LU FACTORIZATION USING MULTITHREADED SYSTEM

Mohammad Osama Badawy
College of Computing and Information Technology
Arab Academy for Science and Technology and Maritime Transport
Alexandria, Egypt
mohammad.osama.badawy@gmail.com

Yasser Y. Hanafy
Arab Academy for Science and Technology and Maritime Transport
Alexandria, Egypt
yhanafy@aast.edu

Ramy Eltarras
College of Computing and Information Technology
Arab Academy for Science and Technology and Maritime Transport
Alexandria, Egypt
ramy@aast.edu

***Abstract*—The problem of solving large systems of linear equations of the form (Ax = b) arises in various applications such as finite element analysis, computational fluid dynamics, and power systems analysis, which is of high algorithms complexities, that takes a lot of execution time. The high computational power required for fast solution of such problem is beyond the reach of present day conventional uniprocessor. Furthermore, the performance of using a system of uniprocessor tends to display an early saturation in relation to their costs. This implies that even modest gains in performance of a uniprocessor come at an exorbitant increase in its cost, that made the use of new technologies mandatory to minimize the execution time. This paper presents a parallel implementation of the classical solution of system of linear equations at high and reasonable speed up. The speed up achievement is obtained through the fine granularity in data and tasks, and asynchronicity to hide latency of memory access**

***Index Terms*—** *Multithreading, LU factorization, Task scheduling, Multicore, Many-core, Dense linear algebra*

I. Introduction

Parallel implementations of problems that requires solving large systems of linear equations; were limited primarily to multiprocessor computers [1,2,3]. However because of the exponentially increasing complexity of such applications, the high cost of supercomputing forces researchers to explore new, sustainable, and affordable high performance computing platforms. Configurable computing[3], where hardware resources can be configured appropriately to match specific hardware designs, has recently demonstrated its ability to significantly improve performance for computing intensive applications.

The solution of this problem involved direct and iterative methods. Direct methods involved Gaussian Elimination, LU Factorization, and Cholesky factorization, that yields an exact final solution by executing a predetermined number of arithmetic operations, although these methods are more computationally intensive but they are important for solving linear systems due to their accuracy, robustness, and generality.

In this paper, a parallel implementation of the classical solution of system of linear equations at high and reasonable speed up was introduced. This speed up can be obtained through the fine granularity in data and tasks, and asynchronicity to hide the latency of access to memory.

The rest of this paper is organized as: Section II holds the background; which includes parallel processing, Multithreading, and LU Factorization concepts. Section III holds literature review and related work. Section IV is the proposed solution for the system. Section V views the experimental results. Finally the conclusion is introduced in section VI.

II. Background

A. Parallel Processing

A formal definition of the parallel processing can be expressed as increasing the sampling rate by replicating hardware so that several inputs can be processed in parallel and several outputs can be produced at the same time.

"To pull a bigger wagon, it is easier to add more horses than to grow a gigantic horse", the previous paraphrased quotation is expressing the basic concept of parallel processing nicely and informally.[4]

The previously mentioned definitions express the power of processing with respect to some different tasks that are independent of each others. And can be implemented upon the architecture of the computer that can be a Multi-processors architecture, many-core architecture, or multi-core architecture, or even any combination of previously mentioned architectures.

With this mix of shared and private caches, the programming model for multi-core processors is becoming a hybrid between shared and distributed memory:

Core: The cores have their own private L1 cache, which is a sort of distributed memory. The above mentioned Intel 80-core prototype has the cores communicating in a distributed memory fashion[4].

Socket: On one socket, there is often a shared L2 cache, which is shared memory for the cores[4].

Node: There can be multiple sockets on a single 'node' or motherboard, accessing the same shared memory[4].

Network: Distributed memory programming is needed to let nodes communicate[4].

B. Multithreading

The word multithreading can be translated as many threads of control. While a traditional process always contains a single thread of control, multithreading separates a process into many execution threads, each of which runs independently.

The main benefits that arise from multithreading are:

- Improved application responsiveness and better program structure - any program in which many activities do not depend upon each other can be redesigned so that each activity is executed as a thread.
- Efficient use of multiple processors - numerical algorithms and applications with a high degree of parallelism, such as LU factorization, can run much faster when implemented with threads on a multiprocessor.
- Use fewer system resources - the cost of creating and maintaining threads is much smaller than the cost for processes, both in system resources and time.

