Modelling Parallel Quantum Computing using Transactional Memory

Juliana Kaizer Vizzotto[1]

Centro Regional Sul-CRS
Instituto Nacional de Pesquisas Espaciais-INPE
Santa Maria, Brazil

André Rauber Du Bois[2]

Programa de Pós-Graduacão em Informática
Universidade Católica de Pelotas
Pelotas, Brazil

Abstract

We propose a model for parallel quantum computing in a single ensemble quantum computer using Haskell's software transaction memory. The parallel ensemble quantum computer possesses, besides quantum parallelism, a kind of classical single-instruction-multiple-data parallelism. It explores additional speedup by making quantum computers working in parallel, as in classical computation. The whole state is prepared in such a way a subset of qubits is in a mixed state representing the communicating quantum computers while the other qubits in pure state are the proper argument registers of each quantum computer. Essentially, this particular way of structuring the state of the parallel quantum computer fits with what is well know as multithreading programming. Software transactional memory is a promising new approach to programming shared-memory parallel programs. The functional programming language Haskell elegantly implements this abstraction for concurrent communication.

Keywords: Ensemble quantum computing, parallel quantum computing, transactional memory

1 Introduction

Ensemble quantum computing [3] (EQC) is in general physically realized by some scheme using NMR. It essentially differs from traditional quantum computing only in that it uses many copies of a quantum system (e.g., a liquid solution - such as each molecule is potentially a single quantum computer) and the result of a measurement is the expectation value of the observable, rather than a random eigenvalue. Parallel quantum computing in a single ensemble quantum computer [9] (PQC from here) explores the ensemble to gain additional speedup. Besides quantum parallelism, intrinsic from the use of superposed quantum states, a kind of classical single-instruction-multiple-data parallelism is achieved

[1] Email: juvizzotto@gmail.com
[2] Email: dubois@atlas.ucpel.tche.br

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

by making quantum systems (the molecules) working in parallel, as in classical computation. In the PQC the whole state is prepared in such a way a subset of qubits is in a mixed state representing the communicating quantum computers while the other qubits in pure state are the proper argument registers of each quantum computer. The authors have shown that the PQC enables additional speedup to important quantum algorithms like Grover and Shor. Specially, unsorted database search can be speedup greatly.

We have noted that this particular way of structuring the state of the PQC fits with what is well know as multithreading programming. Basically, in a multithreading environment, a process has many execution threads, each of which run independently and which may share a common memory area. What is interesting here is that computationally, the PQC quantum state, living in a Hilbert space, is interpreted as *global*. Hence, all quantum computer of the PQC share a commom global memory area.

That is not new the need of synchronization mechanisms for parallel programs and there are many alternative approaches in the literature, such as *locks* and *mutexes*. However has been claimed by several authors [11,8,7] that those programming styles are hard to use and may easily produce programs with errors.

Software transactional memory is a promising new approach to programming shared-memory parallel programs. The functional programming language Haskell elegantly implements this abstraction for concurrent communication.

This work is a stepping stone towards the development of a high level and elegant approach to structure and project parallel quantum algorithms.

2 Parallel Quantum Computing in a Single Ensemble Quantum Computer

The idea in the PQC [9] is to run many copies of quantum systems, which are in the ensemble, in parallel. The goal, as in classical computation, is to achieve additional speedup running tasks in parallel. By running several identical quantum computers in parallel, unsorted database search, for instance, can be speeded up greatly [4].

Consider an EQC quantum computer model with $N_1 = 2^{n_1}$ molecules, such that each molecule can be operated and measured. The PQC computer works in a state called argument register which is divided into two parts: one part with n_1 qubits called n_1-register and another part with n_2 qubits called n_2-register, and $n = n_1 + n_2$. Before a computation, the argument register is in a mixed state with N_1 constituent. Each constituent is characterized by the state of the n_1-register. The n_2-register in a given constituent is in a superposed state of its $N_2 = 2^{n_2}$ basis states. The density operator of the ensemble is

$$\rho = \frac{1}{N_1} \sum_{j_1=0}^{N_1-1} [\sum_{j_2=0}^{N_2-1} c_{j_1,j_2} |j_1, j_2\rangle][\sum_{j_2=0}^{N_2-1} c_{j_1,j_2} \langle j_1, j_2|]$$

In the EQC, there are N_1 constituents and N_1 molecules. Each molecule is in a different state $\sum_{j_2=0}^{N_2-1} c_{j_1,j_2} |j_1, j_2\rangle$, which is a superposition of N_2 number of computational basis states.

