Shared and distributed memory implementations for parallel simulations of a quantum search algorithm to solve the 3-SAT problem

Sergio Nesmachnow, Marcos Barreto, Gonzalo Abal
Facultad de Ingeniería, Universidad de la República, Uruguay
{sergion,mbarreto,abal}@fing.edu.uy

Abstract

This article presents two parallel implementations for simulations of a new quantum search algorithm to solve the 3-SAT problem, by using shared-memory and distributed-memory approaches. Very large computing time are needed to perform the simulations of quantum computing algorithms to solve realistic problems, thus high performance computing approaches are the only viable option to compute results in reasonable times. The experimental analysis demonstrates the advantage of using both shared-memory and distributed-memory parallel computing techniques in order to take advantage of available computing resources in a distributed computing environment to perform higly complex quantum simulations. Speedup values up to 5× for the shared-memory implementation and up to 6× for the distributed-memory implementation are reported.

1 Introduction

Quantum computing has emerged as a new computing model that can offer significantly improvement over classical computation. By combining physics, mathematics and computer science concepts, quantum computing speeds up classical computation for important problems such as search and factorization [18]. Several quantum algorithms have been proposed that, when executed on a quantum computer, could provide significantly computational improvements compared to their best-known analogue classical methods [2].

Nowadays, the research community is in the quest for a practical quantum information processing machine, but the existing prototypes are only useful to handle a few quantum units of information (qubits). Meanwhile a quantum machine is not available, quantum algorithms are tested by performing simulations on classical computers. These simulations demand exponential computing resources (memory to storage and CPU time to process), since a quantum state of n qubits is represented by a matrix of dimension 2^n.

Parallel and distributed computing techniques are viable strategies to reduce the execution time and memory needed to simulate quantum algorithms as if they were executed on a quantum computer. This techniques make quantum simulations tractable by sharing the large amounts of calculations among different processing units that cooperate to perform the simulation. In the past ten years, parallel computing techniques received a renewed interest due to the emergence of distributed computing platforms (cluster, grid, and cloud) and multicore processors [9, 14].

In this line of work, this article presents two parallel implementations for the simulation of a quantum search algorithm to solve the 3-SAT. The 3-SAT is a case of the Boolean satisfiability problem (with m clauses and 3 variables in each clause), a classical optimization problem with important applications in many areas. The 3-SAT is NP-hard [7] since no polynomial-time algorithm is known to solve, it and it is used to prove that many other problems are in the NP-complete class [10]. Stochastic algorithms have been successfully applied to solve hard-optimization problems as they allow to find solutions in reasonable times [24]. The proposed quantum algorithm combines one of the best known classical methods for the 3-SAT problem (the Schöning algorithm) [21], with a local search variant of Shenvi's quantum search algorithm [22]. This article extends our previous work on spatial quantum search algorithms [1] and its application to 3-SAT [4].

The main contribution of this article are: i) to introduce shared- and distributed-memory parallel implementations for the simulation of a quantum search algorithm to solve the 3-SAT problem, and ii) to report efficient simulation times for the parallel quantum algorithm, achieving speedup values up to 5× for the shared-memory implementation and up to 6× for the distributed-memory implementation.

The article is organized as follows. Section 2 briefly introduces basic concepts about parallel computing and performance metrics. A review of related works on simulation techniques for quantum computing algorithms in presented in Section 3. After that, Section 4 introduces the quantum search algorithm designed to solve the 3-SAT problem.

The proposed parallel implementations for the quantum simulations are described in Section 5. The experimental evaluation of the proposed parallel algorithms is reported in Section 6, with especial emphasis on studying the computational performance of both implementations. Finally, Section 7 summarizes the conclusions and discusses the main lines for future work.

2 Parallel Computing

This section introduces the main concepts about parallel computing techniques.

2.1 Introduction

Parallel computing techniques are commonly used to design algorithms that are able to execute faster than its sequential counterpart. Moreover, parallel computing allows researchers to tackle larger and more difficult problems by applying a rather simple approach: the original problem is divided in subproblems, which are simultaneously solved using several computing resources. This cooperative approach allows exploiting the current availability of computing resources in multicore and parallel/distributed computing environments [9].

Nevertheless, parallel computing also introduces some difficulties when developing a program and when executing or verifying its correctness. The main difficulties are related with the simultaneous execution of several processes in different computing resources and the communication and synchronization required to achieve the cooperation between different processes [9].

