Rev. Roum. Sci. Techn.– Électrotechn. et Énerg.
Vol. **61**, *1*, pp. 47–52, Bucarest, 2021

Automatique et ordinateurs

AN EVOLUTIONARY COMPUTATIONAL SYSTEM ARCHITECTURE BASED ON A SOFTWARE TRANSACTIONAL MEMORY

BRANISLAV KORDIC[1], MARKO POPOVIC[1], MIROSLAV POPOVIC[1], MOSHE GOLDSTEIN[2], MOSHE AMITAY[2], DAVID DAYAN[2], ERICK FREDJ[2]

Key words: Software transactional memory, Python, Diffusion equation evolutionary programming simulated annealing method (DEEPSAM), Python software transactional memory (PSTM), Evolutionary programming (EP).

In the last decade software transactional memory has become a prominent programming paradigm, which aims to improve the execution performance of concurrent programs. However, most research in the field is done in programming languages such as C, C++ and JAVA. In this paper, we present a PSTM-based architecture of DEEPSAM, a computational chemistry program written in Python language. The PSTM-based architecture aims to improve execution performance of the original computational chemistry system architecture based on evolutionary programming and to provide transactional-memory-based means for its future optimizations. The metric analysis includes system execution time and problem size scalability. The experimental results show that the new PSTM-based version gained better execution time results relative to the original version. Likewise, the results did not reveal any architectural bottleneck of the new architecture.

1. INTRODUCTION

Engineering of computer based systems (ECBS) is an important field of electrical engineering, which uses computers as system control units. Transactional memory (TM) [1, 2] is a specific computer system component, which may be implemented as hardware, software, or hybrid (hardware and software). Computational chemistry system architecture, based on a particular TM, is presented in this paper. A similar approach may be used in ECBS of components for smart homes, smart grid, *etc*.

TM provides the alternative to traditional lock-based mechanisms by replacing them with a lock-free mechanism. In general, TM aims to provide better performance for parallel computation from multicore architectures and to ease the writing and maintenance of parallel programs as well. Since transactional execution hardware support is still missing in modern processors, Software Transactional Memory (STM) [3] continues to be the major tool for researchers.

In this paper, we present a case study of a new computational chemistry system architecture based on a STM. Many computational tasks in chemistry and biology, such as small molecule screening for drug discovery, next-generation sequencing (NGS) short reads alignment, protein structure prediction, and many more, are complex CPU-intensive real-world applications whose long execution times limit their practicality. The case study presented here analyses the impact of STM on the execution performance of such a real-world application: Diffusion Equation Evolutionary Programming Simulated Annealing Method (DEEPSAM) [4, 5].

DEEPSAM is a hybrid evolutionary programming (EP) [6, 7, 8] computational chemistry program. It was designed to deal with the Protein Structure Prediction (PSP) problem [19]. DEEPSAM's EP algorithm is written in Python 2.x on top of *mgtinker*, a modified version of the TINKER molecular modeling software package [9], which is written in Fortran 77. DEEPSAM's implementation contains dozens of thousands lines of Python and Fortran source code. The shortcoming of the current implementation, which we attempted to mitigate by using a STM-based approach, is barrier-based process synchronization, which limits full potential of parallel processes execution.

The main goals of this research are: (i) to develop a new STM-based DEEPSAM architecture in order to enhance the existing barrier-based process synchronization of the original version of DEEPSAM utilizing STM for Python language, named as Python Software Transactional Memory (PSTM) [10], and (ii) to try to quantify the impact of PSTM on the system performance of such a complex program like DEEPSAM. Although the first goal is related to DEEPSAM in particular, it provides details about the complexity and potential labor effort needed for the integration of a STM into a quite complex real-world application in general. The attainment of the second goal provides a detailed analysis about how PSTM influences system execution time and problem size scalability.

Since DEEPSAM is a program, which implements an evolutionary algorithm, this case study may be valuable for other researchers who extensively use and exploit the power of evolutionary and genetic algorithms, such as parallel-series electrical systems design optimization [11], power control optimization [12], multi-objective optimizations [13], machine design development [14], *etc*.

In earlier research in the field, very few notable examples, which tried to quantify a STM-based solution in a real-world application, exist. Usually, they are evaluated against benchmark programs, which are convenient for early experiments, but may not be applicable for complex programs such as DEEPSAM. Among the first researchers who demonstrated a rather complex real-world application based on a STM are Zyulkyarov *et al*. [15] and Gajinov *et al*. [16]. In their work, they experimented with the internationally famous multi-player game Quake, using different versions of C/C++ based STM. Nakaike *et al*. [17] tried to overcome the scalability issue of JAVA-based applications, namely HSQLDB database and Geronimo and GlassFish application servers, using STM-based approach. Hofmann et al. [18] analyzed how hardware TM influences an operating system architecture and performance.

