Using Data Parallelism in Lisp for Implementing a Quantum Simulator

Leonardo Uribe, Pascal Costanza, Charlotte Herzeel, Theo D'Hondt

Programming Technology Lab
Vrije Universiteit Brussel
{leonardo.uribe|pascal.costanza|charlotte.herzeel|tjdhondt}@vub.ac.be

Abstract. This paper describes two implementations of QLisp (a Lisp extension to simulate quantum computations) in two different data-parallel extensions of Lisp: *Lisp and Paralation Lisp. First, the basic concepts of the languages and the quantum simulator are explained. Then, the porting process is described and a comparative evaluation of the two implementations is made. It is shown that data parallel languages are well suited for parallelizing QLisp. Also, in the porting process, we discovered some non-obvious differences between the different data parallel languages.

1 Introduction

The recent advent of multi-core architectures promises significant increases in computing power, but comes with the cost of increased complexity with regard to programmability. Parallel programming allows the programmer to distribute code on different processors. There are two main forms of parallelism: data-parallelism and task-parallelism. Data-parallelism emphasizes on the distribution of data as the key to get parallel processing. Task-parallelism focuses on distributing execution processes. However, parallel programming is not a new invention, but a respected research field whose popularity peaked in the 1980's, especially when it comes to language research. For example, several dialects of Lisp were developed for programming the infamous Connection Machine, like Connection Machine Lisp, *Lisp and Paralation Lisp. Given the current multi-core hype, we believe it is worthwhile to revisit the concepts introduced by these languages. In this paper, we present a comparison between *Lisp and Paralation Lisp. This comparison is based on our own experiences in using *Lisp and Paralation Lisp to re-implement QLisp, an existing simulator for quantum computers.

This paper is structured as follows: Section 2 introduces the major Lisp extensions for data-parallel programming: *Lisp, Connection Machine Lisp and Paralation Lisp. Sections 3 gives a short introduction to QLisp. Section 4 describes how it was implemented in *Lisp and Paralation Lisp. In Section 5 a comparison of the two implementations of Qlisp is made. A final section presents the conclusions.

2 Data Parallel Programming in Lisp

2.1 Connection Machine Lisp

Connection Machine Lisp (CM-Lisp), a data-parallel extension of Common Lisp, was invented by Daniel Hillis and Guy Steele and was intended to be used for programming symbolic AI applications on the Connection Machine [1]. The Connection Machine is a highly parallel supercomputer organized using a multidimensional cubic grid structure with one processor on each vertex. It has thousands of simple processors, each one having a local memory. Programming a Connection Machine is similar to programming a single processor that can receive multiple data in parallel and apply a single instruction to all of these data at the same time. CM-Lisp has special tools to describe instructions operating in parallel over multiple data. Though CM-Lisp was never fully implemented, it introduces a number of important concepts and influenced later languages like *Lisp and Paralation Lisp.

CM-Lisp introduces a new data structure called `xapping` and some additional instructions to introduce parallelism. A `xapping` (which refers to a kind of 'mapping') is an unordered set of ordered pairs. The first element of the pair is called the 'index' and the second the 'value'. As in a set representation of a mathematical mapping, no two pairs in a `xapping` can have the exact same indexes. The following is an example of a simple `xapping`: $\{a\text{->}4\ c\text{->}25\ b\text{->}9\}$. In terms of implementation on a parallel computer, the indexes can represent labels for different processors and each value represents a datum associated to the corresponding processor. If the indexes are the first n natural numbers, the `xapping` becomes a `xector` (in analogy to a vector), which can be represented as [4 25 9] if the indexes (a, b, c) are changed by 1, 2 and 3 in the last `xapping`.

