Paper accepted for presentation at the 2011 IEEE Trondheim PowerTech

Power Flow Using Thread Programming

Hasan DAĞ
Dept. of Information Technology,
Kadir Has University,
Istanbul, 34083, Turkey
E-mail: hasan.dag@khas.edu.tr

Gürkan SOYKAN
Informatics Institute
Istanbul Technical University,
Istanbul, 34469, TURKEY
E-mail: gurkan.soykan@be.itu.edu.tr

***Abstract*—Power flow problem, which is one of the most important applications in power system, is solved using OpenMP Application Program Interface (API), a thread based programming technique for the shared memory programming environments. The parallel implementation of power flow is tested on a shared memory machine with cache-coherent non-uniform memory access (ccNUMA) architecture. Results are presented for various power test systems. The thread programming based power flow program decreases the runtime substantially.**

OpenMP lets a user concentrate on his/her own work rather than the parallel programming details. OpenMP is a set of compiler directives and requires minimal knowledge regarding parallel programming. It can easily be used not only on the shared memory systems but also on the systems with hyper threading and multi-core technologies as well. Thus, speeding up any kind of power system related study is now viable without having to have expensive parallel computers and extensive experience on theirs programming. Since todays' and future personal computers and servers will mostly be based on the multi-core technologies the proposed method is an alternative to hard-coded parallel programming.

I. Introduction

The power flow problem is a well known application in power systems. It is necessary for control of an existing system. It is used for obtaining the power system information in terms of bus voltages and line flows for a given load, generation, and network configuration. Bus voltages and line flows can then be used amongst many other studies for both analyzing the transient stability and for the observation of the state of the power system at hand. One of the most important power system problems is the dynamic stability problem, of which transient stability constitutes one part. Speeding up the transient stability analysis close to real-time for on-line purposes has been a desire but both the cost of the computing hardware and the difficulties of parallel programming have limited the process. Though there have been many attempts to parallelize the transient stability problem with considerable successes [1], [2], [3], [4], [5], the efforts have not reached to a point where the problem can be solved on-line in real-time.

Since power flow, which is simply a set of non-linear equations $f(x) = 0$ to be solved for an x^* such that $f(x^*) = 0$, is the base for most of the power system related studies, there have been a lot of work either to parallelize or somehow to come up with a new formulation using characteristics of power system to get a solution in shorter time as well [6], [7], [8], [9], [10], [11], [12]. Due to the recent development in the area of high-performance computing technology, the parallel programming is one of the ways to decrease the runtime of the power flow. There are mainly two different parallel programming models, which are message passing and threads, to parallelize the power flow. Latter one is used on shared memory or multi-core systems.

The goal of this study is to show that the parallelization of the power system studies in general but that of power flow problem in specific using OpenMP directives on the shared memory multiprocessor architectures not only can lower the computing time appreciably but it can also simplify the parallel processing work. Additionally, there is really no need for a super computer. The parallelization exploiting thread programming techniques can be done using PCs with more than one processor, which are common on todays servers even on desktops and laptops. Moreover, multi-core technologies are promising for thread programming usage. Taking full advantage of these technologies requires multithreaded implementation for any kind of power system studies.

II. Power Flow Problem

Power flow problem is defined as the calculation of bus voltage magnitudes and bus voltage angles in a given power system. The solution of the problem corresponds to the steady state of the system at hand. For any network, which is composed of n buses, the following equation can be written as

$$\bar{\mathbf{Y}}\bar{V} = \bar{I} \tag{1}$$

where $\bar{\mathbf{Y}}$ is the bus admittance matrix, $\bar{V}$ is the voltage vector and $\bar{I}$ is current injection vector. Moreover, the current injection for any bus can be represented

$$\bar{I}_i = \frac{(P_i + jQ_i)^*}{\bar{V}_i^*} \tag{2}$$

where P_i is real power injection at bus i, Q_i is reactive power injection at bus i and $\bar{V}_i^*$ is conjugate of bus voltage at bus i. Substituting (1) into (2), the power flow equations are obtained as

$$P_i + jQ_i = V_i e^{j\theta_i} \sum_{k=1}^{n} Y_{ik} e^{-j\alpha_{ik}} V_k e^{-j\theta_k} \quad i = 1 \cdots n \tag{3}$$

