AI XX, 2 (2015) 7–13 ANNALES UMCS INFORMATICA DOI: 10.1515/umcsinfo-2015-0005

Transforming Source Code to Mathematical Relations for Performance Evaluation

Habib Izadkhah[1]*

[1] *Department of Computer Science,*
Faculty of Mathematical Sciences, University of Tabriz
Tabriz, Iran

Abstract – Assessing software quality attributes (such as performance, reliability, and security) from source code is of the utmost importance. The performance of a software system can be improved by its parallel and distributed execution. The aim of the parallel and distributed execution is to speed up by providing the maximum possible concurrency in executing the distributed segments. It is a well known fact that distributing a program cannot be always caused speeding up the execution of it; in some cases, this distribution can have negative effects on the running time of the program. Therefore, before distributing a source code, it should be specified whether its distribution could cause maximum possible concurrency or not. The existing methods and tools cannot achieve this aim from the source code. In this paper, we propose a mathematical relationship for object oriented programs that statically analyze the program by verifying the type of synchronous and asynchronous calls inside the source code. Then, we model the invocations of the software methods by Discrete Time Markov Chains (DTMC). Using the properties of DTMC and the proposed mathematical relationship, we will determine whether or not the source code can be distributed on homogeneous processors. The experimental results showed that we can specify whether the program is distributable or not, before deploying it on the distributed systems.

Keywords: *Distributed Software Systems, Source Code, Speed-up, Discrete Time Markov Chains*

(*Received: 18.05.2015; Revised: 21.07.2015; Published: 24.09.2015*)

1 Introduction

The need for high speed computation in large-scale scientific applications for analyzing complex scientific problems is very high, so that the common computers would not be able to satisfy it. Therefore, nowadays, using the distributed systems and processing power of numerous processors or cores to reach the favorable speed is known as a fact [1]. Yet, as a fact, creating a large scale distributed program is always more difficult than creating a non-distributed program with the same functionality, as the creation of a distributed system can change into a tedious and error-prone task.

Since the computational programs have many computations, so its execution requires more time. Therefore, if a program does not have the ability to distribute, there will be a lot of waste time. The most important time of a distributed program is invocation or communication time of their methods. These calls spend the most execution time. Certainly, by distributing a program, if two classes of it can be distributed on two different machines, the invocations between those classes will turn into the remote calls. As reference [2] specifies, in some cases, the program distribution can have negative effects on the running time of the program. When there are many calls between two methods, the network traffic increases and as a result, efficiency of the distributed program will be lower than the initial sequential program. So, regarding that constructing the distributed program from the source code is complex and time consuming, it is better to predict whether the source code is distributable or not, before distributing a program on the machines. None of the existing methods and tools can to achieve this goal from source code.

1.1 The Problem and the Claim

The overall problem addressed in this paper is to specify whether the source code has the potential for parallelization on homogeneous processors; i.e., in case of distribution, whether it brings the maximum concurrency compared to the sequential mode. We claim that it is possible to provide a solution to the mentioned problem by doing the following tasks:

(1) Modeling software's method invocations by Markov chains as (described in section III) as :
- Markov chains nodes represent methods and edges between nodes represent calls between methods,

*izadkhah@tabrizu.ac.ir

AI XV, 2 (2015) 7–18 DOI: 10.1515/umcsinfo-2015-0005

- weight of the edges in Markov chains, shows the number of calls between the methods.

(2) Determine the maximum potential of distributability of each method (described in section III)
(3) Determine the expected performance of the source code from obtained Markov chain (described in section III)
(4) ompute the speedup. Speedup is defined as the execution time of a sequential program divided by the execution time of a parallel program that computes the same result. In particular, Speedup $= T_s/T_p$ where T_s the sequential time and T_p is the expected performance.

1.2 The Paper Outline

The other sections of the paper are organized as follows: A literature review on the researches conducted by others is discussed in section II. In section III, we propose a mathematical relation of time estimation by which the potential for distribution of the source code can be specified. Case study is discussed in sections IV. At the end, section V deals with conclusions and future works.