Parallelism is introduced by applying a function concurrently to all elements of a `xapping` using the 'apply-to-all' instruction: When a function is applied to a `xapping` using the 'apply-to-all' instruction, the result is a new `xapping` whose elements are the result of applying the function to each of the elements. The following code illustrates the use of `xectors` and the 'apply-to-all' instruction:

```
1   (defvar a '[4 25 9])
2   [4 25 9]
3   α (+ · a 1)
4   =>[5 26 10]
```

In the first line, a `xector` is created. Line 3 implements a parallel sum: The 'α' symbol indicates that a parallel operation is to be done. The bullet prefixed to the name of the `xector` ($\cdot$) indicates that the Lisp form that follows should be executed once (element by element), instead of several times in parallel. The result of the operation is shown on line 4. A kind of 'distributive law' can be observed for α's and $\cdot$'s: The expression in line 3 can be re-expressed as α (+ $\cdot$a 1) = (α a α1), where α has been distributed and has being canceled when multiplied by $\cdot$. The new expression can be interpreted as: "Make a parallel sum between the elements of `xector` 'a' and a `xector` of 1's". This kind of notation is similar to the backquote notation of Common Lisp, where the backquote quotes

everything, except the places where there are commas. Connection Machine Lisp has other constructs to manipulate `xappings` and functions to generate, for instance, composed functions or to allow some elements of a `xapping` to remain unaltered when an "apply-to-all" instruction is executed. With these tools, lots of complex `xapping` transformations can be expressed [1].

2.2 *Lisp

*Lisp (pronounced "StarLisp") was created in 1985 by Cliff Lasser and Steve Omohundro as a high-level language for the Connection Machine [2]. For our experiments, we have made use of J.P. Massar's *Lisp simulator [1]. *Lisp is an extension of Common Lisp and introduces a structure called `Pvar` (parallel variable) -essentially a vector- as its basic data-parallel structure. `Pvars` are a representation of multiple processors. Each element in a `Pvar` represents an entry in a processor's memory. The programmer can do standard operations on `Pvars`, like addition, multiplication, elements reorder, and so on, but it also has primitives for communicating between `Pvars`. *Lisp is well suited for massive parallel computing related to homogeneously composed systems, for instance particle animations [3]. The code below shows the creation of two `Pvars`, followed by an expression that adds both [2].

```
1> (*cold-boot :initial-dimensions '(16 16))
256
(16 16)
2> (defpvar lots-of-sevens 7)
LOTS-OF-SEVENS
3> (defpvar lots-of-threes 3)
LOTS-OF-THREES
4> (ppp lots-of-sevens :mode :grid :end (4 4))
DIMENSION 0 (X) ----->
7 7 7 7
7 7 7 7
7 7 7 7
7 7 7 7
5> (ppp (+ lots-of-sevens lots-of-threes) :mode :grid :end (4 4))
DIMENSION 0 (X) ----->
10 10 10 10
10 10 10 10
10 10 10 10
10 10 10 10
```

The first instruction defines the default size of the `Pvars` to be used in the code. This starts the Connection Machine with 256 processors, arranged in a square pattern of size 16 by 16. The next two instructions generate two `Pvars` called "lots-of-sevens" and "lots-of-threes" initialized to 7 and 3 respectively. The `ppp` function implements a pretty-printer for `Pvars`. It allows showing a whole `Pvar` or just a part of it. In the final instruction, a sum operation is realized between the created `Pvars`. This task generates as many parallel tasks as there are elements in a `Pvar`.

[1] See http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/impl/starlisp/0.html

2.3 Paralation Lisp

Paralation Lisp is an implementation of the Paralation Model (a contraction for PARAllel reLATION Model) described by Gary W. Sabot in his book 'The Paralation Model' [4]. For simulation purposes, Sabot's book describes a compiler for paralation Lisp to Common Lisp

In the Paralation Model, the basic data structure is called a "field": It resembles an array of objects. Fields are grouped in 'paralations', by which the concept of locality or nearness between fields is introduced. The paralation model has three basic operators: `elwise`, `move` and `match`.

The 'elementwise' evaluation operator `elwise` allows the execution of a program instruction in every site of a paralation at the same time: This is the standard way to introduce parallelism. The communication process is steered using the `move` operator, which moves data from one paralation to another, using a 'mapping' as a guide. Such a 'mapping' can be generated by the `match` operator, which is applied over two fields and returns an abstract communication pattern based on the concrete fields' elements. This mapping is applicable to any pair of fields having the same dimension as the originals. Using the three basic operators in combination with the paralation data structure, it is possible to define higher-level operators, and as such construct a complete parallel programming language. The Paralation model was embedded in Common Lisp, and the resulting dialect was dubbed 'Paralation Lisp'. The following example shows the basic usage of the key operators in the language:

```
1   (setq f (make-paralation 4))
2   => (0 1 2 3)
3   (setq g (make-paralation 2))
4   => (0 1)
5   (setq f1 (elwise (f) (elt '(0 0 1 1) f)))
6   => (0 0 1 1)
7   (<- f   :by (match g f1)
8           :with #'+)
9   => (1 5)
```