One can split (3) as:

$$P_i = V_i \sum_{k=1}^{n} Y_{ik} V_k cos(\theta_i - \theta_k - \alpha_{ik}) \quad i = 1 \cdots n \qquad (4)$$

$$Q_i = V_i \sum_{k=1}^{n} Y_{ik} V_k sin(\theta_i - \theta_k - \alpha_{ik}) \quad i = 1 \cdots n \qquad (5)$$

where V_i and V_k are voltage magnitudes at bus i and bus k, θ_i and θ_k are voltage angles for these buses. $Y_{ik}\angle\alpha_{ik}$ is the $(ik)^{th}$ element of the bus admittance matrix. When (4) and (5) are rewritten as mismatch equations, the following equations are obtained.

$$0 = \Delta P_i = -P_i + V_i \sum_{k=1}^{n} Y_{ik} V_k cos(\theta_i - \theta_k - \alpha_{ik}) \qquad (6)$$

$$0 = \Delta Q_i = -Q_i + V_i \sum_{k=1}^{n} Y_{ik} V_k sin(\theta_i - \theta_k - \alpha_{ik}) \qquad (7)$$

For each bus, there are two equations with four variables: bus real power, bus reactive power, voltage magnitude, and voltage angle. Two of these variables should be specified to solve the power flow problem. The system buses are categorized into three types: slack bus, PV bus, and PQ bus. Active and reactive powers are specified for a PQ bus. Active power and voltage magnitude are specified for a PV bus. For a slack bus, which is also called as a reference bus, voltage magnitude and voltage angle are specified. In this situation, if the system has (n-1) PQ buses and a slack bus, the number of unknowns, hence the number of power flow equations, are 2(n-1).

The Newton Raphson method is generally preferred due to its quadratic convergence to find a solution for the nonlinear equations. While using the Newton Raphson method in power flow problem, the linearization of the power flow equations produces a linear set of equations of the form,

$$-\mathbf{J} \begin{bmatrix} \Delta\theta \\ \Delta V \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix}$$

with a Jacobian of the following structure:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \Delta P_2}{\partial \theta_2} & \cdots & \frac{\partial \Delta P_2}{\partial \theta_n} & \frac{\partial \Delta P_2}{\partial V_2} & \cdots & \frac{\partial \Delta P_2}{\partial V_n} \\ \vdots & & \vdots & \vdots & & \vdots \\ \frac{\partial \Delta P_n}{\partial \theta_2} & \cdots & \frac{\partial \Delta P_n}{\partial \theta_n} & \frac{\partial \Delta P_n}{\partial V_2} & \cdots & \frac{\partial \Delta P_n}{\partial V_n} \\ \frac{\partial \Delta Q_2}{\partial \theta_2} & \cdots & \frac{\partial \Delta Q_2}{\partial \theta_n} & \frac{\partial \Delta Q_2}{\partial V_2} & \cdots & \frac{\partial \Delta Q_2}{\partial V_n} \\ \vdots & & \vdots & \vdots & & \vdots \\ \frac{\partial \Delta Q_n}{\partial \theta_2} & \cdots & \frac{\partial \Delta Q_n}{\partial \theta_n} & \frac{\partial \Delta Q_n}{\partial V_2} & \cdots & \frac{\partial \Delta Q_n}{\partial V_n} \end{bmatrix}$$

The compact notation of the Jacobian

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \Delta P}{\partial \theta} & \frac{\partial \Delta P}{\partial V} \\ \frac{\partial \Delta Q}{\partial \theta} & \frac{\partial \Delta Q}{\partial V} \end{bmatrix}$$

consists of four submatrices, each of which shows the partial derivatives of mismatch equations w.r.t. state variables. ΔP, ΔQ, ΔV, and $\Delta\theta$ are defined as

$$\Delta\theta = \begin{bmatrix} \Delta\theta_2 \\ \Delta\theta_3 \\ \vdots \\ \Delta\theta_n \end{bmatrix}, \qquad \Delta V = \begin{bmatrix} \Delta V_2 \\ \Delta V_3 \\ \vdots \\ \Delta V_n \end{bmatrix}$$

$$\Delta P = \begin{bmatrix} \Delta P_2 \\ \Delta P_3 \\ \vdots \\ \Delta P_n \end{bmatrix}, \qquad \Delta Q = \begin{bmatrix} \Delta Q_2 \\ \Delta Q_3 \\ \vdots \\ \Delta Q_n \end{bmatrix}$$

and the unknown variables can be updated by

$$\Delta\theta_i = \theta_i^{k+1} - \theta_i^k \qquad (8)$$

$$\Delta V_i = V_i^{k+1} - V_i^k. \qquad (9)$$

This linear system of equations is solved by either a direct method or by an iterative method. In this study, LU factorization based direct method is used to find the solution of the linear system. For each Newton Raphson step, **L** and **U** matrices are recomputed. Therefore, the LU factorization of the Jacobian matrix is the most time consuming stage of every Newton iteration.