2 Related Work and Background

The complicated computational applications cannot be executed in an acceptable time on the computation machine, so they should be divided into small tasks. We can use distributed or multiprocessors systems for executing of these tasks. Nowadays, most distributed and multiprocessor tools use scheduling methods for distribution. The aim of scheduling is execution of a program on several processors such that the time of execution of the whole program will be minimal, considering the time of tasks and communication time between the processors [3]. The scheduling methods can be divided into two groups; including those which can assurance the quality of service, and those which cannot. The former scheduling systems are preferred to the latter ones. CONDOR [4], SGE [5], PBS [6] and LSF [7] can be referred to as some of the most popular and widely used scheduling systems. These scheduling systems do not guarantee the service quality. These tools perform the scheduling only at the job level and not at applications'. Unlike the above systems, there are some which observe the service quality in scheduling. Such systems observe Job Characteristics, Planning in Scheduling, Rescheduling and Scheduling Optimization in their scheduling. AppleS [8], GrADS [9] and Nimrod/G [10] are among the most famous systems of this kind. Moreover, none of the aforementioned schedulers can predict whether an offered program has the potential to become parallelized, or whether speedup can be achieved in case of parallelization. Also, a tool called DAGC is presented to find the optimal architecture distribution [11]. DAGC uses clustering method for finding optimal architecture distribution. The tool uses a mathematical relation to measure the quality of the obtained clusters. The main problem in mathematical relation used in this tool and such tools is described above it does not have the ability to determine whether a program has the capability of being parallel or not. In the previous work [12], we proposed an analytical model for determining distributability of a specific method. However, our method in the previous work cannot determine overall distributability of a program; also, the effectiveness of each method is not considered in the distribution of it. In this research, we want to determine the overall distributability of a program using DTMC considering the effectiveness of each method.

2.1 Overview of Discrete Time Markov Chains

In this section, we discuss Discrete Time Markov Chains (DTMCs), which we use to model the source code's invocations [13]. A DTMC is described by its states and transition probabilities between the states; where we indicate the transition probabilities between the states as one-step transition probability matrix. The one-step transition probability is the probability that the process, when in state i at time n, will next transition to state j at time $n+1$. We write:

$$P_{ij}(n) = P(X_{n+1} = j \,|\, X_n = i)\,. \tag{1}$$

Note that all the elements in a row of P add up to 1 and each of the $P_{i,j}$'s lie in the range $[0, 1]$. For our purpose, we use absorbing DTMC. One DTMC is called absorbing if at least one state has no outgoing transition. Each DTMC with several final states can be converted into an absorbing DTMC. It is performed by adding a final state to DTMC. Next, a transition is drawn to the added absorbing state from all the final states available in DTMC. We can partition the transition probability matrix of an absorbing DTMC as:

$$P = \begin{bmatrix} 1 & 0 \\ C & Q \end{bmatrix}. \tag{2}$$

If the DTMC has n states with m absorbing states, Q would be a $(n-m)\times(n-m)$ sub-stochastic matrix (with at least one row sum < 1) describing the probabilities of transition only between transient states, 1 being a $m \times m$ identity matrix, 0 would be an $n\times(n\times m)$ matrix of zeros, and C would be an $(n-m)\times m$ matrix describing the probabilities of transition between transient states and absorbing state. The (i,j)-th entry of Q^k denotes the probability of arriving to state s_j after exactly k steps,

AI XV, 2 (2015) 7–13 DOI: 10.1515/umcsinfo-2015-0005

starting from state s_i. Hence the inverse matrix $(I-Q)^{-1}$ exists. This is called the fundamental matrix F:

$$F = (I-Q)^{-1} = I + Q + Q^2 + Q^3 + \cdots \sum_{l=0}^{\infty} Q^1 . \tag{3}$$