2. DEEPSAM

The PSP problem [19] entails the challenge of predicting

[1] University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovica 6, Novi Sad, Serbia, branislav.kordic@rt-rk.uns.ac.rs
[2] Jerusalem College of Technology, Jerusalem 9372115, Israel, goldmosh@g.jct.ac.il

the native structure of a bio-molecule. Actually, PSP is the problem of finding the global minimum of a bio-molecule's Potential Energy Surface (PES), which is a multi-dimensional function having approximately 10^N local minima (N is the length of the bio-molecule sequence).

Certain PSP computational tools rely on predefined sub-structure fragments. These tools include PEP-FOLD [20], ROSETTA [21], PEPstr [22], PEPStrMod [23], *etc*. On the other hand, other tools implement methods that do not rely on predefined sub-structure fragments, but, on the bio molecule's sequence and on the PES modeling capabilities of purely ab-initio empirical Molecular Mechanics (MM) force fields, only. Examples of such methods include Diffusion Equation Method (DEM) [24], Molecular Dynamics (MD) [25], Monte Carlo (MC) [25], Simulated Annealing (SA) [26] in its two versions, MDSA and MCSA, Replica Exchange Molecular Dynamics (REMD) [27], etc. DEEPSAM implements a hybrid EP algorithm whose mutation operators, called DEMSA, combine tools of this kind. In Table 1, we present a qualitative comparison of those PSP tools. In our earlier research [4, 29, 30], DEEPSAM's protein structure prediction quality was scored by the results of calculating root mean square deviation (RMSD) and rank of maximal substructure (MAXSUB), among the calculated 3D geometry structures and those obtained experimentally. Description of those structural similarity comparison techniques is provided in [4].

The default population size ($n > 1$) used in this research is 5, the same as in previous studies in which DEEPSAM produced successful bio-molecules' structure predictions.

The population-oriented approach of EP provides a good sampling of the exponential PSP search space by exploring in parallel a set of widely distributed sub-regions of the PES. A set of specially designed mutation operators, called DEMSA, which are generated on-the-fly, at run time, combine complementary advantages of DEM, MDSA, and the L-BFGS quasi-Newton local minimization procedure [28]. For each one of the five bio-molecule's conformations in the population, five probabilistically chosen different DEM smoothing levels are applied upon the PES. This generates 25 different smoothed PESs that have *fewer minima* relative to the bio-molecule's (unsmoothed) PES, achieving *reduced search spaces,* which are simultaneously sampled, in parallel. Each one of those DEMSA operators has two optimization steps. At the first step, it applies MDSA (or L-BFGS) at its smoothed PES, upon the corresponding conformation. At the second step, it applies MDSA (or L-BFGS) at the un-smoothed PES, upon the conformation calculated at the first step. MDSA is used if DEEPSAM decides that a *long step size* minimization is needed. L-BFGS is used if DEEPSAM decides that a *small step size* minimization is needed. For more details, see [7, 8]. The probabilistic choice of smoothing levels, together with the choice of MDSA and/or L-BFGS at the smoothed and/or un-smoothed PESs, may cause the runtime of each iteration to defer significantly from the run time of any other iteration. This means that we may expect the run times of two DEEPSAM runs to defer significantly.

At each iteration, DEEPSAM implements two levels of parallelism: a process is created for each one of the 5 molecular conformations in the current population. Each one of those 5 parent processes creates 5 child processes that accordingly run 5 different DEMSA operators, each one upon a *replica* of the same molecular conformation (Fig. 8 [4]). In other words, DEEPSAM's behavior, as explained here, has some similarity to REMD's behavior.

DEEPSAM provides four alternative run modes, namely *serial*, *serial-parallel*, *parallel-serial*, and *parallel*. Here we focus exclusively on the *parallel* mode, in which all the processes are executed in parallel.