III. Thread Programming with OpenMP

In the last decade, the CPU architectures have changed drastically. They are classified mainly as multiprocessor, hyperthreading technology, and multi-core in the development stage. Fig. 1 shows the configuration of each approach. To compare these developments, dual processor and dual core are used in this figure. Dual processor architecture acts as shared memory. Due to the development of CPU architecture, thread programming is a current issue in programming techniques. Thread programming is defined as code threading, which breaks up a programming task into subtasks. Each subtask is called a ”thread”. Threads run simultaneously and independently. Thread programming essentially makes use of computing of CPU technologies. This necessity increases enormously by the advent of multi-core processors [13].

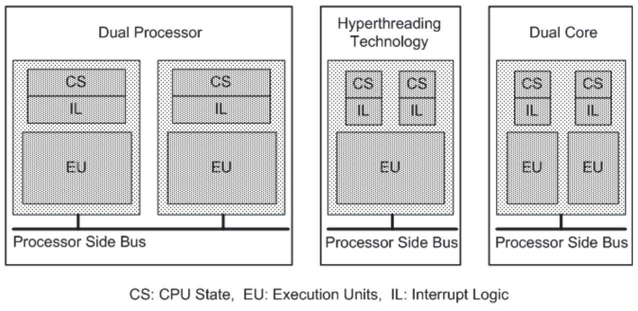


Fig. 1. CPU Architectures [13]

Posix threads and OpenMP are generally used as thread programming techniques on a computer that has shared memory based architecture. The advantage of low level posix threads is that it has more flexibility in creating and in killing of threads. However, it is harder to implement on parallel environments. OpenMP is a de-facto standard and high level programming language, which makes it easier to use for scientific applications [14].

OpenMP is a set of compiler directives and has became a parallel programming standard implemented on the shared memory architectures. The most popular distinctive features of it are being thread based and having explicit parallelism. The compiler directives are used for the construction of the parallel regions. They create threads in a given sequential program. Every OpenMP program begins as a single process that is a thread. This is called master thread and it executes the rest of the program sequentially until it encounters the first parallel region construct. At the beginning of the parallel region multiple threads are created according to the number of available processors or cores. Each thread has a thread number, which is an integer from 0 to the number of threads minus one. 0 shows the master thread. If any thread terminates within a parallel region, all threads in that region will terminate. The number of threads which will be used in the parallel region, can be controlled by using of the omp_set_num_threads() library function or setting of the omp_num_threads environment variable. It is important that one of them still behaves as the master thread. At the end of this parallel region all threads are terminated except the master. This type of model is described as the fork-join model. It is shown in Fig. 2. Fig. 3 also demonstrates OpenMP general code structure in C programming language. It simplifies loop level parallelism. In Fig. 3, *private* and *shared* are data scope clauses of an OpenMP directive. *private(var1)* means that each thread has its own copy of the var1 variable in the parallel region whereas, *shared(var2)* demonstrates that the var2 variable is shared among all threads in the parallel region. That is, each thread access the same memory locations for var2. The shared memory architectures provide loop level parallelism without decomposing data structures [15].