The strategies for designing parallel programs differ from the traditional ones used in sequential programming. Since the resolution of many subproblems have to be taken into account, applying wise strategies for splitting data and control between processes is highly desirable, since it can significantly increase the performance improvements provided by a parallel algorithm. Another aspect to consider are the communication paradigms between processes and the way to perform them (e.g. shared or distributed memory, see next subsection) [25].

2.2 Parallel programming

Two main paradigms exists for parallel programming: shared-memory and distributed-memory. The shared-memory paradigm is based on a number of processes (usually light processes or threads) executing on different cores of a single computer, while sharing a single memory space. The main advantages of this model are that it is easy to implement, and it has a relatively low communication cost via the shared memory resource.

However, the scalability of the shared-memory approach is limited by the number of cores integrated in a single multicore server (up to 64 cores nowadays) and it cannot take advantage of parallel environments composed by many computers. Several libraries are available to implement the thread creation, management, communication and synchronization, including the POSIX thread library for C, OpenMP [5], and others.

On the other hand, the distributed-memory paradigm relies on many processes executed on different computers. The communication and synchronization is performed by explicit message passing, since there is no other shared resource available than the network used to interconnect the computers. The main feature of this model is that it can take advantage of parallel environments with a large number of computer resources available (i.e. large clusters and grids of computers). Nevertheless, the communication between processes is more expensive than in the shared-memory approach, and carefully implementation is needed to achieve maximum efficiency. Several languages and libraries have been developed for designing and implementing parallel programs using the distributed-memory approach, including the Message Passing Interface (MPI) standard [12].

2.3 Performance Metrics

The most common metrics used by the research community to evaluate the performance of parallel algorithms are the *speedup* and the *efficiency*.

The speedup evaluates how much faster a parallel algorithm is than its corresponding sequential version. It is computed as the ratio of the execution times of the sequential algorithm (T_1) and the parallel version executed on m computing elements (T_m) (Equation 1). When applied to non-deterministic algorithms, the speedup should compare the *mean* values of the sequential and parallel execution times (Equation 2). The ideal case for a parallel algorithm is to achieve linear speedup ($S_m = m$), but the most common situation is to achieve sublinear speedup ($S_m < m$), mainly due to the times required to communicate and synchronize the parallel processes.

The efficiency is the normalized value of the speedup, regarding the number of computing elements used to execute a parallel algorithm (Equation 3). This metric allows the comparison of algorithms eventually executed in non-identical computing platforms. The linear speedup corresponds to $e_m = 1$, and in the most usual situations $e_m < 1$.

$$S_m = \frac{T_1}{T_m} \quad (1) \qquad S_m = \frac{E[T_1]}{E[T_m]} \quad (2) \qquad e_m = \frac{S_m}{m} \quad (3)$$

2.4 Models for communication

Three main communication models have been proposed for parallel algorithms:

- *master-slave*: a distinguished process (the master) controls a group of slave processes. The master process creates and manages the slaves, by sending information regarding the subproblem or function to perform. The slaves perform the computing and then they communicate back to the master process the results of their execution. The communication and control are centralized in the master process.
- *client-server:* two classes of processes are used; the clients, which request services from a process of another kind, and the servers, that handle requests from clients and provide the services. Each server is always waiting for requests, and typically many clients consume services from only one server process. This paradigm provides a model to communicate distributed applications that simultaneously behave as clients or servers for different services.
- *peer-to-peer:* each client process has the same capabilities to establish communication. Nowadays, this model is well-known by its use in P2P networks to share files. This model can be implemented by developing the logic of a both a client and a server process within the same peer process.

The parallel quantum algorithms proposed in this work follow the master-slave model for communication, using shared-memory and distributed-memory approaches, respectively.

3 Related work

In the last decade quantum mechanics have become an important matter of research, but the existing technology only allows to construct quantum computers of few qubits [11]. Meanwhile, the behavior of quantum computers has to be simulated on classical computers. The intrinsic parallelism of quantum computing allows performing simultaneous operations on a set with an exponential number of elements. Thus, parallel computing techniques seem an intuitive idea to simulate such parallelism. The main limitations when simulating quantum algorithms on classical computers are the memory and CPU time required, which grow exponentially with the number of qubits used in the quantum algorithm. Therefore, parallel computing techniques are used to distribute both the state and the computations among many processes, in order to reduce the overall simulation time.

Raedt et al. [20] presented a quantum computer simulator developed in Fortran 90 that can execute on parallel environments such as supercomputers and cluster of computers. The reported software balances memory consumption among computers on an distributed environment using the MPI library. Quantum registries are distributed in many computers to reduce memory consumption, but the communication between distributed process is increased and computational performance is reduced. Nevertheless, this is a useful approach to simulate quantum algorithms to solve real size problems. The authors proposed to use the quantum simulator as a new type of benchmark to assess the computational power of the new generation of high-end computer systems.