In [4] the prediction of the structure of a set of cyclic peptides, for which X-ray crystallographic data are available, was used as DEEPSAM's proof of concept. As part of that study, DEEPSAM's structure prediction capabilities were compared with its components': *nwsapss* (one of DEEPSAM's DEMSA operators), *anneal* (Tinker's MDSA implementation), and *pss* (Tinker's DEM implementation). Two kinds of comparisons were implemented: by their run times (Table 8 [4]) and by the calculated deepest minima of the PESs of the studied cyclic peptides (Table 9 [4]). Those comparisons showed that DEEPSAM was better than any of its components alone. DEEPSAM was also used to study the relative influence of widely used Implicit Solvent Models on structure prediction [29]. DEEPSAM was also used to predict the structure of linear peptides whose lengths are between 10 and 20 amino acids [30]. The predicted structures were compared with available NMR structures and with structures predicted by PEP-FOLD [20]. Run time detailed information of those calculations may be found in Table 2S of the Supporting Information of [30]. In that study, DEEPSAM was shown to be a good structure predictor relative to the structure prediction capabilities of PEP–FOLD.

3. **PSTM**

The Python Software Transactional Memory (PSTM)

Table 1
A qualitative comparison of Protein Structure Prediction tools

Tool	Approach	Calculated minima for cyclic peptides	Calculated minima for linear peptides	Comments
Tools that implement methods that rely on predefined sub-structure fragments				
PEP-FOLD, ROSSETTA, PEPstr, PEPStrMod		-	Compared	
Tools that implement methods that rely only on bio-molecule's sequence and PES modeling				
DEM	Global Minimization by PES Smoothing.	Compared		
MDSA	Global Minimization by Molecular Dynamics. (MD) Simulated Annealing (SA).	Compared	-	
L-BFGS	Quasi-Newton Local Minimization.	Compared	-	
REMD	Global Minimization by MD parallel execution upon molecular conformation replicas.	-	-	DEEPSAM's DEMSA operators are also executed upon replicas.
DEEPSAM	Global Minimization by EP whose DEMSA mutation operators execute in parallel upon molecular conformation replicas.	Best (Table 9 [4])	Best (Table I [30])	

[10] system comprises (i) software transactional memory, *i.e.*, PSTM, and (ii) transactions. As the central component of PSTM, transactional execution is provided through its public API. In the PSTM execution model, a transaction starts execution by reading data from PSTM, then follows data processing, and finally, the transaction ends with writing (updating) data back to PSTM. Each transaction is part of an application, and it is executed in the context of that application. A transaction is executed sequentially as a set of instructions, possibly sharing common transactional variables with other transactions. From the application's perspective, a transaction is executed atomically, and its outcome is either true (*commit*) or false (*abort*).

Two versions of PSTM exist, one for Python 3.x and other for Python 2.x. Both are built on top of Python's *multiprocessing* package. PSTM's version for Python 2.x was used because the current version of DEEPSAM is implemented in Python 2.x.

The API of the Python 2.x version of PSTM follows:

- *new*(*comm_links*)
- *addVars*(*comm_links, vars*)
- *getVars*(*comm_links, vars*)
- *cmpVars*(*comm_links, vars*)
- *commitVars*(*comm_links, rw_sets*)
- *delete*(*comm_links*)

The function *addVars*() adds new t-var(s) in PSTM. The function *getVars*() is used to read t-var(s) from PSTM. The function *cmpVars*() is an auxiliary function used to compare a set of t-vars defined by the transaction with the current t-var versions stored within PSTM. All the functions share a common argument *comm_links* used for communication between PSTM and transactions. The function *commitVars*() tries to commit (write) new values to PSTM. The argument *rw_sets* contain two sets of t-vars, a read set and a write set. The read set contains transaction's local t-vars which are used for processing. The write set contains new t-vars values ready to commit (write). The functions *new*() and *delete*() create and delete a set of *comm_links*.

4. IMPLEMENTATION

4.1 DEEPSAM ARCHITECTURE

DEEPSAM starts the execution with the main process *degsam*, which is located in the *degsam.py* module. Based on the input arguments, it executes the main iteration loop of DEEPSAM's EP algorithm. The main synchronization functionality is located in the *demsa.py* module. It coordinates the creation (spawning) and the synchronization of Python processes which constitute the first level of parallelism. Dictated by the fact that the algorithm processing is implemented in two distinct execution environments, namely Python and Fortran, and due to prior design choices, the processes' synchronization is facilitated by using the operating system's file system.

The central data structure, which contains substantial details for child processes, is the *degsamdictonary* dictionary file. In the first iteration, the dictionary is created based on the input parameters and it is regularly updated throughout the execution. The main process distributes it to its child processes by storing it in the file *degsamdict.dat*. The child processes (the first level of parallelism), which are Python scripts, only read it; later, each one of them spawns a family of Fortran child processes (the second level of parallelism). After this point, depending on the run mode, processes are executed in parallel. When the Fortran programs end their execution, their parent Python process summarizes the results computed by its children. When all the *degsam*'s child processes end their execution (Python processes), and when the control returns to the *degsam* process, the main process takes the data produced by its child processes and updates the dictionary file. Afterwards, the *degsam* process enters the next iteration, and the cycle described above repeats again, until all the iterations of the evolutionary loop gets to its end.