All threads created from the same initial thread (standard process) exist as a part of the same process, sharing its resources (address space, operating system state ...). Beside that the multithreaded applications use fewer system resources than multiprocessing applications; communication between threads can be made without involving the operating system, thus improving performance over standard inter-process communication. From these reasons multithreading is so popular today, and modern operating systems support it.

Multithreading also brings some problems, like signal handling, function safety under possible parallel threads execution (parallel use and change of global variables), alarms, interval timers, and profiling. The problem is how to change this process oriented and defined elements to support threads and to be defined on a thread level. One of the main problems in this work was a time measurement (real, user and system time) for a single thread in a multithreaded application. This was resolved using specific system dependent features (interval timers provided for each lightweight process).

C. LU Factorization

LU factorization is a common algorithm used for solving systems of linear equations. This system can be presented in a matrix form:

$$\mathbf{A\,x = b}$$

Where **A** is the coefficient matrix, **x** is the unknown vector, and **b** is the excitation vector.

With LU factorization, the original coefficient matrix is factored, or decomposed, into the product of a lower-triangular matrix **L** and upper triangular matrix **U**. To attain a unique decomposition, the dialog terms of **L** or **U** are set to unity. After the decomposition is performed, the solution is determined by a forward substitution step and a backward substitution.

So far, eliminating unknowns was focused on. Now suppose it is needed to solve more than one linear system with the same matrix, but different right hand sides. Can the work done in the first system be used to make solving the second one easier?

The answer is yes. The solution can be split in a part that only concerns the matrix, and part that is specific to the right hand side. If a series of systems are to be solved, the first part will be done only once, and, luckily, that even turns out to be the larger part of the work.

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 2 & -5 & 12 \\ 0 & 2 & -10 \end{pmatrix}$$

In the elimination process, lower and the upper triangular matrices are saved in place as shown, while the Lower triangular matrix can be shown as

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}$$

The upper triangular matrix can be shown as

$$U = \begin{pmatrix} 1 & -2 & 3 \\ 0 & -1 & 6 \\ 0 & 0 & 2 \end{pmatrix}$$

This is the identity with the elimination coefficients in the first column, below the diagonal. It can be noticed that the first step in elimination of variables is equivalent to transforming the system Ax = b to LUx = b, where multiplying L and U will bring back A.

Then now A = LU; this is called an 'LU factorization'. It is noticed that the coefficients of L below the diagonal are the negative of the coefficients used during elimination. Even better, the first column of L can be written while the first column of A is being eliminated, so the computation of L and U can be done without extra storage, at least if it could be afforded to lose A.

III. Literature Review and Related Work

A. Related Work

The most common algorithm in scientific modeling and simulation is a sparse iterative solver such as Conjugate Gradient (CG). This CG scheme uses standard data structures for storing the sparse matrix A and vectors p, q, r. Only the nonzero of sparse matrix A and its corresponding indices are explicitly stored using a standard sparse format. The vectors p, q and r are stored as one dimensional arrays in contiguous locations in memory. A single iteration of CG requires one matrix-vector multiplication, two vector inner products, three vector additions and two floating point divisions. Among these operations, the matrix-vector multiplication dominates the computational cost accounting for more than 90% of the overall execution time. Due to the sparse nature of the matrix A, the number of floating point operations per access to the main memory is relatively low during matrix vector multiplication. Additionally, the access pattern of the elements in the vector p depends on the sparse structure of A.

To provide a multithreaded version of CG algorithm, the matrix was divided by row-wise. Each thread multiplies row block of matrix with the specific source vector members and stores at the destination vector members. Since the destination vector is not shared, this module has a perfect parallelism. However, sparse matrix algorithm could not benefit from cache as does in dense matrix since it lacks data reuse and locality [6].

The previously mentioned simulation resulted in almost optimal speed up using the multicore architecture, however, the results was measured regardless of the accuracy needed. Leonardo Jelenković [7] presented experiments with multithreading in several computationally intensive examples in which the time of parallel, multithreaded execution is significantly shorter than sequential, single-threaded. Algorithms used in these examples are not fully optimized for that wasn't the goal. The intention was to find out multithreading mechanism and structure and how to improve the performance on a multitasking, multi-user and multiprocessor operating system. The target operating system was SunOS on two processor workstation Ultra sparc 2.

Based on the results presented, it is concluded that the multithreading can improve the performance of given algorithms which are running on multiprocessor system. If threads run independently or with very low communication, speedup is only limited by the number of processors. If the communication between threads is heavier, speedup can be reached only if the time of computation between synchronization is at least several times greater than the synchronization time.