Obenland [19] proposed another quantum computer simulator. It simulates one of the most promising physical implementations: the Cirac and Zoller implementation [6]. Just like in the work by Raedt et al. [20], parallel computing techniques are used to distribute memory consumption on a cluster of computers. The simulator is implemented in C language and the MPI library is used to implement the communication between process. The reported results exhibit close to ideal speedup for as many as 256 processors.

In recent years, researchers have put attention to the highly parallel structure of GPUs (graphic processing units) to execute scientific applications. This is also the case of quantum computing. Gutierrez et al. [13] proposed a parallel quantum computer simulator that runs on GPU. The speedup obtained is up to 85 relative to the CPU implementation reported. Nevertheless, there is a maximum registry size allowed of 26 qubits, imposed by the available memory in the GPU device.

Apart from parallel simulations of quantum computers, there are quantum programming languages that can exploit the advantages provided by a parallel/distributed environment. For instance, LanQ [16] is an imperative programming language which supports interprocess communication to allow the parallel execution of quantum algorithms.

In this article, parallel simulations of a new quantum algorithm are presented, by extending our previous work on spatial local search quantum algorithm for the 3-SAT [4]. High performance parallel computing techniques applying both shared-memory and distributed-memory are used to speedup the execution time required to simulate the quantum algorithm on classical computer.

4 Quantum-search applied to the 3-SAT

This section introduces the 3-SAT problem and the quantum algorithm proposed to solve it.

4.1 The 3-SAT problem

The 3-SAT is a classical optimization problem in the field of computing theory, with important applications in many other research fields including electronic design and verification planning [15], scheduling [8], cryptography [23], and others.

Boolean satisfiability. In propositional logic, a literal is either a logical variable or its negation, and a Boolean expression is a conjunction of m clauses, each of whom is a disjunction of literals. The Boolean satisfiability (SAT) problem consists of determining, if it exists, a truth assignment for the variables that makes a given Boolean expression true. The k-SAT is the variant where clauses have k literals, and the 3-SAT problem is a special case for which $k = 3$. The k-SAT for $k > 3$ can always be mapped to an instance of a 3-SAT [10].

Mathematical formulation. The mathematical formulation of the 3-SAT problem is as follows:

- Let a set of n literals $X \equiv \{x_1, \ldots, x_n\}$, where $x_r = \{0, 1\}$.
- Let a Boolean expression $\Phi = \bigwedge_{i=1}^{i=m} C_i$, formed by m clauses $C = \{C_1, \ldots, C_m\}$, with $C_i = \bigvee_{j=1}^{3} l_{ij}$, where l_{ij} is either a literal x_r or its negation $\neg x_r$.
- The 3-SAT problem consists in determining the set of values for the literals $X \equiv \{x_1, \ldots, x_n\}$ that makes the Boolean expression Φ true.

When $k = 2$ the problem is in the complexity class P, as it can be solved in polynomial time [3]. On the other hand, the k-SAT problem is NP-Complete when $k \geq 3$ [7]; in fact, it was the first problem proved to be NP-Complete and many NP-complete problems have been proved so, by reducing them to an instance of 3-SAT. Thus, if a polynomial time algorithm to solve the k-SAT for $k \geq 3$ is known, then every NP–complete problem can be solved in polynomial time. However, no such efficient algorithm to solve the 3-SAT is known.

4.2 A quantum search algorithm for the 3-SAT problem

The proposed method combines the ideas in the algorithms by Schöning [21] and Shenvi [22], applying a quantum search using neighborhoods, a common procedure in metaheuristic optimization methods [24]. The proposed algorithm is simulated using parallel computing techniques to evaluate its correctness.

Two elements in the search space are in the same neighborhood if they are "close", considering the number of movements needed to get to one element from the other. In the 3-SAT problem, a neighborhood is a subspace of qubits: let be the element $x = [x_1, x_2, ..., x_n], x_i \in \{0, 1\}$ and $r \leq l \leq n$. Two strings t_1, t_2 belong to the same neighborhood if and only if $x_k^{t_1} = x_k^{t_2}\ \forall\ k \notin [r, l]$ (these are the *fixed* qubits that defines the neighborhood). For example, $|00011\rangle$, $|00010\rangle$ and $|00001\rangle$ belong to the neighborhood defined by the elements in the search space that have x_5, x_4 and x_3 fixed in $|0\rangle$. The neighborhood is $N = |000 * *\rangle$, a subspace of size 2 qubits ($*$ stands for a 'don't care' boolean value).