The shortcomings of this process synchronization approach are as follows: (i) input data to the child processes are passed over a set of read and write I/O operations, which are sluggish, and (ii) using a busy-waiting loop as synchronization barrier for the Python processes which creates a bottleneck.

The file *degsamdict.dat* is updated when the function *writedict*() is called. The function is called only from the *degsam* program and the *demsa* module, and it was a good starting point to introduce PSTM.

4.2 PSTM-BASED DEEPSAM ARCHITECTURE

The PSTM integration was accomplished in four steps. In the first step, all the modules which used the obsolete Python process control interface were identified. In particular, in order to avoid shared files between Python processes in the first level of parallelism, all the processes had to be created using the *multiprocessing* package. The *multiprocessing* package enabled us to utilize inter-process communication facilities, such as queues and pipes, which are essential for PSTM functionality. Only two DEEPSAM modules were modified: the *degsam* module and the *demsamutation* module.

In the second step, the initial PSTM-into-DEEPSAM integration was performed. The aim of this step was to run PSTM inside DEEPSAM without interfering with the main execution path of the algorithm. The PSTM functionality was implemented in the module *stm.py*. We had gradually shifted the existing DEEPSAM functionality to use PSTM, because of DEEPSAM's complexity and size. In this step, PSTM is successfully initialized and cleaned-up by the functions *init_pstm*() and *cleanup_pstm*(), respectively. The function *init_pstm*() creates communication links, initializes t-vars, and runs the PSTM server. The function *cleanup_pstm*() deletes all the t-vars and the communication links, and stops the PSTM server. The message sequence chart diagram of the PSTM-based-DEEPSAM architecture is shown in Fig 2. A set of new PSTM-related functions, which were added during the integration process, are given in Table 2.

In the third step, all the Python processes of the first level of parallelism, including the main process *degsam*, were supplied with a set of pipes *comm_links* used for the communication with PSTM. As function argument, the *comm_links* set is passed down to the functions *popdemsa()* and *demsamutation_main_program()* in the *degsam.py* and *demsamutation.py* modules, respectively. Each new argument, which is added to a function's argument list, is placed in the first position. This argument passing convention is defined since some of the existing arguments were defined with default values which must be at the end of the formal argument list. The utility function *launch*() was also modified – it takes a set of pipes and a process