In the first line, a paralation is created and a field with the index for each entry is returned. The third line generates another paralation and again such an 'index field' is returned. In the fifth line, the `elwise` operator is used to make a parallel assignment to the values of a field. The field `f` acts here as the index position for the `elt` function. The seventh and eighth lines show the use of `move` (<-): The entries of `f` are moved using a mapping generated by the `match` operator. If more than one values in the field `f1` have the same 'image', they will be reduced to one with the `+` operator. The final result is given in line nine.

3 Quantum simulation

3.1 Basic Concepts of Quantum Computation

In the beginning of the 1980's, the well known physicist Richard Feynman realized that there were some quantum physics phenomena that cannot be simulated

by a standard digital computer, but suggested that computation can be greatly improved by taking advantage of those phenomena.

In 1985, David Deutsch – also a physicist – described a first model for a quantum computer, something similar to the Turing machine model proposed in 1936 for the digital computer. The interest in quantum computers increased significatively in 1994 when Peter Shor, from AT&T, described a quantum algorithm for efficient prime factorization, a useful tool to decrypt RSA[2] encrypted messages.

In quantum computation, the basic unit of information is a quantum bit (`qubit`). This quantum bit can be in the two classical states (0 and 1) at the same time, but once it is observed it collapses to a unique state. A `qubit` can be mathematically represented as a superimposition of classical states. The probability to get one of these states when the `qubit` is observed is represented as a coefficient. In the formula below, the coefficients v_0 and v_1 represent these probabilities.

$$v = v_0|0\rangle + v_1|1\rangle$$

A new quantum register can be constructed by applying the tensor product to two (or more) existing `qubits`. A 2-`qubit` `qureg` is represented as a superimposition of all the possible states of two classical bits and so on:

$$v = v_{00}|00\rangle + v_{01}|01\rangle + v_{10}|10\rangle + v_{11}|11\rangle$$

The fact that a `qureg` can be in multiple classical states at the same time makes the quantum computation inherently parallel. As the number of `qubits` in a `qureg` grows linearly, its classical representation grows exponentially. For instance, a 5-`qubit` `qureg` needs $2^5 = 32$ coefficients to be represented in a classical computer. In this resides the power of quantum computers and also its difficulty to be simulated on a classical machine. A more useful representation of `quregs` is a vector representation:

$$\mathbf{v} = \begin{pmatrix} v_{00} \\ v_{01} \\ v_{10} \\ v_{11} \end{pmatrix}$$

This representation is well suited to apply quantum operators (`qop`) to a `qureg`. A quantum operator is represented as a squared matrix acting over the vector representation of a `qureg`. The entries for such a matrix belong to the field of complex numbers. This mathematical representation shows the fact that a quantum operator is a linear transformation of a `qureg`, a much more general operation than the equivalent classical transformation, which is restricted to all the permutations between bits in a register. A `qop` acting over a `qureg` can be represented as follows:

[2] In cryptography, RSA is an algorithm for public-key cryptography. See: http://en.wikipedia.org/wiki/RSA

$$\mathbf{Xv} = \begin{pmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} v'_0 \\ v'_1 \\ v'_2 \\ v'_3 \end{pmatrix}$$

The tensor product is the mathematical operator used to build bigger `quregs` and `qops` from simpler ones[3]. The example below shows how the tensor product between two `qops` is computed.