Algorithm 1 describes the proposed method, named Shenvi with Local Search in Neighborhood (SLSN). It randomly chooses a string x^* and searches for a solution using Shenvi's algorithm in a neighborhood of x^*. If a solution is not found in this subset of elements, the algorithm jumps randomly to another location in the search space and repeats the process.

Algorithm 1 Description of the SLSN algorithm

```
while not solution found do
    x* = generate random state
    define neighborhood N as the closest elements to x*
    if (N is satisfiable) then
        apply Shenvi(x*, N)
        measure quantum state
    end if
end while
```

5 Parallel implementations of the quantum simulations

Simulating quantum computing demands high computational resources, as the required memory and processing power grows exponentially with the number of qubits used in quantum algorithms. Parallel computing techniques are an useful approach to reduce the execution time of such simulations, by exploiting the simultaneous execution on many available computing resources.

This section presents two parallel implementations of the SLSN quantum algorithm using shared-memory and distributed-memory approaches.

5.1 Shared-memory parallel SLSN algorithm

This section presents the shared-memory parallel quantum simulation of the proposed search algorithm to solve the 3-SAT.

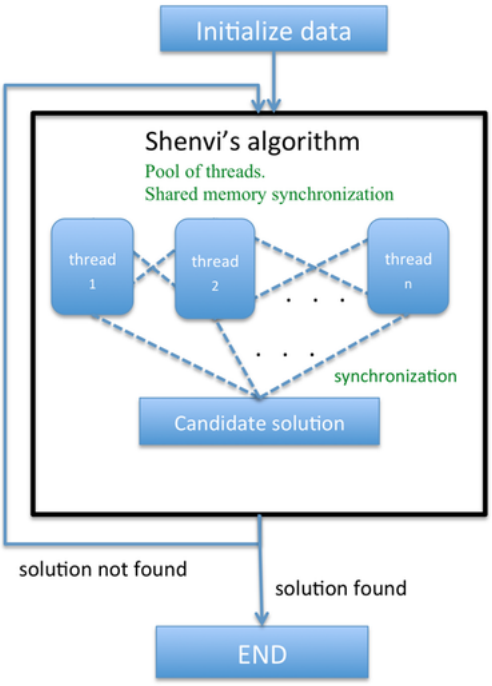

Figure 1. Diagram of the shared-memory parallel SLSN algorithm

All shared-memory versions of the proposed quantum search algorithms were developed using the OpenMP library [5]. The shared-memory approach was developed by splitting the workload of matrix operations in different cores. Figure 1 shows a diagram of the implementation, where shared-memory parallelization was used in the implementation of Shenvi algorithm using 8 processing cores. The parallel version of Shenvi is also used as a baseline for comparing the efficiency of the new SLSN algorithm.

5.2 Parallel Distributed SLSN

The second proposed implementation of SLSN is able to execute over a distributed computing resource (i.e. a cluster of computers), by combining both the shared-memory and the distributed-memory approaches.

Such combination of distributed- and shared-memory parallelism is not an easy-to-develop solution, since several issues had to be taken into account, including communication, synchronization, and load balancing between tasks. The distributed-shared-memory version of the proposed quantum search algorithm was developed using the MPI library [12].

The distributed simulation was developed using the master-slave model of communication between processes, and a specific load balancing method is applied to equally distribute the processing load among processes. Within each process, the implementation of the Shenvi algorithm applies a shared-memory parallelism strategy using the OpenMP library [5].

Initially, the master process randomly chooses a compatible neighborhood for each slave and another for itself. The slaves receive information to execute the SLSN algorithm; when a solution is found, the slave sends it to the master node. If the master node finds a solution or receives one, a message is sent to every slave to end the algorithm. Algorithm 2 describes the master process. Messages from slaves are listened in the *reader thread* (Algorithm 3), while the *execution thread* (Algorithm 4) sends necessary data to slaves and executes the shared-memory parallel SLSN algorithm itself (an *active* master-slave model is used, since the master also performs the same work than the slaves).