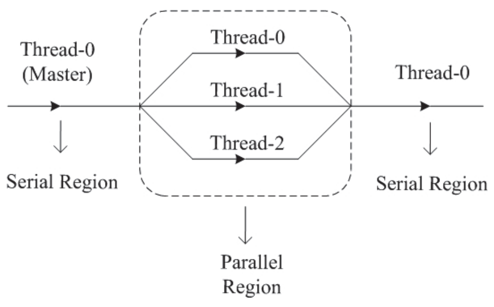
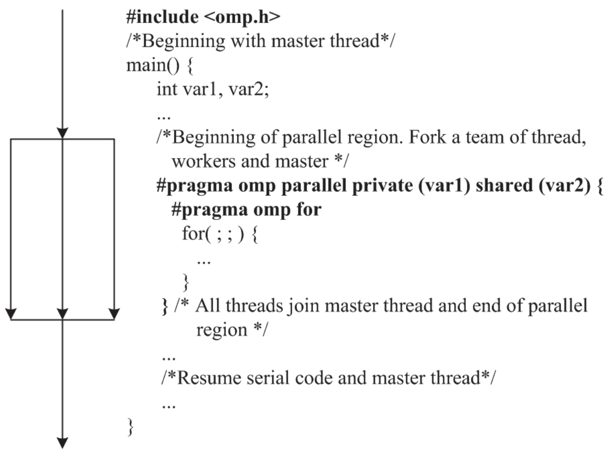


Fig. 2. Fork-Join model

OpenMP consists of the following directive types: parallel region construct, work-sharing constructs, combined parallel work-sharing constructs, synchronization constructs, and data environment [16].

OpenMP API exploits thread-level parallelism, which leads to its widespread use. It is not only used on shared memory multiprocessor architectures but also on-chip multiprocessor and on Pentium and Itanium architectures, for which compiler



```
#include <omp.h>
/*Beginning with master thread*/
main() {
    int var1, var2;
    ...
    /*Beginning of parallel region. Fork a team of thread,
      workers and master */
    #pragma omp parallel private (var1) shared (var2) {
      #pragma omp for
        for( ; ; ) {
          ...
        }
    } /* All threads join master thread and end of parallel
         region */
    ...
    /*Resume serial code and master thread*/
    ...
}
```

Fig. 3. OpenMP general code structure

and runtime support for OpenMP API is available and on multi-core architectures [17], [18], [19].

IV. An OpenMP Based Newton Power Flow

The Newton power flow solution consists of the following steps:

Algorithm 1

```
1: Read line and bus data from a file
2: Initialize the bus voltage (OMP)
3: Construct the bus admittance matrix
4: Calculate the bus powers (OMP)
5: Calculate power mismatches (OMP)
6: Check the convergence of the Newton method. If the
   convergence criterion is met, stop the iteration. If not go
   to Step 7.
7: Compute the elements of Jacobian matrix (OMP)
8: Compute LU factorization of Jacobian matrix and solve
   the linear system to find the voltage corrections (OMP)
9: Compute the new bus voltages (OMP) and return to the
   Step 4.
```

In the Algorithm 1, OMP indicates that the OpenMP directives are used in that step. The execution time of power flow solution reduces by using loop-level parallelism in this algorithm. To achieve loop-level parallelism, *parallel for* directive can be used in the algorithm. However, in our implementation, a *parallel* directive and a *for* directive, which is a work-sharing directive, as shown Fig. 4 are used instead of *parallel for* directive as shown Fig. 5. If several loops are needed to parallelize consecutively, one parallel region is created for all of them. Then, before each loop, *for* directive is written to execute the loop in parallel. This implementation brings about decreasing the execution time. In step 7, *reduction* clause is used with *for* directive. This clause allows safe global summation for a variable in the parallel region.

In order to find the voltage corrections in step 8, LU factorization of Jacobian is obtained and the system of linear

```
#pragma omp parallel private(i)
{
      #pragma omp for
         for(i=0; i<N; i++) {
               ...
         }
      #pragma omp for
         for(i=0; i<N; i++) {
               ...
         }
      #pragma omp for
         for(i=0; i<N; i++) {
               ...
         }
}
```

Fig. 4. The usage of a *parallel* and a *for* directives to parallelize three loops

```
#pragma omp parallel for private(i)
         for(i=0; i<N; i++) {
               ...
         }
#pragma omp parallel for private(i)
         for(i=0; i<N; i++) {
               ...
         }
#pragma omp parallel for private(i)
         for(i=0; i<N; i++) {
               ...
         }
```

Fig. 5. The usage of a *parallel for* directive to parallelize three loops

equations is solved. In our implementation, the linear algebra routines from Sun Performance Library are used to achieve Step 8. These routines consists of Blas Level 3 and Lapack [20]. This library is a highly tuned shared memory parallelized library. In virtue of this parallel property, the routines from the library are preferred in this implementation.