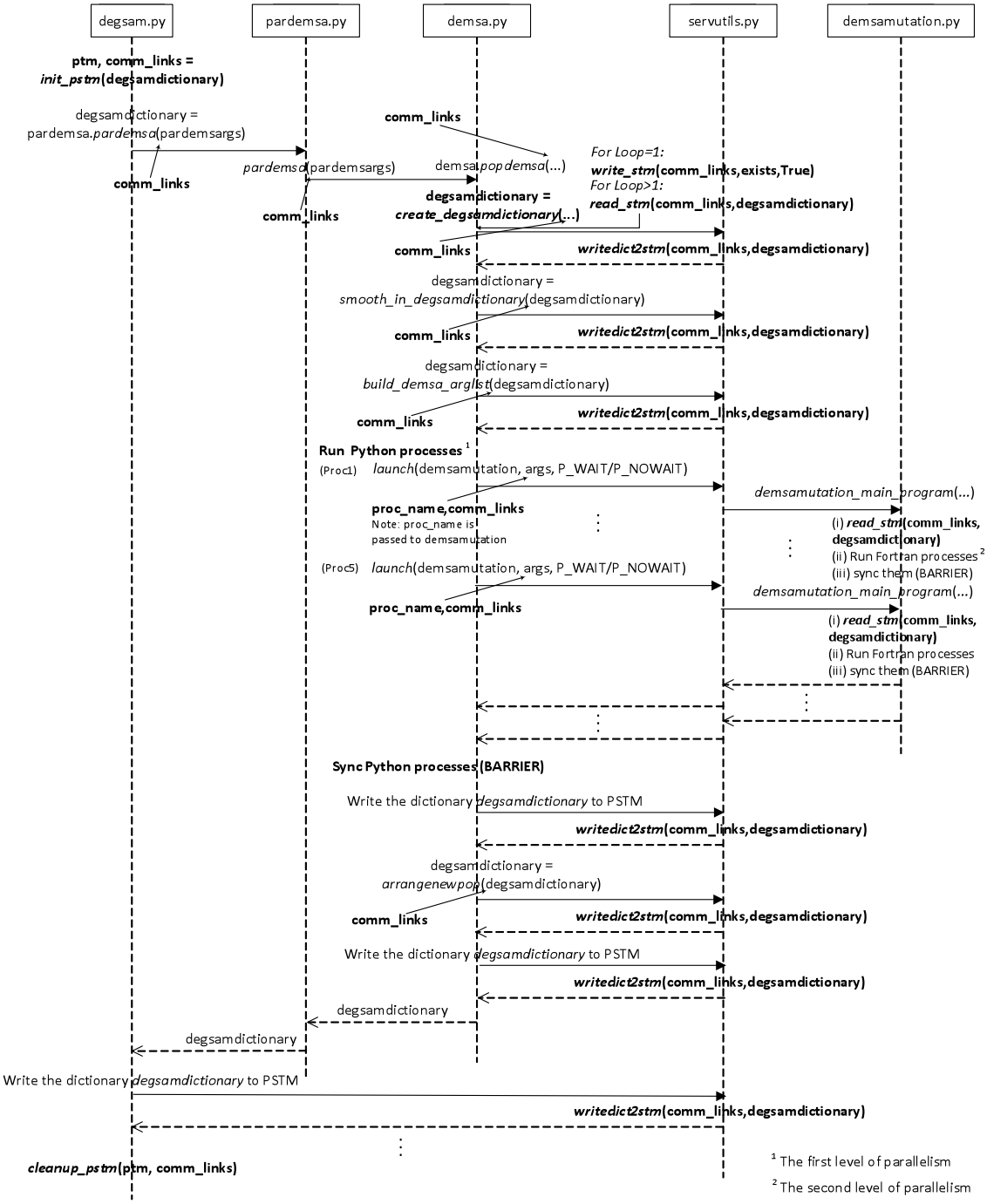

Fig. 2 – The message sequence chart diagram of the PSTM-based DEEPSAM architecture.

Table 2

A set of new PSTM-related functions

Function name	Description
writedict2stm(comm_links, degsamdictionary)	A version of the function *writedict()* which is PSTM based. It is used to write an object of *degsamdictionary* to PSTM.
read_stm(comm_links, tvar_name)	General purpose function used to read a t-var (*i.e.*, data) from PSTM.
write_stm(comm_links, tvar_name, new_tvar_value)	General purpose function used to write a t-var (*i.e.*, data) to PSTM.
init_pstm(degsamdictionary)	The function used for PSTM initialization – create communication links, run the PSTM server, add t-vars and initializes them.
cleanup_pstm(ptm, comm_links)	Releases all the resources allocated by the function *init_pstm()* and shuts-down the PSTM server.

name, which is used for Python child processes spawning. After this step, all the Python processes were equipped with the facilities needed to start using the PSTM functionality.

In the fourth step we had to introduce t-vars to PSTM, and to switch to PSTM-based DEEPSAM functionality. In terms of the transactional system, such as PSTM, the child processes of the first level of parallelization are transactions and their input-output dictionaries are t-vars. Therefore, now, instead of writing the content of the *degsamdictonary* dictionary to a file, the main process stores it in the PSTM, and child processes read it without issuing any file I/O operations. Similarly, instead of writing to output files, child processes write the results to the PSTM, making them available to the main process.

The dictionary is stored to PSTM by the *writedict2stm*() function, which is the counterpart of the function *writedict*() that relies on the file *degsamdict.dat*, *i.e.*, the function *writedict2stm*() writes a *degsamdictionary* object to PSTM instead into a file. The t-var *exist* is introduced, too. It is used in the first iteration of the evolutionary loop to check whether the dictionary exists. If it does not exist, it is created; otherwise it is loaded from the file system. The semantics of this t-var is the same as in the previous case, but with the difference that now it is stored in PSTM rather than in the file system. The functions *stm_read*() and *stm_write*() provide an interface for reading and writing a t–var(s) from PSTM, respectively.