Algorithm 2 Master process of the parallel SLSN

```
read 3-SAT instance
create reader thread {it waits for messages of the slaves}
create execution thread {it sends data to the slaves and executes Shenvi}
while solution NOT found or received do
    wait
end while
{solution found or received}
for all s in slaves do
    send DIE signal
end for
```

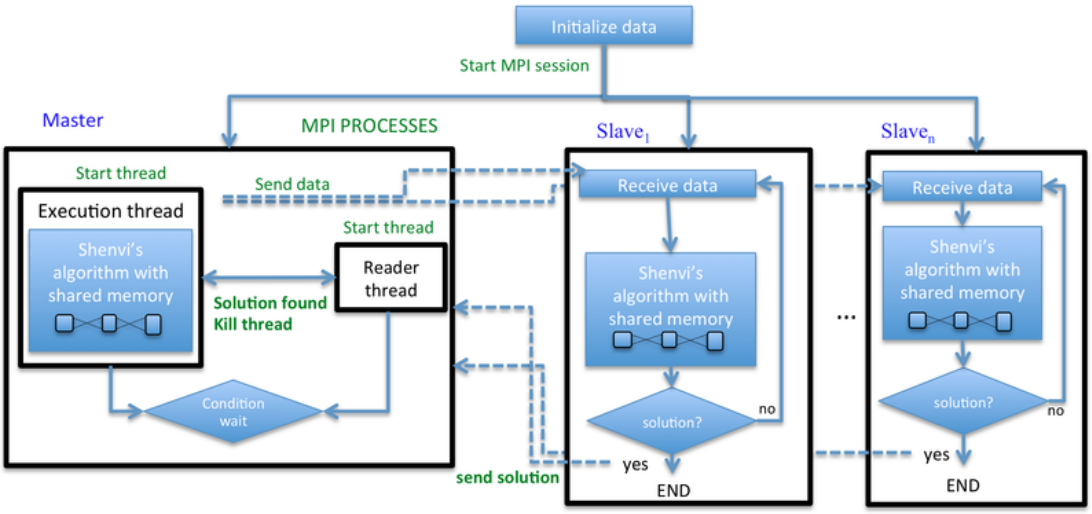

Figure 2. Diagram of the distributed-memory parallel SLSN algorithm

Algorithm 3 Reader thread (master process)

```
receive message from slave
if message received then
    kill execution thread
    save solution
    wake up master process
end if
```

Algorithm 4 Execution thread (master process)

```
repeat
    for all s in slaves do
        repeat
            x* = generate random state
            define neighborhood N as the closest elements
            to x*
            if N is satisfiable then
                send N,x* to s
            end if
        until N is satisfiable
    end for
    x* = generate random state
    define neighborhood N as the closest elements to x*
    res = Shenvi(x*, N)
    if res is solution then
        kill reader thread
        save solution and wake up master process
    end if
until solution found
```

The slave processes (Algorithm 5) perform the search until they receive a *DIE* message from the master, meaning that some slave process has found a solution to the problem.

Algorithm 5 Slave process

```
read 3-SAT instance
repeat
    x*, N = receive neighbourhood and state of master
    process
    res = apply Shenvi(x*, N)
    if res is solution then
        send res to master process
    end if
    {If a DIE message is not received, a solution is not
    found yet.}
until DIE message received
```

Figure 2 presents a diagram of the hybrid distributed-memory parallel implementation of the master process of the simulation of the quantum algorithm.

6 Experimental analysis

This section presents the problem instances and the execution results of the parallel implementations of the proposed algorithm. Also, it briefly presents the development and execution platform used to develop the simulations.

6.1 Development and execution platform

The parallel SLSN simulations were developed in C/C++ programming language. The shared-memory parallel simulation was developed using the OpenMP library version 4.3 and the distributed-memory simulation was developed using MPICH library version 1.2.7.

The experimental evañuation of the proposed parallel quantum search implementations was performed on a cluster of Dell Power Edge servers, with Quad-core Xeon E5430 processors at 2.66GHz and 8 GB of RAM, from the Cluster FING high performance computing infrastructure (cluster website: `http://www.fing.edu.uy/cluster`).

6.2 3-SAT instances

A benchmark set of 3-SAT instances with up to 23 variables and 83 clauses were used in the experimental analysis.

The G3 algorithm [17] was used to generate 3-SAT instances with few solutions. The theoretical relationship $m^* = 4.26n + 6.24$ defines a phase transition in which the 3-SAT problem goes from being a simple problem to an extremely hard-to-solve problem [8]. This relation was considered in order to generate hard-to-solve 3-SAT instances with few solutions.

6.3 Shared-memory simulations

The efficiency analysis results for the shared-memory simulations are presented in Table 1. The table reports the average execution times and speedup values obtained in the 10 executions of the Shenvi simulation performed for each problem instance. The results in Table 1 indicate that acceptable sub-linear speedup values are obtained when using 8 computing elements.

n	m	Shenvi execution time (in s.)		speedup
		sequential	parallel	
15	52	78	23	3.39
16	55	316	76	4.15
17	60	1258	309	4.07
18	64	2989	818	3.65
19	67	10904 (3 hours)	2705 (45 minutes)	4.03
20	70	57308 (16 hours)	11428 (3 hours)	5.01
21	73	84730 (23.5 hours)	21239 (5.9 hours)	3.99

Table 1. Comparison of the sequential and parallel simulations for Shenvi's algorithm.