$$\mathbf{t} = \begin{pmatrix} t_0 \\ t_1 \end{pmatrix}, \mathbf{u} = \begin{pmatrix} u_0 \\ u_1 \end{pmatrix}, \mathbf{t} \otimes \mathbf{u} = \begin{pmatrix} t_0 u_0 \\ t_0 u_1 \\ t_1 u_0 \\ t_1 u_1 \end{pmatrix}$$

$$\mathbf{Y} = \begin{pmatrix} y_0 & y_2 \\ y_1 & y_3 \end{pmatrix}, \mathbf{Z} = \begin{pmatrix} z_0 & z_2 \\ z_1 & z_3 \end{pmatrix}, \mathbf{Y} \otimes \mathbf{Z} = \begin{pmatrix} y_0 z_0 & y_0 z_2 & y_2 z_0 & y_2 z_2 \\ y_0 z_1 & y_0 z_3 & y_2 z_1 & y_2 z_3 \\ y_1 z_0 & y_1 z_2 & y_3 z_0 & y_3 z_2 \\ y_1 z_1 & y_1 z_3 & y_3 z_1 & y_3 z_3 \end{pmatrix}$$

3.2 QLisp

QLisp is a language extension to Common Lisp to simulate quantum computations. It uses the mathematical model of quantum mechanics to simulate quantum phenomena, which is a different approach than the so called 'reality-based simulation', which tries to imitate the real world facts as accurately as possible[5]. Because QLisp uses a mathematical model for simulating quantum computations, the `quregs` can be observed without disturbance, something that is not possible for a real `qureg` due to the laws of quantum physics. This fact allows some flexibility in the calculations, makes some complex quantum operations easier and makes QLisp well suited for educational purposes.

QLisp implements two optimizations in order to simplify and speed up its execution: Sparse matrices and single operator application. Sparse matrices were introduced based on the fact that the quantum mathematical model uses big matrices and vectors whose entries are, generally speaking, mostly zeros. Representing such matrices as hash tables including only the non-zero values allows QLisp to make certain operations faster. Secondly, "single operator application" optimizes how a single operator (an operator acting over only one `qubit`) needs to be applied to a multiple-`qubit` `qureg`. Instead of creating a big sparse operator, QLisp uses a special operator that acts only over the `qubit` of interest in the `qureg`. As an example of QLisp code, the implementation of the Deutsch-Jozsa algorithm is shown below:

```
1  (defun deutsch-jozsa (n unitary-fn)
2    "returns T if unitary-fn is constant"
3    (let* ((_phi1_ (make-qureg n (hadamard-init)))
4           (_phi2_ (qc-apply
5                    (make-qureg 1 (standard-init 1)) (-h-)))
```

[3] For specific information about the tensor product, see for instance: http://en.wikipedia.org/wiki/Tensor_product

```
6           (_psi_ (funcall unitary-fn
7                           (tensor-items _phi1_ _phi2_))))
8      (constant-qureg-p
9       (collapse-basis
10       (qc-apply-range _psi_ -h- 0 (1- n))))))
```

This algorithm determines if a given function $f : 0,1^n \rightarrow 0,1$ is constant (returns always the same value) or balanced (returns 0 for half of the elements and 1 for the other half). It is assumed that the given function belongs to one of these two categories. The 'qc-apply' function in the fourth line represents the application of a `qop` (-h-) to a `qureg`. The function 'tensor-items' in the seventh line generates a `qureg` 'combining' the `quregs` _phi1_ and _phi2_, previously defined in lines 3 and 4. In the last line, 'qc-apply-range' is the repeated application (from 0 to n-1) of a single `qop` (-h-) to a multi-`qubit qureg`. Collapse-basis -line 9- simulates the process of collapse of the `qureg` _psi_ once it is observed. Constant-qureg-p -line 8- returns true if the observed `qureg` allows to conclude that the given function is constant and false if it is balanced. The QLisp source code frequently uses loops to implement quantum operations. These loops are perfect candidates for parallelization.

4 Re-implementing QLisp in Data-Parallel Languages

Qlisp is well suited to be ported to a data-parallel language because the mathematical model of quantum mechanics uses matrices, vectors and operations between them to simulate quantum operators, quantum bits and their mutual interactions. Nevertheless, the parallelization process involved a lot of code rewriting, because the original optimizations made in QLisp to represent matrices and vectors as hash tables is an optimization beneficial for a sequential computer, but is ignored in the new parallel code. This requires a change in the representation of structures and operators of QLisp.