5. **RESULTS**

5.1 TESTING AND VALIDATION

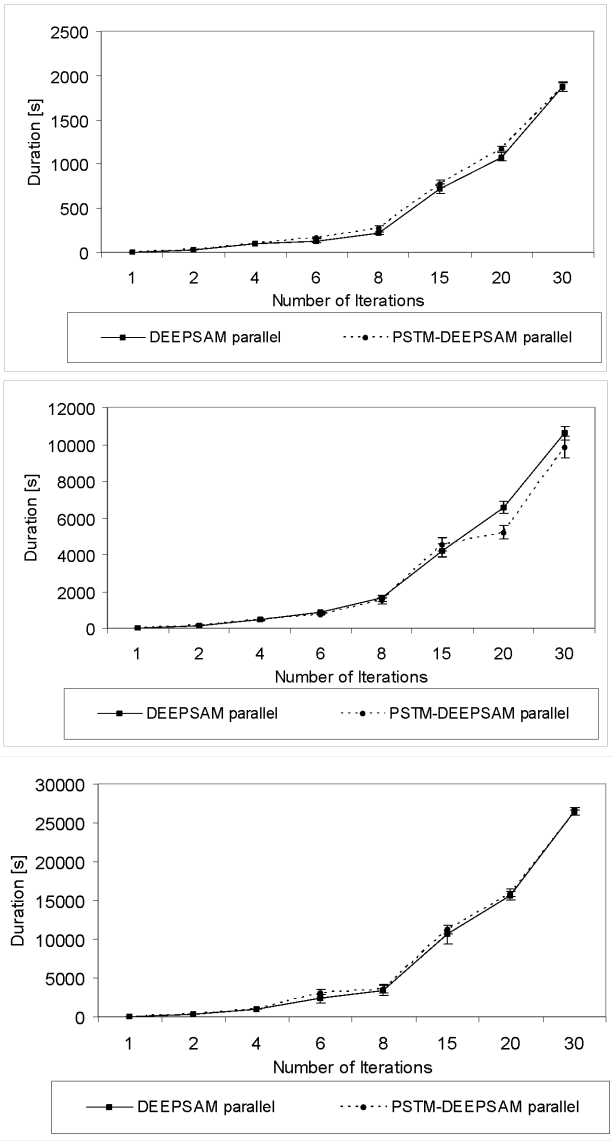

Fig. 3 – Execution time results for (a) enkephalin, (b) 2mq5 and (c) 1l2y in parallel execution mode.

The new PSTM-based DEEPSAM architecture was tested applying both gray-box and black-box testing approach. Namely, the gray-box testing approach was used in the earlier phases of development since it was not manageable to conduct overall system functionality testing and validation. Each function, which was modified to use PSTM functionality, was instrumented and its output was tested every time it was called. The testing was performed at run-time, in every loop of the evolutionary algorithm. The function testing includes checking the data of the *degsamdictionary* object which is stored in the operating memory (and shared through a file) and its counterpart object which is stored in PSTM. The data of those two objects must be bit-exact.

The black box testing approach was used in the later phases of development when it was possible to run the DEEPSAM version, which completely relies on the PSTM functionality. Since the evolutionary algorithm may produce different results in consecutive runs, bit-exact matching was not possible. Instead of validating the output results, in this phase the focus was to validate the program run-time execution behavior. During the testing, no errors, failures or undesirable behaviors were noticed.

5.2 EXPERIMENTAL RESULTS

The aim of the experiments was to quantify PSTM's influence on the performance metrics such as the system execution time and the problem size scalability. The amino acid sequence length identifies the problem size (the structural complexity of peptides and proteins), influencing execution time, and in some cases that influence is significant. In order to analyze the problem size scalability, DEEPSAM was run upon three bio-molecules: *enkephalin* [31](a short peptide), *2mq5* [32](a medium size peptide), and *1l2y* [33](a mini-protein).

The experiments were conducted in DEEPSAM's *parallel* execution mode and for 1, 2, 4, 6, 8, 15, 20, and 30 iterations. Each experiment was run five times. Based on the obtained execution time results, the average execution time and the standard deviation were calculated. Prior to the calculation, the outliers were filtered out. The measured execution times varied significantly; in general, the variation increased as the number of iterations increased. The calculated standard deviation actually expresses that significant variation, indicating the range of possible execution times, and the non-deterministic-like nature of DEEPSAM's algorithm.

The experiments were executed under Linux Centos 7 OS, running on top of a hardware machine equipped with two Intel Xeon CPU E5-2683 v3 @ 2.00GHz CPUs (total 28 CPUs) and 128 GB of RAM.