The experimental results of the shared-memory parallel SLSN evaluation are shown in Table 2. The table reports the problem dimensions and the average values for the execution time, for the number of steps until a solution was found, and for the measures required. All average were computed over 10 independent executions performed for each problem instance.

Figure 3 compares the execution times (in logarithmic scale) of the parallel SLSN simulations using the best and the worst neighborhood size, and the Shenvi simulation. The best and worst neighborhood sizes are defined as the neighborhood sizes that provided the lowest and highest execution times, respectively.

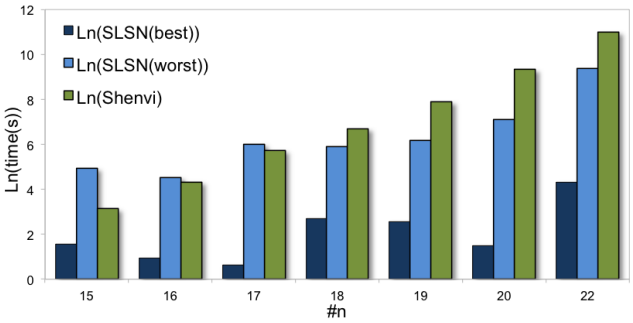

Figure 3. Execution time comparison: Shenvi, SLSN (best) and SLSN (worst) (logarithmic scale)

The results in Table 2 indicate that the configuration that computed the best results was always the one with the lower number of qubits. The default value of qubits in the neighborhood was $2\log_2(3n)$—in order to perform $3n$ steps, as proposed in Schöning's algorithm—, but the best efficiency values were obtained when using fewer qubits, since the neighborhood size notably impacts in the performance of simulation of the SLSN algorithm. SLSN successfully solved all the problem instances, and the number of steps significantly reduced as the number of variables grows. The results demonstrate that use of shared-memory parallel computing techniques significantly reduced the execution time of the quantum simulations.

6.4 Parallel distributed SLSN algorithm

The experimental results of the distributed-memory parallel SLSN algorithm are presented on Table 3. The table reports the problem dimensions and average values for execution time in 10 independent executions performed for each problem instance for 1, 2, and 3 computer nodes. The number of steps until a solution was found, measures required, and speedup are reported for 2 and 3 nodes.

Figure 4 graphically compares the execution time (in logarithmic scale) of the distributed-memory parallel SLSN implementation with the shared-memory parallel SLSN implementation.

n	m	# sol.	s	time(s)	steps	measures	n	m	# sol.	s	time(s)	steps	measures
15	52	14	2	0.2	3822	3.5	19	67	53	5	0.9	5970	6.0
			5	0.1	1041	9.3				7	2.2	2324	7.5
			7	0.5	1142	8.7				9	13.1	1469	4.5
			9	4.7	1155	5.0				12	91.8	316	5.2
			11	18.5	369	4.6				14	479.9	273	6.0
			13	140.8	226	10.1							
16	55	27	2	0.1	2646	5.3	20	70	233	5	0.2	510	3.6
			5	0.2	1286	4.2				7	1.2	506	3.2
			7	0.1	250	3.3				10	4.4	73	8.6
			9	2.6	479	2.9				11	10.4	97	2.4
			11	9.5	181	6.4				12	16.2	36	3.0
			13	93.6	237	4.4				14	131.4	32	6.4
										16	1225.2	40	2.0
17	60	55	2	0.2	3966	3.2	22	78	54	5	4.2	13688	4.2
			5	0.1	1370	6.1				7	16.2	11498	5.4
			7	0.2	639	4.7				10	75.6	1529	6.6
			9	1.9	370	7.7				11	319.0	2741	8.6
			11	15.1	290	4.5				12	894.0	2375	8.6
			13	43.7	111	4.1				14	1982.6	454	4.8
			15	409.5	86	4.1				16	11676.8	353	9.8
18	64	13	9	15.0	530	2.6	23	83	50	10	137.4	2653	2.0
			12	353.8	791	4.2				12	353.8	791	4.2
			11	26.8	160	4.0							
			13	108.0	107	2.4							
			15	370.6	28	8.6							

Table 2. SLSN results for the 3-SAT.