Let $X_{i,j}$ represent the number of visits to state j starting from the state i before process is absorbed. It can be shown that the expected number of visits to state j with starting from state i (i.e, $E[X_{i,j}]$), before entering an absorbing state is given by the (i,j)-th entry of the fundamental matrix F [14, 15]. So

$$E[X_{i,j}] = m_{i,j} , \tag{4}$$

$m_{i,j}$ is the (i,j)-th entry of the fundamental matrix F. The variance of the expected number of visits could also be computed using the fundamental matrix. Let $\sigma_{i,j}$ denote the variance of the number of visits to the state j starting from state i. Define $F_D = [md_{i,j}]$ such that:

$$md_{i,j} = \begin{cases} m_{i,j}\, if\, i = j \\ 0\, otherwise . \end{cases} \tag{5}$$

In other words, F_D represents a diagonal matrix with the diagonal entries the same as that of F. If we define $F_2 = [m^2_{i,j}]$, we have:

$$\sigma^2 = F(2F_D - I) - F_2 . \tag{6}$$

Hence:

$$Var[X_{i,j}] = \sigma^2_{i,j} . \tag{7}$$

3 Predicting Performance Of A Source Code

In this section we describe our approach for modeling a software system that method invocations are represented by an absorbing DTMC; such that DTMC states represent the software methods, and the transitions between states represent transfer of control from one method to another. We assume that the system consists of n methods, and has a single initial state denoted by 1, and a single absorbing or exit state denoted by n. Consider Fig. 1. Numbers on edges indicate the probability of movement from one method to another method. In this paper the probability to go from method x to method y is computed as [number of method call from x to y / total number of out method call of x (i.e. fan out)]. The method invocations of the source code are given by the one-step transition probability matrix P.

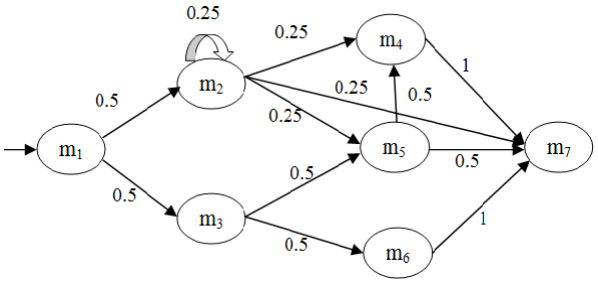

FIGURE 1. Modelling method invocations for a sample program with DTMC

Equation (8) shows the one-step transition probability matrix P for Figure 1.

$$P = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0.25 & 0.25 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \tag{8}$$

Let PD_i denotes the potential of distributability of method i that indicated by node i in the DTMC. During a single execution, the performance of the software, denoted by the random variable P is given by:

$$P = \prod_{i}^{n} PD_i^{X_{1,i}} . \tag{9}$$

where $X_{1,i}$ denotes the number of visits to the transient state i starting from the state 1. Therefore, the expected performance of a software system is as follows:

$$E[P] = E\left[\prod_{i}^{n} PD_i^{X_{1,i}}\right] = \prod_{i}^{n} E\left[PD_i^{X_{1,i}}\right] . \tag{10}$$

Thus to obtain the expected performance of the source code, we need to obtain $E\left[PD_i^{X_{1,i}}\right]$, which is the expected potential of distributability of method i for a single run of the software. Using the Taylor series expansion, $E\left[PD_i^{X_{1,i}}\right]$ in relation 10 can be written as relation 11.

$$E\left[PD_i^{X_{1,i}}\right] = PD_i^{E[X_{1,i}]} + \frac{1}{2}(PD_i^{E[X_{1,i}]})(logPD_i)^2 VarPD_i . \tag{11}$$

Let $E[X_{1,i}] = m_{1,i}$ and $Var[X_{i,j}] = \sigma^2_{i,j}$, relation (11) may be written as:

DOI: 10.1515/umcsinfo-2015-0005

(12)
$$E\Big[PD_i^{X_{1,i}}\Big] = PD_i^{m_{1,i}} + \frac{1}{2}(PD_i^{m_{1,i}})(logPD_i)^2\sigma^2 1, i\,.$$

$m_{1,i}$ is the expected number of visits to state i and $\sigma^2_{1,i}$ is the variance of the number of visits to state i. $m_{1,i}$ and $\sigma^2_{1,i}$ can be obtained from DTMC analysis. Relation (10) can thus be written as:

(13)
$$E[P] = \left[\prod_i^n (PD_i^{m_{1,i}} + \frac{1}{2}(PD_i^{m_{1,i}})(logPD_i)^2\sigma^2_{1,i})\right].$$

3.1 Computing Potential Of Distributability Of Method *i*

In this section, we are going to determine Potential of Distributability (PD) of each method to determine overall performance (i.e., P) of a program. For achieve this aim, we determine PD_i, to measure the values of different distributions for method i. Invocation (or call) between methods are two types of asynchronies and sequential. If by distributing a program, two methods of the program distribute in two different machines, calls between those methods will turn into asynchronies; and in sequential call, two methods of the program are placed on the same machine. Considering of communication time, our method considers two asynchronies and sequential mode for each call; to determine which mode (sequential or parallel) can reach a maximum speed up.

To estimate the speed-up, the execution time of all instructions should be estimated. The execution time of all instructions, except the nested calls, can be computed by the existing methods [16-17]. The existing methods cannot be applied easily to calculate the execution time of nested calls because the execution time of a caller method is depending on the fact that the calls inside it are carried. out in a sequential or asynchronous manner. For example, consider Listing 1. In the Listing 1, in the time t_1, the current method (caller method) will continue to work in a non-stop manner until reaching the use point of the results of a callee method. We call these points' synchronization points [18] and is shown by S. So, one method continues to work after calling a method from a remote locations (other distributed segments) and waits for a call response only when requires that response. As shown in Listing 1, the level of concurrency in executing the caller and the callee methods depends on the time interval between the call point and use point of the call results. The problem is the estimation of this interval time. As shown in Listing 1, there may be other calls between the call point and use point and the execution of these calls can be either synchronous or asynchronous.

LISTING 1. Several nested calls

```
Method m ( ) {
        Some statements       // t0
        Call R
        Some statements       // t1
        Use R                 // S
        Some statements       // t2
}
Method R ( ) {
        Some statements       // t3
        Call P
        Some statements       // t4
        Use P                 // S
        Some statements       // t5
}
Method P ( ) {
        Some statements       // t6
}
```

3.1.1 Estimated execution time for sequential mode

In Listing 1, considering methods m, R and P, if all of them executed sequentially (or synchronously), the estimated execution time will be calculated as follows:

(14) $$PD_m^{sequential} = t_0 + t_3 + t_6 + t_4 + t_5 + t_1 + t_2\,.$$

We can write above relation for Listing 1 in the recursive form and expand it for the nested call with any depth.

(15) $$PD_m^{sequential} = t_0 + PD_R^{sequential} + t_1 + t_2\,.$$

(16) $$PD_R^{sequential} = t_3 + PD_P^{sequential} + t_4 + t_5\,.$$

(17) $$PD_P^{sequential} = t_6\,.$$

Generally, for the sequential call, estimated execution time relation, is as relation:

(18) $$PD_m^{sequential} = \sum t_i + PD_R^{sequential}\,.$$

3.1.2 Estimated execution time for asynchronous mode

Now we calculate the estimated execution time when methods are executed parallel (or asynchronously). See again Listing 1. If methods m, R and P are executed