A unitary transformation on the computation state described above can be denoted by:

$$\rho \rightarrow \rho_c = U_c \rho U_c^{-1} = \frac{1}{N_1} \sum_{j_1=0}^{N_1-1} [\sum_{j_2=0}^{N_2-1} c_{j_1,j_2} U_c |j_1, j_2\rangle][\sum_{j_2=0}^{N_2-1} c_{j_1,j_2} \langle j_1, j_2| U_c^\dagger]$$

This quantum computation is defined as the parallel quantum computing. In fact it is N_1 quantum computers working in parallel. The computation U_c can be the same for all molecules, but the databases, numbers represented by different molecules, can be different.

Measurements are treated as average expectation values in the PQC and will be discussed in the further version of this work.

3 Software Transaction Memory in Haskell

In [5], STM Haskell, a new concurrency model for Haskell based on *software transactional memory* is proposed. In this model, programmers define *atomic blocks* that are executed atomically with respect to every other atomic block. STM Haskell provides the $atomically$ primitive to define atomic blocks:

$$atomically :: STM\ a \rightarrow IO\ a$$

The $atomically$ primitive takes a *memory transaction* ($STM\ a$) as an argument and executes it atomically. A *memory transaction* is committed only if no other transaction has modified the memory its execution depends on. If there was concurrent access to shared variables the transaction is restarted. An execution of $atomically$ block must guarantee [10]:

- **Atomicity**: The effects of an $atomically$ block are visible all at once to other threads
- **Isolation**: The execution of an $atomically$ block can not be affected by the execution of other threads. An $atomically$ block executes as if it had its own copy of the state of the program

Inside of a memory transaction a program can read and write into *transactional variables*. A variable of type $TVar\ a$ is a transactional variable that can hold a value of type a. STM Haskell provides the following primitives for reading and writing on transactional variables:

$$readTVar :: TVar\ a \rightarrow STM\ a$$
$$writeTVar :: TVar\ a \rightarrow a \rightarrow STM\ ()$$

The $readTVar$ primitive takes a $TVar$ as an argument and returns an STM action that, when executed, returns the current value of the $TVar$. The $writeTVar$ primitive is used to write a new value into a $TVar$. STM actions can be composed together using the same **do** notation used to compose IO actions in Haskell:

$$addTVar :: TVar\ Int \rightarrow Int \rightarrow STM\ ()$$
$$addTVar\ tvar\ i = \mathbf{do}\ \{v \leftarrow readTvar\ tvar$$
$$; writeTVar\ tvar\ (v + i)\}$$

The $addTVar$ function can be used to read and then write a new value into a $TVar$. These two actions can be executed atomically by using the $atomically$ primitive:

$$incTVar :: TVar\ Int \rightarrow IO\ ()$$
$$incTVar\ tvar = atomically\ (addTVar\ tvar\ 1)$$

Inside a memory transaction, only pure functions and STM actions can be executed. As a transaction may be aborted and re-run, the type system guarantees that no other irrevocable side-effects like IO actions can be performed inside an atomic block.

STM Haskell also provides a $retry :: STM\ ()$ primitive that is used to abort a transaction so that it can be restarted from the beginning:

$withdraw :: TVar\ Float \to Float \to STM\ ()$
$withdraw\ tvar\ v = \mathbf{do}\ \{ r \leftarrow readTVar\ tvar$
$\quad ; \mathbf{if}\ (r < v)\ \mathbf{then}\ retry$
$\quad\quad \mathbf{else}\ writeTVar\ tvar\ (r - v)\}$

Transactions can also be composed as *alternatives* using the $orElse$ function. The transaction t_1 `` `orElse` `` t_2 will first attempt to execute t_1, if it retries then transaction t_2 will be executed. If t_2 also retries then the entire call retries.

