interconnection network and a routing scheme for massively parallel message-passing computers*, Third Symposium on Frontiers of Massively Parallel Computation, Washington 1990.

[Giorgi & Le Métayer 90] J-F Giorgi, Daniel Le Métayer, *Continuation-Based Parallel Implementation of Functional Programming Languages*, 1990 ACM Conference on Lisp and Functional Programming, Nice, France, pp 209–217.

[Halstead 84] Robert H. Halstead, Jr., *Implementation of MultiLisp: Lisp on a Multiprocessor*, 1984 Symposium on Lisp and Functional Programming, Austin, Texas, 1984, pp 9–17.

[Halstead 85] Robert H. Halstead, Jr., *Multilisp: A Language for Concurrent Symbolic Computations*, ACM Trans. on Prog. Languages and Systems, October 1985, pp 501–538.

[Halstead 90] Robert H. Halstead, *New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools*, US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989, Lecture Note on Computer Science 441, Springer-Verlag, 1990.

[Hewitt 80] Carl Hewitt, *The Apiary Network Architecture for Knowledgeable Systems*, Conference Record of the 1980 Lisp Conference, pp 107–118.

[Katz & Weise 90] Morry Katz, Daniel Weise, *Continuing Into the Future: On the Interaction of Futures and First-Class Continuations*, 1990 ACM Conference on Lisp and Functional Programming, Nice, France, pp 176–184.

[Kessler & Swanson 90] Robert R. Kessler, Mark R. Swanson, *Concurrent Scheme*, US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989, Lecture Note on Computer Science 441, Springer-Verlag, 1990.

[Kohlbecker *et al.* 86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, Bruce Duba, *Hygienic Macro Expansion*, Proceedings of 1986 ACM Conference on Lisp and Functional Programming, pp 151–161, ACM Press, New York, 1986.

[Mitsolides & Harrison 90] Thanasis Mitsolides, Malcolm Harrison, *Generators and the Replicator Control Structure in the Parallel Environment of ALLOY*, ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York, 1990, pp 189–196.

[Mohr & Kranz & Halstead 90] Eric Mohr, David A. Kranz, Robert H. Halstead, Jr., *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*, 1990 ACM Conference on Lisp and Functional Programming, Nice, France, pp 185–198.

[Osborne 90] Randy B. Osborne, *Speculative Computation in Multilisp, An overview*, 1990 ACM Conference on Lisp and Functional Programming, Nice, France, pp 198–208.

[Queinnec 90] Christian Queinnec, *PolyScheme, a Semantics for a Concurrent Scheme* High Performance and Parallel Computing in Lisp Workshop, Twickenham, England, November 1990.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[Stamos & Gifford 90] James W. Stamos, David K. Gifford, *Implementing Remote Evaluation*, IEEE Trans. on Software Engineering, Volume 16, Number 7, July 1990, pp 710–722.

[Wand 80] Mitchell Wand, *Continuation-based Multiprocessing,* Conference Record of the 1980 LISP Conference.

Appendix: Benchmark Suite: Functions Definitions

```
(set! fib (lambda (n)
              (if (<= n 2) 1
                  (pcall + (remote-funcall fib (- n 1))
                           (remote-funcall fib (- n 2)) ) ) ) )
(set! qsort (lambda (l)
              (if (null? l) l
                  (merge (separate (car l) (cdr l) '() '())) ) ) )
(set! separate
      (lambda (pivot l lesser greater)
        (if (pair? l)
            (if (< (car l) pivot)
                (separate pivot (cdr l) (cons (car l) lesser) greater )
                (separate pivot (cdr l) lesser (cons (car l) greater) ) )
            (cons lesser (cons pivot greater)) ) ) )
(set! merge
      (lambda (sorted)
        (pcall append
               (remote-funcall qsort (car sorted))
               (cons (car (cdr sorted))
                     (remote-funcall qsort (cdr (cdr sorted))) ) ) ) )
(set! queens (lambda (n)
     (search n 0 '() (iota 0 n)) )
(set! iota (lambda (start end)
             (if (< start end)
                 (cons start (iota (+ start 1) end))
                 '() ) ) )
(set! search (lambda (n p s l)
  (if (= n p)
      (list s)
      (foreach (lambda (i)
                 (if (check p i s)
                     (search n (+ 1 p) (cons i s) l)
                     '() ) )
              l ) ) ))
(set! check (lambda (p i s)
  (or (null? s)
      (and (not (memq i s))     ;check line
           (check-others (- i 1) (+ i 1) s) ) ) ))
(set! check-others (lambda (i j s)
  (or (null? s)                 ;check diagonals
```

```
      (and (not (= i (car s)))
           (not (= j (car s)))
           (check-others (- i 1) (+ j 1) (cdr s)) ) ) ))
(set! foreach
      (lambda (f l)
          (if (pair? l)
              (pcall append (remote-apply f (car l))
                            (foreach f (cdr l)) )
              '() ) ) )
```