AI XV, 2 (2015) 7–23 DOI: 10.1515/umcsinfo-2015-0005

asynchronously, the estimated execution time will be calculated as follows:

$$PD_m^{asynch} = t_0 + t_1 + I_{init} + max(PD_R^{asynch} - t_1 + C_t + I_{init}, 0) + t_2 \,. \tag{19}$$

$$PD_R^{asynch} = t_3 + t_4 + T_{init} + max(PD_P^{asynch} - t_4 + C_t + I_{init}, 0) + t_5 \,. \tag{20}$$

$$PD_P^{asynch} = t_6 \,. \tag{21}$$

C_t is the communication time and I_{init} shows the preparation time for doing remote call. Generally, the estimated time relation for the parallel (or asynchronous) is calculated as follows:

$$PD_m^{asynch} = \sum t_i + \sum I_{init_i} + max(PD_R^{asynch} - t_i + C_t + I_{init}, 0) \,. \tag{22}$$

3.1.3 Determining the Potential of Distribution

Considering the relations (18) and (22), the general mathematical form of a PD relation is written as follows:

$$PD_m = \sum t_i + \sum a_i * PD_{Ii} + \sum (1 - a_i) \times (I_{init_i} + max((PD_{Ii} + C_t) - t_i + I_{init}, 0)) \,. \tag{23}$$

In the above relation, depending on the call to be synchronous or asynchronous, the value of a_i is considered as 1 and 0, respectively. The goal is to determine a_i, so that this minimizes PD_m . In the relation (23), the communication time is C_t and t_i is the estimated time between the callee point of I_i and the synchronization point of S_i (use point).

For example, to obtain PD for Listing 1, we need to combine the estimated times for the asynchronous (relation 22) and sequential execution (relations 15-17) as follows:

$$\begin{aligned} PD_m &= t_0 + a_1 * PD_R + t_1 + (1 - a_1) \times (I_{init} + max(PD_R - t_1 + C_t + I_{init}, 0)) + t_2 \,, \\ PD_R &= t_3 + a_2 * PD_P + t_4 + (1 - a_2) \times (I_{init} + max(PD_P - t_4 + C_t + I_{init}, 0)) + t_5 \,, \\ GTE_P &= t_6 \,. \end{aligned} \tag{24}$$

In relation 24, the aim is to determine a_1 and a_2 in a way to minimize PD_m, PD_R and PD_P.

Listing 2. A sample program code

```
Class A {
        Public void m ( ) {
                // some statements T1
                B b=new B();
                int r1 = b. m();
                print (r1); //S1
                C c=new C();
                int r2= c. n();
                D d=new D();
                int r3= d.p();
                // some statements T2
                if (r2==1) {...} //S2
                //some statements T3
                F f=new F();
                int r4= f.g();
                If(r1>r2 && r1>r3 && r1>r4)
                {...} // S3 and S4
                // some statements T4
        }
} // class

Class B extends A{
        static int m() {
        // some statements T5
        }
} // Class

Class C extends A{
        static int n() {
        // some statements T6
        }
} // Class

Class D {
        int p( ) {
                D d=new D();
                int r=d.p();
                Print (r); //S5
                F f=new F();
                int r1= f.g();
                If(r>r1) {...} // S6
        }
} //Class

Class F {
        // some statements T7
} //Class
```

Considering the program code in the Listring 2, $PD_{A.m}$ can be written as (25).

 DOI: 10.1515/umcsinfo-2015-0005

(25)

$$PD_{(A.m)} = T_1 + a_1 * PD_{(B.m)} + (1 - a_1) * T(S_1) + a_2 * PD_{(C.n)} + a_3 * PD_{(D.p)} + T_2 + (1 - a_2) * T(S_2) + T_3 + a_4 * PD_{(F.g)} + (1 - a_3) * T(S_3) + (1 - a_4) * T(S_4) + T_4\,,$$

$$PD_{(B.m)} = T5,\ PD_{(C.n)} = T6,\ PD_{(D.p)} = a_5 * PD_{(D.p)} + (1 - a_5) * T(S_5) + a_6 * PD_{(F.g)} + (1 - a_6) * T(S_6)\,,$$