Although all results present execution times on a lightly loaded system, speedup is also reached at higher loads when all times increase but with almost the same ratio. Speedup slowly decreases as load increases, which is the result of a slower communication through the operating system.[7]

In [8] a parallel execution model for matrix factorization, that is not naturally bulk-synchronous, was described. Starting with the UPC language, which has one-sided communication via its global address space, locality control through the partitioning of the address space, and a static parallelism model with barrier synchronization which lends itself well to a bulk-synchronous style. Extensions of the basic UPC execution model to better support problems such as matrix factorization with interesting dependence patterns was explored.

In this discussion, dense LU factorization was introduced, which is simpler than the sparse case, but still has nontrivial dependencies that lead to many possible parallel schedules. In addition, the high computational intensity of dense LU factorization means that arithmetic unit utilization should be very high, and the prevalence of LU performance data across machines from benchmark implementations of LU sets a high standard for success. For an arbitrarily large dense matrix, the high computation to communication ratio leads to a computation that scales with machine size and processor performance, as long as the input matrix is large enough to mask communication, memory, and synchronization costs.

For small matrices or for sparse ones, the dependencies inherent in the algorithm can result in poor scaling due to memory costs, communication overhead, synchronization, and load imbalance. Two of the most common parallel LU factorization codes for distributed memory machines are from the ScaLAPACK library and the High Performance Linpack (*HPL*) benchmark used in determining the Top 500 list. Both of these codes are written for portability and scalability using the two-sided message passing model in MPI, and are written to keep the processors somewhat synchronized in order to manage the matching of sends and receives and the associated buffer space for messages. The ScaLAPACK code synchronizes for each distinct phase of the algorithm, while the HPL code allows for a statically determined amount of communication and algorithmic overlap.

In [8], an alternative parallelization strategy for the LU was presented based on the *PGAS* model in *UPC* augmented with an event-driven multithreaded execution model. The global address space provides for one-sided communication, which decouples data transfer from synchronization; the partitioning gives application control over data layout; and the multithreading relaxes the static (*SPMD*) model used in UPC. The code is designed with latency hiding as their primary goal, and the programmability and performance benefits of UPC's one-sided communication model was explored, coupled with a dynamic parallelism model. Scheduling decisions must be made to balance the needs of parallel progress, memory utilization, and cache performance.

Latency tolerance for both memory and network latencies has become increasingly important in high performance algorithm design, as latencies have remained relatively stagnant over years of tremendous gains in clock speed and bandwidth scaling. Current high performance implementations, such as *HPL* and ScaLAPACK, view each task's execution as a single thread of control, which unnecessarily serializes the computation, and results in ad hoc and restrictive solutions to latency hiding. Rather than designing an algorithm for a specific degree of parallelism, a dataflow interpretation of the algorithm with a multithreaded implementation is used, that exposes all available parallelism at runtime. Lewis and Richards used this dataflow approach successfully for LU in a shared memory parallel setting on up to 24 processors, but not in a scalable distributed memory context. This work was continued by Kurzak and Dongarra and colleagues for additional matrix factorizations on multicore processors. A mixed shared/distributed memory code in this style was also run on the NASA Columbia machine, using the data driven approach within shared memory [8].

The advantages of this approach was mentioned to be:

- Parallel progress is ensured by prioritizing panel factorizations and lower numbered trailing updates.
- Memory was controlled using our dependency based allocation scheme.

- Cache performance was maximized by increasing the size of the matrix operations.

While any process does not hold the same block of data each factorization iteration, which decrement the locality of reference and did not make an efficient use of locality. Also the communication over network is an overhead rather than being a speedup technique as the amount of data sent and received through the network is massive.

IV. Proposed System

Starting from partial pivoting by using the row pivoting only, then the main thread invokes a number of threads to be used during the factorization step as the number of threads to be used is mentioned in the beginning of the code.

It has been assumed that the rows of the matrix to be factorized are distributed among the threads working on it, using the thread number and the block size according to the number of threads. Thread numbering starts from 0 and ends with n-1. Each thread starts working on the block of rows belonging to its scope, since the scope is sent to the thread as a thread parameter. While the block size is calculated as ceil(s/n). Then the row and the column of the pivot is been removed from the matrix to be prepared for the next pass.

The complexity of the system, can be calculated as follows, in case of the single threaded run it has been found that the complexity of the algorithm will be roughly of $O(n^3)$ [9], in case of using multiple threads, the complexity should be decreased as the number of threads increases, but the minimum time can be reached is one half of the original execution time; as the algorithm is now implemented on a two processors system that can divide the work on a maximum of two processors simultaneously.