n	m	s	1 node	2 nodes				3 nodes			
			time(s)	time	steps	measures	speedup	time	steps	measures	speedup
15	52	11	18.5	12.2	50	2.9	1.5	8.5	24	4.7	2.2
		13	140.8	560.0	682	66.0	0.3	607.5	404	6.7	0.2
16	55	11	9.5	9.1	35	3.7	1.0	12.0	48	2.7	0.8
		13	93.6	128.3	27	3.1	0.7	104.0	52.3	6.5	0.9
17	60	11	15.1	9.9	29	4.5	1.5	7.6	109.6	27	2.0
		13	43.7	57.8	21	3.4	0.8	37.4	7	7.3	1.2
		15	409.5	310.9	10	7.1	1.3	273.9	13	5.8	1.5
18	64	11	26.8	21.8	69	4.0	1.2	18.4	58	5.2	1.5
		13	108	88.0	23.0	6.4	1.2	61.3	39	1.5	1.87
		15	370.0	395.1	27	3.0	0.9	278.0	9	7.2	1.3
19	67	12	91.8	72.0	70	5.7	1.3	36.1	36	8.4	2.5
		14	479.9	322.0	33	5.6	1.5	191.6	19	5.2	2.5
20	70	12	16.2	18.8	26.0	5.2	0.9	11.3	33.2	14.8	1.4
		14	131.4	143.8	17	3.2	0.9	82.3	4	12.7	1.6
		16	1225.2	787.4	13	8.8	1.6	184.3	11	8.3	6.6
22	78	12	894.0	724.2	36	4.4	1.2	598.2	97	3.4	1.5
		14	1982.6	1010.4	102	9.8	2.0	283.3	59	1.4	7.0
		16	11676.8	9516.0	222	5.0	1.2	2065.0	47	5.8	5.6

Table 3. Parallel distributed SLSN results for the 3-SAT problem.

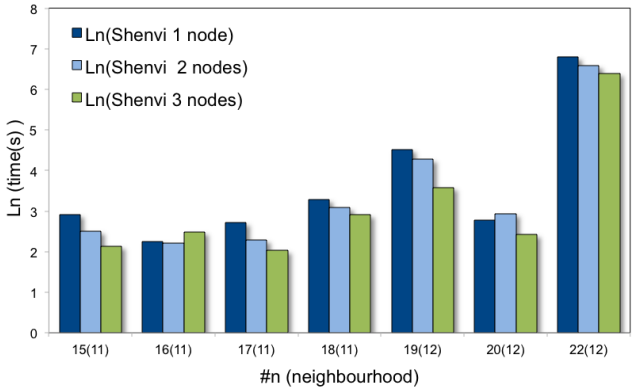

Figure 4. Execution time comparison: parallel distributed SLSN (best) using 1, 2, and 3 computing nodes (logarithmic scale)

The improvement in computational efficiency compared to the sequential simulation of SLSN algorithm is graphically analyzed in Figure 5.

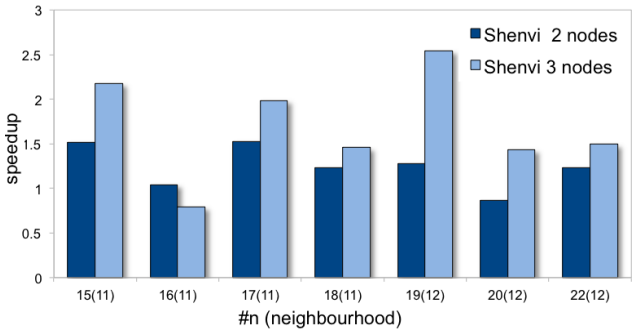

Figure 5. Speedup comparison: parallel distributed SLSN (best), executed with 2 and 3 computing nodes

The results in Table 3, and Figures 4 and 5 demonstrate that the distributed-memory parallel implementation of the SLSN algorithm provides improvements in the execution time when the work performed by each slave is relatively high. As the neighborhoods grow, the workload performed by each slave grows as well, and a better load distribution among processes is achieved, thus the speedup increases. For neighborhoods with more than 10 qubits (2^{10} elements in the neighborhood's search space) the parallel distributed SLSN algorithm provides improvements in the execution time compared to the parallel shared-memory SLSN.

7 Conclusions and future work

This article has presented two parallel implementations for the simulation of SLSN, a new quantum computing algorithm to solve the 3-SAT problem. Quantum methods to solve the 3-SAT problem have been previously proposed, but few simulated on classic computers. Thus, the main contributions of the research reported in this article are the SLSN quantum algorithm itself, and the parallel simulations over classical computers, especially the new hybrid parallel implementation for quantum simulation using both shared and distributed-memory approaches.