4 Quantum Arrows

In an early work [12] we have shown that the superoperators formalism used to express general quantum operations is an instance of a generalization of monads called *arrows* [6].

In this section, we briefly review state vectors represented as monads and the density matrix approach. Then we discuss how superoperators can be well fit in the concept of arrows. The presentation is in the context of the functional programming language Haskell.

4.1 Vectors as Monads

Given a set a representing observable (classical) values, i.e. a *basis* set, a pure quantum state is a vector $a \to \mathbb{C}$ which associates each basis element with a complex probability amplitude. In Haskell, a finite set a can be represented as an instance of the class $Basis$, shown below, in which the constructor $basis :: [a]$ explicitly lists the basis elements. The basis elements must be distinguishable from each other, which explains the constraint $Eq\ a$ on the type of elements:

$\mathbf{class}\ Eq\ a \Rightarrow Basis\ a\ \mathbf{where}\ basis :: [a]$
$\mathbf{type}\ K = Complex\ Double$
$\mathbf{type}\ Vec\ a = a \to K$

The type K (notation from the base field) is the type of probability amplitudes.

The monadic functions for vectors are defined as:

$return :: Basis\ a \Rightarrow a \to Vec\ a$
$return\ a\ b = \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0$
$(\gg\!=) :: (Basis\ a, Basis\ b) \Rightarrow Vec\ a \to (a \to Vec\ b) \to Vec\ b$
$va \gg\!= f = \lambda b \to sum\ [(va\ a) * (f\ a\ b) \mid a \leftarrow basis]$

$return$ just lifts values to vectors, and $bind$, given a *unitary operator* (i.e., *unitary operator*) represented as a function $a \to Vec\ b$, and given a $Vec\ a$, returns a $Vec\ b$ (that is, it specifies how a $Vec\ a$ can be turned in a $Vec\ b$). Actually, as explained in [12], because of the $Basis$ constraint over the sets which we can build vectors, we use a slight more general concept of monads called *kleisli structure* [1] or *indexed monads*.

4.2 Superoperators as Arrows

Intuitively, density matrices can be understood as a statistical perspective of the state vector. In the density matrix formalism, a quantum state that used to be modelled by a vector v is now modelled by its outer product.

$\mathbf{type}\ Dens\ a = Vec\ (a, a)$
$pureD :: Basis\ a \Rightarrow Vec\ a \to Dens\ a$

$$pureD\ v = lin2vec\ (v) \rangle\!*\!\langle (v)$$
$$lin2vec :: (a \to Vec\ b) \to Vec\ (a, b)$$
$$lin2vec = uncurry$$

The function $pureD$ embeds a state vector in its density matrix representation. For convenience, we uncurry the arguments to the density matrix so that it looks more like a "matrix."

Operations mapping density matrices to density matrices are called *superoperators*:

$$\mathbf{type}\ Super\ a\ b = (a, a) \to Dens\ b$$

The application function $\gg\!=$ above defines how the superoperator is going to act over the matrix.

The concept of arrows [6] extends the core lambda calculus with one type and three constants satisfying nine laws. The type is $A \to B$ denoting a computation that accepts a value of type A and returns a value of type B, possibly performing some side effects. The three constants are: arr, which promotes a function to a pure arrow with no side effects; $\ggg$, which composes two arrows; and $first$, which extends an arrow to act on the first component of a pair leaving the second component unchanged.