The execution-time results are presented in Fig. 3 (standard deviation is annotated by small vertical lines). The results show that execution time for both DEEPSAM versions are comparable for all the inputs and for all the number of iterations. In general, both DEEPSAM versions exhibit similar behavior, which is demonstrated by small differences in the execution time results. In some cases, execution times of the PSTM-based architecture are slightly better, which is consequence of reduced system overheads, such as Python process management overheads and replacing active busy-waiting synchronization with more efficient synchronization facilities. Actually, the busy-waiting loop synchronization in the original DEEPSAM architecture constantly occupies one of the underlying cores, which implies overall longer execution times for all participating processes in DEEPSAM. Note that protein execution times may be an order of magnitude longer than peptide execution times, thus, these slight improvements of execution times in the PSTM-based architecture can be significant in long computation runs. However, the non-

deterministic nature of the algorithm may cause wide range of possible execution time results.

Considering that size of PSP problems are identified with sequence length, the approach to the problem size scalability analysis taken here refers to the size and the complexity of the input bio-molecules. According to this scalability approach, and based on the results obtained by running experiments upon three peptides whose sizes and complexity are significantly different, it can be concluded that the PSTM-based DEEPSAM architecture retained the good problem size scalability performance of the original DEEPSAM.

The obtained results reveal sporadic glitches, which appeared during the execution of experiments. It would be preferable if DEEPSAM could be run without running anything else in the computer, *i.e.*, on the bare machine, but unfortunately, this is not feasible. It cannot be isolated (sandboxed) either. Therefore, some glitches may occur due to interference with OS services and other background processes, which we cannot control or prevent.

6. **CONCLUSIONS**

In this paper, we present a case study of STM integration in a real-world Computational Chemistry application based on evolutionary programming using DEEPSAM (a protein structure prediction program) and the Python Software Transactional Memory (PSTM). Starting from the original DEEPSAM version, the new PSTM-based DEEPSAM architecture was developed. The aim of the new architecture is to provide better performances for parallel computation by utilizing more power from multicore architectures. The experimental results indicate that the new PSTM-based architecture is able to achieve better execution performance and retain the problem size scalability without introducing any side effect to the existing architecture. Simultaneously, it enables the potential of lock-free facilities provided by the STM paradigm itself, which can be capitalized in the future by algorithm and architecture modifications.

As directions of future work we plan (1) to develop an algorithm that would fully exploit STM facilities and (2) to develop system architecture for distributed computing based on a framework such as EFL [34] and TensorFlow.