The experimental analysis performed over a set of hard-to-solve 3-SAT instances generated using a well-known methodology demonstrated that the proposed method allows computing accurate results. In addition, the computational efficiency analysis proved that the parallel implementations significantly speed up the execution time required to perform the quantum simulations.

The shared-memory simulation of the proposed algorithm was an easy to develop solution that significantly reduced the execution time of the simulation, reaching an speedup up to $5\times$. On the other hand, the hybrid implementation, combining distributed and shared-memory, applies a more sophisticated simulation approach, taking into account several important features such as problem decomposition, load balancing, and communication between processes. Nevertheless, the hybrid distributed-memory parallel implementation allowed to significantly reduce the execution time of the quantum simulation, and a speedup of $6\times$ was obtained.

Mainly due to the computational complexity of the quantum simulation, the best speedup and efficiency results were obtained when using small neighborhoods in the hybrid distributed-memory parallel implementation of the SLSN algorithm.

The main lines for future work are related with further studying the behavior of the SLSN algorithm and the parallel implementations of the quantum simulations. Regarding the first line, the impact of the neighborhood size on both the quality of results and the computational efficiency shall be better analyzed; and it would be interesting to investigate the distortion effect of noise in the quantum algorithm. Regarding the parallel implementations, the design of distributed algorithms able to execute in large computing infrastructures (i.e. grid and volunteer-computing platforms), is also a promising line of work. Finally, this line of research could be also extended to tackle other known NP-hard problems, taking advantage of the intrinsic parallelism of quantum computing to build better alternatives to the best known classic algorithms.

References

[1] G. Abal, R. Donagelo, F. Marquezino, and R. Portugal. Spatial search on a honeycomb network. *Mathematical Structures in Computer Science*, 21:1–11, 2010.

[2] A. Ambainis. Quantum search algorithms. *SIGACT News*, 35:22–35, 2004.

[3] B. Aspvall, M. Plass, and R. Tarja. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[4] M. Barreto, G. Abal, and S. Nesmachnow. A parallel spatial quantum search algorithm applied to the 3-SAT problem. In *Proceedings of XII Argentine Symposium on Artificial Intelligence*, pages 84–95, Córdoba, Argentina, 2011.

[5] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 1999.

[6] J. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74(20):4091–4094, 1995.

[7] S. Cook. The complexity of theorem proving procedures. In *Proceedings 3^{rd} Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, New York, 1971.

[8] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11^{th} National Conference of the American Association for Artificial Intelligence*, pages 21–27, Washington, D.C., 1993. AAAI.

[9] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[10] M. Garey and D. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[11] I. Glendinning and B. Omer. Parallelization of the qc-lib quantum computer simulator library. In R. Wyrzykowski et al., editor, *Proceedings of the 5^{th} International Conference on Parallel Processing and Applied Mathematics*, pages 461–468, Czestochowa, Poland, 2004. Springer.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with Message-Passing Interface*. MIT Press, 1999.

[13] E. Gutierrez, S. Romero, M. Trenas, and E. Zapata. Parallel quantum computer simulation on the CUDA architecture. In M. Bubak et al., editor, *Computational Science*, volume 5101 of *Lecture Notes in Computer Science*, pages 700–709. Springer Berlin/Heidelberg, 2008.

[14] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2009.

[15] J. P. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37^{th} Annual Design Automation Conference*, pages 675–680. ACM New York, NY, USA, 2000.

[16] H. Mlnarík. Operational semantics of quantum programming language lanq. Technical Report FIMU-RS-2006-10, FI MU, Brno, 2006.

[17] M. Motoki and O. Watanabe. Random generation of unique–solution 3SAT instances. Technical report, Tokyo Institute of Technology, 2011. Available at `http://www.is.titech.ac.jp/~watanabe/gensat/a1`. Retrieved April 2013.

[18] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Series on Information and the Natural Sciences. Cambridge University Press, 2000.

[19] K. Obenland and A. Despain. A parallel quantum computer simulator. *quant-ph/9804039*, 1998.

[20] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, 2007.

[21] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40^{th} Annual Symposium on Foundations of Computer Science*, page 410. IEEE, 1999.

[22] N. Shenvi, J. Kempe, and B. Whaley. Quantum random walk search algorithm. *Physical Review A*, 67:(052307), 2003.

[23] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12^{th} International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer-Verlag Berlin, Heidelberg, 2009.

[24] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley, 2009.

[25] L. Yang and M. Guo. *High-Performance Computing: Paradigm and Infrastructure*. Wiley-Interscience, 2005.