V. Test Results

To show the effectiveness and ease of use of OpenMP, we apply it to power flow problem using some of the standard test systems and some synthesized systems. All simulations are conducted on a SunFire 12K high-end-server, which has 16-900 MHz and 16-1200 MHz Ultra-SPARC III Cu processors with a total of 64 GB RAM on a shared memory architecture.

The optimized sequential power flow code is parallelized using OpenMP directives. We do not employ sparse matrix techniques for the power flow problem and we use Sun Performance Library when solving the $\mathbf{A}x = b$ system as part of the Newton-Raphson method. For the rest of the problem, OpenMP directives are inserted into the serial code to parallelize. Test networks used are some IEEE standard test cases (118 and 300 buses) and some synthesized networks (686, 864, 1400 and 1944 buses) [10].

The performance of parallel implementation is denoted by speedup which is the ratio of the runtime of the serial solution to that of the parallel solution. The value of speedup, Spd, is expected to be between 0 to n, which is the number of processors used. IEEE standard test cases results are left out from table because they are not big enough to show the performance of parallelization. Parallel power flow speedup results are shown in Table I.

TABLE I
ABSOLUTE SPEEDUP FOR DIFFERENT TEST CASES

	Number of Processors		
Case Name	**2**	**4**	**8**
SYN686	1.67	2.61	3.66
SYN864	1.67	2.83	4.28
SYN1400	1.82	3.12	5.13
SYN1944	1.85	3.36	5.67

In addition to speedup, parallel efficiency can also be used to show the performance of parallel implementation. For n processors case, it is defined as:

$$E = \frac{Spd}{n}. \tag{10}$$

The value of efficiency is expected to between 0 to 1. The parallel efficiency is affected by the amount of overhead. The overhead, due to OpenMP constructs, depends on the number of processors used [21]. When the number of processors is increased, the overhead also increases. The increased overhead leads to a decrease in parallel efficiency. Parallel efficiencies for our implementation are given in Table II.

Table II shows that when the problem is kept constant but the number of processors is increased the efficiency decreases as well. This is due to overheads, such as thread creation, collection of results, process id checking etc. Much larger systems need to be solved in order to utilize processing units at hand with larger efficiencies.

TABLE II
EFFICIENCY FOR DIFFERENT TEST CASES

	Number of Processors		
Case Name	**2**	**4**	**8**
SYN686	0.84	0.65	0.46
SYN864	0.84	0.71	0.54
SYN1400	0.91	0.78	0.64
SYN1944	0.93	0.84	0.71

Fig. 6 is plotted using Table I. It shows how the size of test system affects the performance of parallel power flow. The speedup is getting closer to the ideal speedup for large cases.

VI. Conclusion

The results show that the parallelization of the Newton-Raphson power flow using OpenMP directives is quite effective. This work was performed on a shared memory multiprocessor architecture. However, the Pentium and Itanium

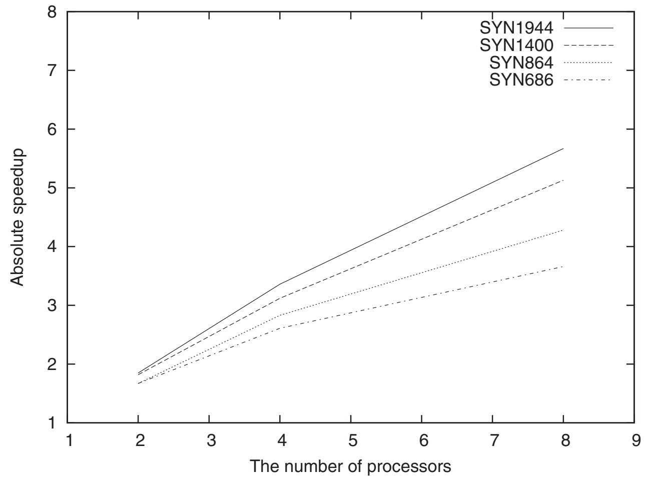


Fig. 6. Absolute speedup factors

architectures also provide the same opportunity. Nowadays, because of rapid growth of the usage of multi-core processors in current computers, the usage of OpenMP API is the simplest parallel techniques to reveal the compute power of processors. Therefore, the Intel C++ and Fortran compilers, which support the OpenMP directives on Linux and Windows platforms, can be used to solve power flow problem. Hence, low-cost parallelization of other types of power system studies, such as contingency analysis, state-estimation, transient stability analysis etc., is now possible. Low-cost refers to the cost of both hardware and human power (programming effort).