Just as the probability effect associated with vectors is not strictly a monad because of the $Basis$ constraint, the type $Super$ is not strictly an arrow as the following types include the additional constraint requiring the elements to be comparable. We have defined the concept of *indexed arrows* in [12], which allows the constraint. Bellow we show the instantiation of type $Super$ as an arrow.

$$arr :: (Basis\ b, Basis\ c) \Rightarrow (b \to c) \to Super\ b\ c$$
$$arr\ f = fun2lin\ (\lambda(b_1, b_2) \to (f\ b_1, f\ b_2))$$
$$\ggg ::(Basis\ b, Basis\ c, Basis\ d) \Rightarrow Super\ b\ c \to Super\ c\ d \to Super\ b\ d$$
$$f \ggg g\ b = (f\ b \gg\!= g)$$
$$first :: (Basis\ b, Basis\ c, Basis\ d) \Rightarrow Super\ b\ c \to Super\ (b, d)\ (c, d)$$
$$first\ f\ ((b_1, d_1), (b_2, d_2)) = permute\ ((f\ (b_1, b_2)) \langle * \rangle (return\ (d_1, d_2)))$$
$$\quad \mathbf{where}\ permute\ v\ ((b_1, b_2), (d_1, d_2)) = v\ ((b_1, d_1), (b_2, d_2))$$

The function *arr* constructs a superoperator from a pure function by applying the function to both the vector and its dual. The composition of arrows just composes two superoperators using the *bind* from Section 4.1. The function *first* applies the superoperator f to the first component (and its dual) and leaves the second component unchanged. The definition calculates each part separately and then permutes the results to match the required type.

4.3 A Better Notation for Arrows

Following the Haskell's monadic **do**-notation, Paterson (2001) presented an extension to Haskell with an improved syntax for writing computations using arrows. We concentrate only on the explanation of new forms which we use in our examples. Here is a simple example to illustrate the notation:

$$e_1 :: Super\ (Bool, a)\ (Bool, a)$$
$$e_1 = proc\ (a, b) \to \mathbf{do}$$
$$\qquad r \leftarrow lin2super\ hadamard \prec a$$
$$\qquad returnA \prec (r, b)$$

The **do**-notation simply sequences the actions in its body. The function $returnA$ is the equivalent for arrows of the monadic function $return$. The two additional keywords are:

- the *arrow abstraction* **proc** which constructs an arrow instead of a regular function.
- the *arrow application* $\prec$ which feeds the value of an expression into an arrow.

Paterson (2001) shows that the above notation is general enough to express arrow computations and implemented a Haskell's module which translates the new syntax to regular Haskell. In the case of `e1` above, the translation to Haskell produces the following code:

$$e_2 :: Super\ (Bool, a)\ (Bool, a)$$
$$e_2 = first\ (lin2super\ hadamard)$$

Hence, using the arrows approach in Haskell one can manipulate the quantum state in a high level way. For instance, having defined the right operations, the teleportation algorithm [2] can be programmed as the following:

$$teleport :: Super\ (Bool, Bool, Bool)\ Bool$$
$$\begin{aligned} teleport = proc\ (eprL, eprR, q) \rightarrow \mathbf{do} \\ (m_1, m_2) \leftarrow alice \prec (eprL, q) \\ q' \leftarrow bob \prec (eprR, m_1, m_2) \\ returnA -<\ q' \end{aligned}$$

The code would be a superoperator which acts over a three qubit density matrix (of type $Dens\ (Bool, Bool, Bool)$) and returns a one qubit matrix (of type $Dens\ Bool$). Using superoperators as arrows, the quantum state can be easily manipulated as above.

5 Modelling Parallel Quantum Computing using STM

We propose to use density matrices inside $TVars$:

$$\mathbf{type}\ QSt\ a = TVar\ (Dens\ a)$$

In such a way we can define a global quantum state which can be accessed an manipulated by all parallel/distributed processes. This seems to be exactly what we need to code the multithreading PQC presented in Section 2. The use of superoperators can greatly help the processes to access only small parts of the state.

The goal of using STM is to synchronize the access of critical parts of the state when doing critical operations like measurements.