The algorithm then repeats the steps from the partial pivoting till there is only one element remaining; the matrix then is ready for the back substitution step and the matrix is factorized then.

The algorithm was divided into two parts, the first is considered to be done by the master thread, that is responsible for creating matrix, pivoting, creating the worker threads, setting assignments for threads, and finally synchronization between threads. The other part is related to the worker thread that is responsible only for the factorization of the partition assigned by the master then waiting for the next assignment.

This section presented the designed model of implementation for the LU Factorization over the multicore chips to predict the speed up achievement. The mentioned model expectation of achievement are made regardless of:

- Communication time, as it is as least as possible
- Matrices of small sizes, as the synchronization will be dominant

As the matrix is saved row wise, it is being clustered also row wise which means each consequent number of rows are assigned to one processor to avoid cache misses as much as possible, while maintaining the fine grain size not to keep processors idle, waiting for each others to finish and work on the same page at memory. The backward substitution is not mentioned as it is not of major effect in time, also it can be repeated many times for different solution vectors b.

The proposed system is a scalable one, as it can work on different architectures, holding the advantages of keeping the number of processors available opened, also the matrix size is opened. The input parameters for the proposed system is then the coefficient matrix, and the number of processors available.

The algorithm is working in a manner that, a master thread performing the whole thing till the factorization step, then it calls the worker threads to perform factorization over the number of rows allocated to each thread, while the pivot remains the same with each thread. Using threads guarantees the memory sharing while no locking for the elements to be worked on is made, so the matrix partitioning was taken care of to avoid overlapping between threads. The master thread starts by getting the matrix and the number of threads which is meant to be the number of processors available for usage.

Master thread tasks:

- Getting matrix
- Getting number of processors
- Pivoting
- Creating threads
- Dividing the submatrix
- Sending arguments to the worker threads
- Synchronization between threads after the factorization step is over

While the worker threads task is only performing factorization over the number of rows assigned by the master using the pivot received also from the master, then waiting for the next assignment to be done. Master and worker keeps on working till the matrix is fully factorized.

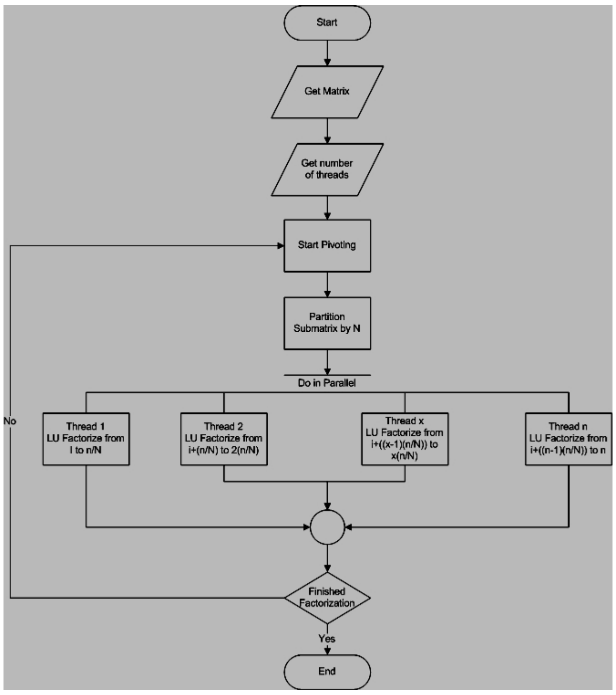
Start
Get Matrix
Get number of threads
Start Pivoting
Partition Submatrix by N
Do in Parallel
No
Thread 1 LU Factorize from I to n/N
Thread 2 LU Factorize from i+(n/N) to 2(n/N)
Thread x LU Factorize from i+((x-1)(n/N)) to x(n/N)
Thread n LU Factorize from i+((n-1)(n/N)) to n
Finished Factorization
Yes
End

Figure 1. Flow chart of the proposed system

The next section describes the experimental results, of the proposed system.

V. Experimental Results

The system has been run on three machines with three different architectures to ensure scalability and integrity, while the results was measured on each machine and compared to the single thread execution on the same machine to avoid misleading results. The operating system used was Linux UBUNTO, while the programming language used is C, and the compiler used is the default C compiler for C under Linux, gcc.

The first is a machine with an Intel Core 2 Duo CPU, with a 3.0 MB of cache memory. The second is a machine with an AMD X4 processor with 3.0 MB of cache size. Finally, a two chips machine each of quad-core with a total of 8 cores used, and a cache of 2.0 MB.