Received on September 16, 2019

REFERENCES

1. M. Herlihy, J. E. B. Moss: *Transactional memory: Architectural support for lock-free data structures*, Proc. of the 20th Annual International Symposium on Computer Architecture, pp. 289-300, ACM, New York, NY, USA (1993).
2. T. Harris, J. R. Larus, R. Rajwar, *Transactional Memory, 2nd edition*, Morgan and Claypool, 2010.
3. N. Shavit, D. Touitou: *Software transactional memory*, Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204-213,ACM, New York,NY, USA (1995).
4. M. Goldstein, E. Fredj, B. Gerber, *A New Hybrid Algorithm for Finding the Lowest Minima of Potential Surfaces: Approach and Application to Peptides*, Journal of Computational Chemistry, **32**, pp. 1785-1800 (2011).
5. M. Goldstein, *DEEPSAM: A Hybrid Evolutionary Algorithm for the Prediction of Biomolecules Structure*, Lect Notes in Comput Sc - Hybrid Metaheuristics, **9668**, pp. 218–221 (2016).
6. A. E. Eiben, J. E. Smith, *Introduction to evolutionary computing*, Springer,Berlin Heidelberg (2007).
7. T. Back, D. B. Fogel, Z. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operations*, IOP Publishing Ltd.,UK (2000).
8. L. J. Fogel, A. J. Owens, M. J. Walsh, *Artificial Intelligence through Simulated Evolution*, Wiley, USA (1966).
9. J. W. L. Ponder, *TINKER Molecular Modelling Package* (2003), https://dasher.wustl.edu/tinker/
10. M. Popovic, B. Kordic: *PSTM: Python software transactional memory*, Proc. of the 22nd Telecomm. Forum, pp. 1106–1109 (2014).
11. K. Guerraiche, M. Rahli, L. Dekhici, A. Zeblah, *Optimal design of redundancy series-parallel electrical systems using metaheuristics*, Revue roumaine des sciences techniques, **63**, *1* , pp. 46-51 (2018).
12. S. Ziane, A. Abdelghani, M. Abid, *Power control of doubly fed induction generator using hybrid adaptive neural fuzzy sliding mode controller optimised by genetic algorithm*, Revue Roumaine des sciences techniques, **63**, *4*, pp. 411–416 (2018).
13. H. Labdelaoui, F. Boudjema, D. Boukhetala, *Multiobjective optimal design of dual-input power system stabilizer using genetic algorithms*, Revue roumaine des sciences techniques, **62**, *1* , pp. 93–97 (2017).
14. A. Boulayoune, C. Guerroudj, R. Saou, L. Moreau, M. E. Zaim, *Optimization with particle swarm and genetic algorithm of flux reversal machine*, Revue roumaine des sciences techniques, **62**, *1* , pp. 19–24 (2017).
15. F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server*, Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 25–34 (2009).
16. V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguadé, T. L. Harris, M. Valero: *QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory*, Proc. of the 23rd International Conference on Supercomputing, pp. 126–135 (2009).
17. T. Nakaike, R. Odaira, T. Nakatani, M. M. Michael: *Real Java Applications in Software Transactional Memory*, Proc. of the IEEE International Symposium on Workload Characterization, pp. 1–10 (2010).
18. O. S. Hofmann, D. E. Portere, E. Witchel, C. J. Rossbach, H. E. Ramadan, A. Bhandari, *MetaTM/TxLinux: Transactional Memory for an Operating System*, ACM SIGARCH Computer Architecture News, **35**, *2* , pp. 92–103 (2007).
19. H. A. Scheraga, J. Lee, J. Pillardy, Y. J. Ye, A. Liwo, D. Ripoll, *Surmounting the Multiple-Minima Problem in Protein Folding*, J Global Optim, **15**, *3*, pp. 235–260 (1999).
20. A. Lamiable, P. Thevenet, J. Rey, M. Vavrusa, P. Derreumaux, P. Tuffery, *PEP-FOLD3: faster de novo structure prediction for linear peptides in solution and in complex*, Nucleic Acids Res, **44**, pp. W449–W454 (2016).
21. R.F. Alford, A. Leaver-Fay, J.R. Jeliazkov, *et al.*, *The Rosetta All-Atom Energy Function for Macromolecular Modeling and Design*, J. Chem. Theory Comput., **13**, pp. 3031–3048 (2017).
22. H. Kaur, A. Garg, G.P.S. Raghava, *PEPstr: a de novo method for tertiary structure prediction of small bioactive peptides*, Protein Pept. Lett., **14**, pp. 626-631 (2007).
23. S. Singh, H. Singh, A. Tuknait, K. Chaudhary, B. Singh, S. Kumaran, G.P.S. Raghava, *PEPstrMOD: structure prediction of peptides containing natural, non-natural and modified residues*, Biol. Direct., **10**, pp. 73 (2015).
24. J. Kostrowicki, H. A. Scheraga, *Application of the Diffusion Equation Method for Global Optimization to Oligopeptides*, J Phys Chem, **96**, *18*, pp. 7442–7449 (1992).
25. T. Schlick, *Molecular Modeling and Simulation – An Interdisciplinary Guide*, Springer (2002).
26. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *Optimization by simulated annealing*, Science, **220**, *4598*, pp. 671–80 (1983).
27. Y. Sugita, Y. Okamoto, *Replica-exchange molecular dynamics method for protein folding*, Chem Phys Lett, 314(1-2), pp. 141-151 (1999).
28. D. C. Liu, J. Nocedal, *On the Limited Memory BFGS Method for Large Scale Optimization*, Math Program, **45**, pp. 503–528 (1989).
29. Y. Goldtzvik, M. Goldstein, R.B. Gerber, *On the crystallographic accurace of structure prediction by implicit water models: Test for cyclic peptides*, Chem Phys, **415**, pp. 168–172 (2013).
30. M. Amitay, M. Goldstein, *Evaluating the peptide structure prediction capabilities of a purely ab-initio method*, Protein Eng Des Sel, **30**, *10*, pp. 723–727 (2017).
31. Y. Isogai, G. Nemethy, H.A. Scheraga, *Enkephalin: Conformational analysis by means of empirical energy calculations*, Proc Natl Acad Sci USA, **74**, *2*, pp. 414–418 (1977).
32. H. Mohanram, S. Bhattacharjya, *Cysteine deleted protegrin-1 (CDP-1): anti-bacterial activity, outer-membrane disruption and selectivity*. Biochim Biophys Acta, **1840**, *10*, pp. 3006–3016 (2014).
33. J. Neidigh, R. Fesinmeyer, N. Andersen, *Designing a 20-residue protein*. Nat Struct Mol Biol, **9**, pp. 425–430 (2002).
34. R. B. Yehezkael, M. Goldstein, D. Dayan, Sh. Mizrahi, *Flexible Algorithms: Enabling Well-defined Order-Independent Execution with an Imperative Programming Style*", Proc. of ECBS-EERC 2015, IEEE Press, pp 75–82 (2015).