A very simple example of an operation on the $QSt\ a$ is the identity operation codded below:

$$qid :: (Basis\ a) \Rightarrow QSt\ a \rightarrow Super\ a\ a \rightarrow STM\ ()$$
$$\begin{aligned} qid\ qtvar\ s = \mathbf{do}\ \{ & d \leftarrow readTVar\ qtvar \\ & ; writeTVar\ qtvar\ (d \ggg= s)\} \end{aligned}$$

Teleportation is a typical distributed quantum algorithm. The idea of teleportation is to disintegrate an object in one place making a perfect replica of it somewhere else. Indeed quantum teleportation [2] enables the transmission, *using a classical communication channel*, of an unknown quantum state via a previously shared epr pair.

Using arrows and the notation introduced by Patterson, we have expressed quantum teleportation in [12].

We break the algorithm in two individual procedures, $alice$ and bob. Besides the use of the arrows notation to express the action of superoperators on specific qubits, we incor-

porate the measurement in Alice's procedure, and trace out the irrelevant qubits from the answer returned by Bob.

$$
\begin{aligned}
&alice :: Super\ (Bool, Bool)\ (Bool, Bool)\\
&alice = proc\ (eprL, q) \rightarrow \mathbf{do}\\
&\quad (q_1, e_1) \leftarrow (lin2super\ (controlled\ qnot)) \prec (q, eprL)\\
&\quad q_2 \leftarrow (lin2super\ hadamard) \prec q_1\\
&\quad ((q_3, e_2), (m_1, m_2)) \leftarrow meas \prec (q_2, e_1)\\
&\quad (m1', m2') \leftarrow trL\ ((q_3, e_2), (m_1, m_2))\\
&\quad returnA \prec (m1', m2')\\
&bob :: Super\ (Bool, Bool, Bool)\ Bool\\
&bob = proc\ (eprR, m_1, m_2) \rightarrow \mathbf{do}\\
&\quad (m2', e_1) \leftarrow (lin2super\ (controlled\ qnot)) \prec (m_2, eprR)\\
&\quad (m1', e_2) \leftarrow (lin2super\ (controlled\ z)) \prec (m_1, e_1)\\
&\quad q' \leftarrow trL \prec ((m1', m2'), e_2)\\
&\quad returnA \prec q'
\end{aligned}
$$

Having defined Alice and Bob procedures we can now codify the teleportantion procedure using the QSt. The idea is to arrange the state inside the QSt as proposed in Section 2. In this way we will have a quantum state shared by Alice and Bob. The first qubit inside the QSt is the identifier, saying if the state is from Alice or Bob.

6 Conclusion

We have proposed a model for parallel quantum computing in a single ensemble quantum computer using Haskell's software transaction memory. We hope this approach will give us a simple and high level way to write and develop parallel quantum algorithms.

Acknowledgements

We would like to thank our colleagues Amr Sabry and Antônio Carlos da Rocha Costa for interesting discussions and feedback on our work.

References

[1] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, 1999.

[2] C. H. Bennett, G. Brassard, C. Crepeau, R. Jozsa, A. Peres, and W. Wootters. Teleporting an unknown quantum state via dual classical and EPR channels. *Phys Rev Lett*, pages 1895–1899, 1993.

[3] David G. Cory, Amr F. Fahmy, and Timothy F. Havel. Ensemble quantum computing by nmr spectroscopy. *Natl. Acad. Sci. USA*, 94:1634–1639, 1997.

[4] Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79(2):325–328, Jul 1997.

[5] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP'05*. ACM Press, 2005.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[7] Simon P. Jones. Beautiful concurrency, 2007.

[8] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[9] Gui Lu Long and L. Xiao. Parallel quantum computing in a single ensemble quantum computer. *Phys. Rev. A*, 69(5):55–92, 2004.

[10] Simon Peyton Jones. *Beautiful Concurrency*. O'Reilly, 2007.

[11] Ravi Rajwar and James Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, 2003.

[12] Juliana K. Vizzotto, Thorsten Altenkirch, and Amr Sabry. Structuring quantum effects: Superoperators as arrows. *Mathematical Structures in Computer Science, special issue on Quantum Programming Languages*, 16:453–468, 2006.