The system has been used through different sized matrices, with different number of threads as varying from a single thread to 100 threads. Matrices of sizes 1000x1000 up to 10000x10000 was used to illustrate the results.

The following set of figures demonstrates this relation between the two variants (Matrix size, and the Number of threads) with respect to the speed up.

The speed up measurement was made with respect to the execution time over the same machine for the single threaded approach.

A. First experiment

The first machine held only two cores, so a maximum of two threads was used, and a maximum speed up of 2 was expected, each run was made ten times and an average of the ten runs was calculated; finally the mean was plotted on the graph, with confidence rate of 95% and a confidence interval of 0.0086.

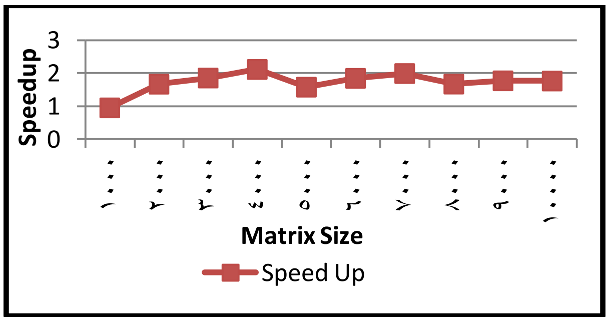

Figure 2. The speedup achieved using two processors

The maximum speed up achieved was 1.98 on the size of 7000, while the speed up converged at last at a value of 1.76 as illustrated in Figure 2.

B. Second Experiment

The second experiment was made on a machine holding four cores, so a maximum of 4 threads was used, and a maximum expected speed up is 4.

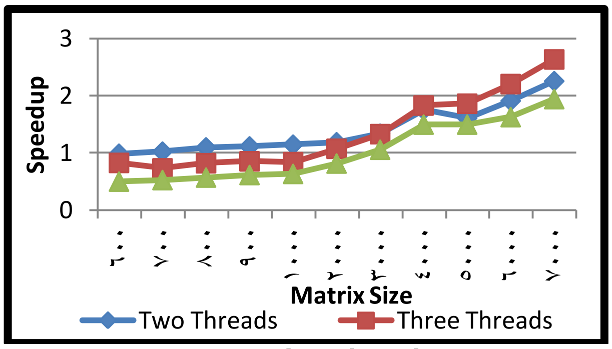

Figure 3. Speedup achieved using 4 cores

Each point on the graph was given by trying the same parameters to the algorithm, to be run 10 times and the average is plotted on the graph, with a confidence rate of 95% and a maximum confidence interval of 1.6, while the maximum speed up achieved was 2.63at the size of 7000.

C. Third Experiments

The last machine held two chips each of 4 cores, that offered a total of 8 cores, with expectation of a maximum speedup = 8.

For clarity, the graph is divided into two partitions, the first shows the results for using two, three and four threads, while the second plots the results of using five, six, seven and eight threads.

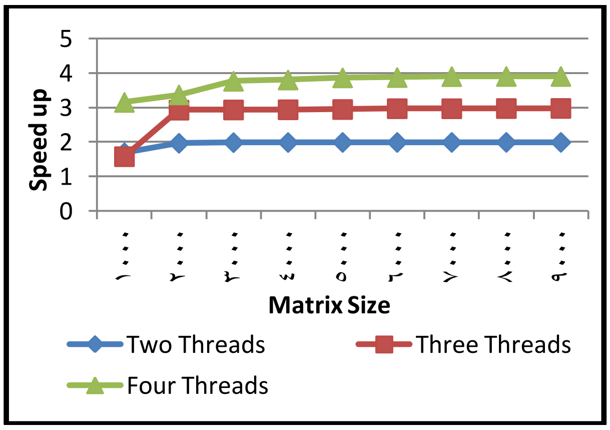

Figure 4. Speedup achieved using 8 cores

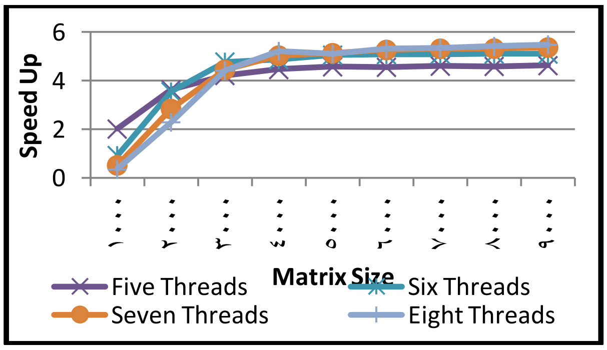

Figure 5. Speedup achieved using eight cores

The maximum speed up achieved using two threads was 190%, three threads 290%, four threads 390%, five threads 460%, six threads 510%, seven threads 530%, and eight threads at 550% speed up.