In what follows, we present a more detailed discussion of our experiences in rewriting QLisp using *Lisp and Paralation Lisp. We will use the implementation of the tensor product as a running example. We initially restricted ourselves to these languages, because they were the only Lisp dialects for which we could find simulators (for *Lisp see: *http://examples.franz.com/index.html*, for Paralation Lisp see [4]).

As a point of reference, the naive Common Lisp implementation of the tensor product, without considering the Qlisp hash-table optimization, is shown in figure 1. `Qops` are represented as vectors. Observe in lines 6 and 9 the use of single loops and in lines 12-14 the use of two parallel loops. The replacement of that kind of loops by parallel computations is the main target of the parallelization process. The semantics of the code will be treated in more detail below.

4.1 QLisp in *Lisp

At the beginning of the *Lisp program, we declare the virtual layout of the Connection Machine's processors. Next we need to create the adequate `Pvars`

```
1  (defun tensor-qop (qop-1 qop-2)
2    (let* ((dim-1 (truncate (sqrt (length qop-1)) 1))
3           (dim-2 (truncate (sqrt (length qop-2)) 1))
4           (dim-12  (* dim-1 dim-2))
5           (dim-122 (* dim-12 dim-2))
6           (entry-1 (loop for self-address from 0 below  (expt dim-12 2)
7                          collect  (+ (* dim-1 (truncate self-address dim-122))
8                                      (truncate (mod self-address dim-12) dim-2))))
9           (entry-2 (loop for self-address from 0 below (expt dim-12 2)
10                         collect (+ (* dim-2 (truncate (mod self-address dim-122) dim-12))
11                                    (mod self-address dim-2)))))
12     (loop for i in entry-1
13           for j in entry-2
14           collect (* (elt qop-1 i) (elt qop-2 j)))))
```

Fig. 1. Tensor product implemented in Common Lisp

for representing `qops` and `quregs`. Internally, these are implemented as matrices. Figure 2 shows how to represent Qlisp's matrix data structures as Pvars, mapped onto the selected layout.

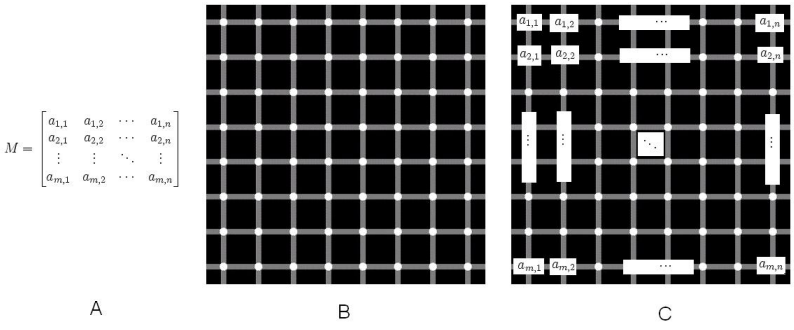

Fig. 2. Representation of a matrix in *Lisp. A: the matrix; B: Virtual processors layout; C: a `Pvar` representing the matrix

Figure 2A shows the matrix. Figure 2B shows the virtual processors layout we declare as default. Figure 2C shows how each entry of the matrix is stored in a particular processor's memory: Only this processor operates on that entry. This kind of `Pvar` is adequate to both represent `quregs` and `qops`. Those are the only structures needed to simulate quantum computations in QLisp. In order to parallelize the process, the programmer has to determine how each entry has to change depending on the function applied, its position on the `Pvar` and the value of the other entries. A particular example of a parallelization process is the tensor product between operators. Figure 3 shows the two operands and the result. The programmer has to build a function that, given the position of the

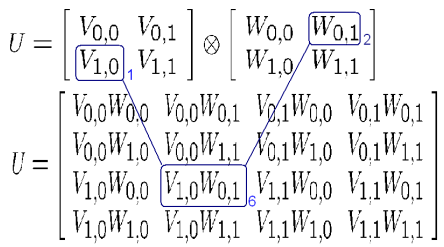

Fig. 3. Tensor product between two qops

entry in the resulting `Pvar` (6), produces the two different positions (1,2) of the related entries in the operands.