References

[1] D. J. Tylavsky, A. Bose, F. L. Alvarado, R. Betancourt, K. Clements, G. Heydt, G. Huang, M. Ilic, M. LaScala, M. Pai, C. Pottle, S. Talukdar, J. VanNess, F. Wu, "Parallel processing in power system computation," IEEE Transactions on Power Systems, vol. 7, no. 2, pp. 629–637, May 1992.

[2] M. A. Pai, P. W. Sauer, A. Y. Kulkarni, "A preconditioned iterative solver for dynamic simulation of power systems," International Symposium on Circuits and Systems, pp. 1279–1281, April/May 1995.

[3] M. A. Pai, A. Y. Kulkarni, "A simulation tool for transient stability analysis suitable for parallel computation," 4th IEEE Conference on Control Applications, pp. 1010–1013, September 1995.

[4] A. Padilha, H. Dağ, F. L. Alvarado, "Transient stability analysis on newtork of workstations using PVM," The 4th IEEE International Conference on Electronics Circuits and Systems, December 1997.

[5] J. Shu, W. Xue, W. Zheng, "A parallel transient stability simulation for power systems" IEEE Transactions on Power Systems, vol. 20, no. 4, pp. 1709–1717, November 2005.

[6] J. Q. Wu, A. Boje "Parallel solution of large sparse matrix equations and parallel power flow" IEEE Transactions on Power Systems, vol. 10, no. 3, pp. 1343–1349, August 1995.

[7] A. Semlyen, F. de Leon, "Quasi-newton power flow using partial jacobian updates," IEEE Transactions on Power Systems, vol. 16, no. 3, pp. 332–339, August 2001.

[8] F. Tu, A. J. Flueck, "A message passing distributed memory parallel power flow algorithm," IEEE Power Engineering Society Winter Meeting, vol. 1, pp. 211–216, January 2002.

[9] F. de Leon, A. Semlyen, "Iterative solvers in the newton power flow problem:preconditioners, inexact solutions and partial jacobian updates," IEE Proceedings Generations Transmission and Distribution, vol. 149, no. 4, pp. 479–484, July 2002.

[10] H. Dağ, A. Semlyen, "A new preconditioned conjugate gradient power flow," IEEE Transactions on Power Systems, vol. 18, no. 4, pp. 1248–1255, November 2003.

[11] Y. Chen, C. Shen, "A jacobian-free newton-GMRES(m) method with adaptive preconditioner and its application for power flow calculations,"IEEE Transactions on Power Systems, vol. 21, no. 3, pp. 1096–1103, August 2006.

[12] Y. Zhang, H. Chiang, "Fast newton-FGMRES solver for large-scalepower flow study," IEEE Transactions on Power Systems, vol. 25, no. 4, pp. 769–776, May 2010.

[13] C. Szydlowski, "Multithreated technology and multi-core processors," infrastructure Processor Division Intel Corporation, 2005 May 01.

[14] M. Bull, "An introduction to openmp," Hpc - Europa Surgery Presentation at HLRS, 2006 February 25.

[15] L. Dagum, R. Menon, " Openmp: An industry-standard api for shared - memory programming, " IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46–55, May 1998.

[16] R. C. et al., *Parallel Programming in OpenMP*. London: Academic Press, 2001.

[17] M. Sato, "Openmp : Parallel programming api for shared memory multiprocessors and on-chip multiprocessors," 15th International Symposium on System Synthesis, pp. 109–111, October 2002.

[18] M. G. X. Tian, S. Shah, D. Armstrong, E. Su, P. Petersen, "Compiler and runtime support for running openmp programs on pentium- and itanium-architectures," Proceedings of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments, pp. 47–55, April 2003.

[19] C. Terboven, D. an Mey, S. Sarholz, "Openmp on multicore architectures," Proceedings of the 3rd International workshop on OpenMP:A Practical Programming Model for the Multi-Core Era, pp. 54–64, June 2007.

[20] R. V. D. Pas, "Course notes," 2003, notes in Sun high performance education in Istanbul Technical University.

[21] A. Prabhakar, V. Getov, B. Chapman, "Performance comparisons of basic OpenMP constructs," Lecture Notes in Computer Science, vol. 2327, pp.293–296, 2006.