The multicore architecture raises a lot of challenging research problems regarding how to use the new architecture effectively. This paper discussed briefly the concept of the classical dense LU Factorization problem, and the multi/many core architecture. Inspired by effort made in order to decrease the execution time effectively and reasonably, we propose to use the affinity based thread scheduling to maximize the memory efficient usage of the number of cores and memory available. Multicore systems include not only shared-memory but also distributed-memory machines.

The execution time depends upon the number of processors that controls the number of threads and the memory available, as when running the system on Core 2 Duo processor, the previous results have been achieved. This proves that if the available memory satisfies the needs of all the number of threads being used, the overall execution time decreases with a ratio that is directly proportional to the number of processors.

VI. Conclusion

In this paper, firstly, an analytical model for parallelizing the solution of LU Factorization was introduced. The shared memory may reduce the number of cache misses if the data is accessed in common by several threads, but it may also result in performance degradation due to resource contention. We use the multithreaded approach over multicore architecture to achieve the estimations made.

To investigate the affinity based thread scheduling, we have proposed an analytical model to estimate the cost of running an affinity based thread schedule on shared-memory multicore systems. The model was implemented over three different architectures to evaluate the cost of executing the Multithreaded proposed model: two multicore architectures, one of two cores over one chip, and the other is of four cores on a single chip, a combined many/multi core architecture, to avoid misleading results. We apply the proposed system to randomly generated matrices with different sizes to ensure scalability. The estimated cost accurately predicts which number of threads will provide better performance. With the aid of the model, we are able to reach the optimal speedup over multicore architecture, on the other hand, the optimal speedup cannot be achieved in the case of many core architecture because of communication overhead. Due to the NP-completeness of the factorization problem, we extend the algorithm to support threads with data dependences. Our experimental results show that using the proposed Multithreaded model can improve the program performance greatly (by up to 400%) on multicore architecture over single chip.

The research finally focus our runtime system design on the performance scalability metric. At any single run, the number of processors available is a parameter that can be changed, also the matrix size is kept as an input parameter. The runtime system distributes data across different compute nodes to achieve scalability. We propose an algorithm to solve data dependences without process cooperation in a distributed way. The runtime system distinguishes thread roles of task generation, task computing, and data communication. Our experiments on shared-memory demonstrate that our runtime system is able to achieve good scalability.

The Multithreaded implementation of LU Factorization over multicore architecture was of great effect on the total execution time, and an achievement of doubling, tripling and quadrupling the speedup was made. On the other hand, the many core architecture suffered the communication between different chips which decreased the speedup not to be the optimal but a slight increase in the speedup achieved, so the maximum speed up using eight cores achieved was 550%.

References

[1] A. Ghali, A.M. Nevill and T.G. Brown, *Structural Analysis: A Unified Classical and Matrix Approach*, Taylor & Francis, 2003.

[2] J. Ogrodzki, *Circuit Simulation Methods and Algorithms*, CRC Press, 1994.

[3] Q.K. Zhu, *Power Distribution Network Design for VLSI*, Wiley-IEEE, 2004.

[4] V. Eijkhout, E. Chow, and R. van de Geijn, Introduction to High-Performance Scientific Computing, March, 2010.

[5] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *(submitted to) Communications of the ACM*, May 2008.

[6] Ingyu Lee, Analyzing Performance and Power of Multicore Architecture Using Multithreaded Iterative Solver , In *Journal of Computer Science 6 (4): 406-412,* 2010.

[7] Jelenković L, Čeko G O (2010). Experiments with multithreading in parallel computing. Faculty of Electrical Engineering and Computing, University of Zagreb. Department of Electronics, Microelectronics, Computer and Intelligent Systems. Unska 3, 10000 Zagreb, Croatia.

[8] Parry Husbands and Katherine Yelick , Multi-Threading and One-Sided Communication in Parallel LU Factorization, *SC07 November 10-16, 2007, Reno, Nevada, USA 2007 ACM.*

[9] Matrix Operations in: Introduction to algorithms 2nd Edition 2001. Coremen T H, Leiserson C E, Rivest R L, Stein C. P754.