The whole function, written in *Lisp is:

```
1  (defun tensor-qop (qop-1 qop-2)
2    (let* ((dim-1   (expt 2 (qop-size qop-1)))
3           (dim-2   (expt 2 (qop-size qop-2)))
4           (dim-12  (* dim-1 dim-2))
5           (dim-122 (* dim-12 dim-2)))
6      (*let* ((entry-1 (+!! (*!! dim-1 (truncate!! (self-address!!) dim-122))
7                            (truncate!! (mod!! (self-address!!) dim-12) dim-2)))
8              (entry-2 (+!! (*!! dim-2 (truncate!! (mod!! (self-address!!) dim-122) dim-12))
9                            (mod!! (self-address!!) dim-2))))
10           (if!! (<!! (self-address!!) (expt (* dim-1 dim-2) 2))
11                 (*!! (pref!! qop-1 entry-1) (pref!! qop-2 entry-2))
12                 nil))))
```

Fig. 4. Tensor product implemented in *Lisp

The code in lines 6–7 defines a mapping that gives as result the required entry in the first operand (entry-1) as a function of the position of the entry in the resulting `Pvar`. Written as a mathematical formula, where 'entry3' represents an entry in the resulting `Pvar` it looks like this:

$$Entry1 = dim1 * (Entry3/(dim1 * (dim2)^2) + (Entry3 \ mod \ (dim1 * dim2))/dim2$$

In a similar way, the code in lines 8–9 defines the entry for the second operand. The double exclamation mark after an operator indicates a parallel operation acting over each entry of the `Pvars` given as arguments. For instance, the function `(self-address!!)` is used to determine the position of each processor in the `Pvar`. *Lisp has parallelized versions of almost all of the basic Common Lisp functions (for instance, look in the example for `if!!`, `mod!!`, `+!!`, etc). In the code, the division operator is replaced with `truncate!!`, which returns an integer instead of a float number. This has to be done in order to pass `entry-1` and `entry-2` as arguments of the function `pref!!` in line 15. That function returns

the value of a given entry in a given `Pvar`. The parallel modulo function (`mod!!`) is used to simulate the periodicity in the subindexes of the tensor-product result (see Figure 3). The main operation in the function definition is in line 11 where two entries of the two argument `Pvars` are multiplied. The communication between `Pvars` is implicit in this process: Each entry of the resulting `Pvar` communicates with entries in the argument `Pvars` by applying an operation on their values.

4.2 QLisp in Paralation Lisp

In this implementation of QLisp, `qops` and `quregs` are represented as fields. Different `qops` or `quregs` belong to different paralations. Each operation between those elements is done by combining communication between paralations (move) and processing in paralations (`elwise`). The code excerpt below shows the tensor product between two `qops`:

```
1  (map1 (elwise (index)
2                (+ (* side1 (truncate index side122))
3                   (truncate (mod index side12) side2))))
4     (map2 (elwise (index)
5                   (+ (* side2 (truncate (mod index side122) side12))
6                      (mod index side2))))
7     (multi1 (<- qfield1 :by (match map1 index1)))
8     (multi2 (<- qfield2 :by (match map2 index2)))
9     (result (elwise (multi1 multi2)
10                    (* multi1 multi2))))
```

Fig. 5. Tensor product implemented in Paralation Lisp

This chunk of code is in a `let*` form. Lines 1-3 generate the mapping that will be used to move data from the field representing the first `qop` operand to a field in the resultant paralation. For clearness, The mathematical formula representing this mapping is repeated below:

$$Entry1 = dim1 * (Entry3/(dim1 * (dim2)^2) + (Entry3 \ mod \ (dim1 * dim2))/dim2$$

Lines 4-6 generate another mapping to move data from the second `qop` operand to the resultant paralation. Then, lines 7 and 8 move the entries of the operand-`qops` to the resulting-`qop` paralation using the generated mappings. Lines 9 and 10 perform the multiplication of the generated fields to produce the field representing the tensor operation. Figure 6 is a graphical representation of the code we just explained.

5 Comparison of QLisp implementations

*Lisp has a limitation related to the size of the `Pvars` used in a program. The programmer has to define, as a global variable, a fixed default size of the `Pvars`

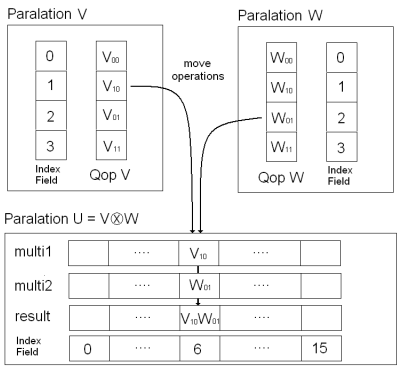

Fig. 6. Tensor product between two `qops`

he is going to create along the program. In some cases, this value can be changed easily as part of the code. However, operating on different `Pvars` with different sizes at the same time is not supported in *Lisp, but can only be simulated, which can be very difficult for the programmer to deal with. As a consequence, all `qureg` and `qop` were implemented over the same `Pvar` layout and dealing with unused entries was a tedious job.

In Paralation Lisp, the parallel operations also have to be done between same-sized fields. Nevertheless, you can have different-sized fields at the same time and use the move operator to adequately change their dimensions and perform operations between them (see Figure 6).

Analyzing Figures 3 and 6 and the related code, it can be seen that *Lisp does not make a clear difference between communication and computing processes. Operations and data-movement are done at the same time and it is not clear for the programmer how much communication between processors and how much computations are needed to run the code. In contrast, Paralation Lisp makes a clear separation of the two processes. The programmer is aware when a communication process takes place (move instruction) or when he carries out a computing process (`elwise` instruction). This fact is very important to make an estimate of the efficiency of a parallel program, given that the programmer knows the time costs of communication and processing for a given machine.

The way to express parallelism in *Lisp is simple and focuses on the operators. For instance, the instruction `(pref!! qop-1 entry-1)` tells the compiler to return a value from the `Pvar qop-1` referenced for an entry of the `Pvar entry-1`. The operator automatically assumes `qop-1` as a 'whole' `Pvar` and `entry-1` as the `Pvar` over which the parallel process has to be done element by element. In Paralation Lisp, the syntax is more complex and a similar instruction looks like: `(elwise (entry-1) (elt qop-1 entry-1))`.

Even though the *Lisp instruction is easer to write, the syntactic difference between the two languages hides a semantic difference: Paralation Lisp allows

nested parallel structures and *Lisp does not. For instance, imagine `qop-1` as a field of fields. Then the instruction `(elwise (entry-1 qop-1) (elt qop-1 entry-1))` makes sense and references elements belonging to the elements (subfields) of `qop-1`.

6 Summary/Conclusions

In the process of parallelizing Qlisp using *Lisp and Paralation Lisp, almost all of the code had to be rewritten. This is because QLisp represents matrices as hash tables, but in *Lisp and Paralation Lisp, in order to achieve the parallelization, matrices have to be represented as `Pvars` and fields respectively. The different representations are incompatible and part of the code written in QLisp had to be re-implemented in terms of the new data structures. Nevertheless, the quantum computation model is very well suited to be represented in data-parallel languages, and the parallelization process looks 'natural'. Paralation Lisp has some advantages over *Lisp: It is better to manage different-sized parallel-data-structures, it makes a clear separation between communication and computation concepts, and it allows for nested parallelism. As a future work, we will provide a 'shape facility' to Paralation Lisp, which will allow as to define matrix-shaped `Pvars` and port Qlisp in a more efficient way.

References

1. Steele, G., Hillis, D.: Connection Machine Lisp: Fine-grained Parallel Symbolic Processing. In: LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1986) 279–297
2. Meglicki, Z.: The CM5 *Lisp Course. Centre for Information Science Research - The Australian National University (January 1994)
3. Sims, K.: Particle Animation and Rendering Using Data Parallel Computation. In: SIGGRAPH '90: Proceedings of the 17th annual Conference on Computer Graphics and Interactive Techniques, New York, NY, USA, ACM (1990) 405–413
4. Sabot, G.W.: The Paralation Model: Architecture-Independent Parallel Programming. MIT Press, Cambridge, MA, USA (1989)
5. Desmet, B., D'Hondt, E., Costanza, P., D'Hondt, T.: Simulation of Quantum Computations in Lisp. 3rd European Lisp Workshop, co-located with ECOOP 2006 (July 2006)