$$PD_{(F.g)} = T_7\,,$$

$$T(S_1) = max(PD_{(B.m)} + 2t_{c_1}, 0)\,,$$

$$T(S_2) = max(T_2 + a_3 * PD_{(D.p)} + 2_{t_c}, 0)\,,$$

$$T(S_3) = max((PD_{(F.g)} + 2t_{c_4}) + T_3 + ((1 - a_2) * T(S_2) + T_2), 0)\,,$$

$$T(S_4) = max((PD_{(F.g)} + 2t_{c_1}, 0)\,,$$

$$T(S_5) = max((PD_{(D.p)} + 2t_{c_1}, 0)\,,$$

$$T(S_6) = max((PD_{(F.g)} + 2t_{c_1}, 0)\,.$$

Sequential Time (seconds)	Expected Distributed Time (seconds)	Speed-up
380	261	1.455

TABLE 1. Distributed execution times, sequential execution times and speed-up for Listing 2

The aim of PD relations in (25) is to determine a_1, a_2, a_3, a_4 and a_5 in a way to minimize $PD_{(A.m)}$, $PD_{(B.m)}$, $PD_{(C.n)}$, $PD_{(D.p)}$ and $PD_{(F.g)}$. We use the Dantzig's simplex algorithm [20] to determine the binary values of a_i (for synchronous call the value of a_i is considered as 1 and for asynchronous calls, the value of a_i is considered as 0). Simplex method is a popular algorithm for linear programming. Then, after determining PD for methods m, n, p and g, we make DTMC for the program of Listing 2 and then we compute the potential of distributability (using relation 24) for each method and then of course we will determine expected performance (relation 12). Also, the sequential execution time of the program is calculated as well. Finally, the speedup is calculated by dividing the sequential time to the expected performance. For relations (25) , the communication overhead is considered as 1 second and T_1, T_2, T_3, T_4 and T_5 (execution time of non-call statements) are considered as 40, 35, 45, 50 and 20 seconds. Table 1 shows the expected distributed potential (using relation 13), sequential, and speed-up execution times for Listing 2. Since speed-up is bigger than one, this indicates that the program is capable of parallel execution; i.e., the parallel execution of the program is faster than the sequential execution of the program.

4 Evaluation Result

In this section, we evaluate the performance of the proposed method. We want to determine when the speed-up achieved by our method is greater than one; the actual execution will speed up. For achieve this goal, we use jDistributor [2] tool. jDistributor is a tool for automatic distribution of the sequential program on the homogeneous distributed systems using the Java Symphony middleware [19]. The algorithm used in the jDistributor is a hierarchical clustering method and its goal is to find an appropriate clustering for distribution. We use the well-known travelling salesman problem (TSP) for evaluating of the proposed method. We compute $PD^{sequence}$ and PD^{asyn} from source code. We then predict from PD relation, the estimated time of the parallel and sequential execution for different graph nodes and then calculate speed-up by them. Afterwards, we distribute the TSP on the network including three computers by use of the jDistributor tool and then we calculate the parallel and sequential executions time. The results are shown in Table 2.

5 Conclusion

In this paper, we introduced a new approach to specify whether the source code is distributable or not, before the distribution. For achieve this goal, by considering asynchronous and sequential calls, a mathematical relationship was proposed to measure different distributions values from the same program code. Then, we model the software's method invocations by Discrete Time Markov Chains (DTMC). DTMC and its properties and proposed mathematical relationship can determine whether or not the source code distribution capabilities on homogeneous processors.

DOI: 10.1515/umcsinfo-2015-0005

Graph No.		Estimated Execution Time			Actual Execution Time (Using jDistributor tool [2])		
Nodes	Edges	Sequential Time (using relation 18)	Expected Distributed Time (using relations 13 and 23)	Speed-up	Sequential Time (millisecond)	Distributed Time (millisecond)	Speed-up
20	30	405	4375	0.092	589	7717	0.076
30	50	801	4932	0.162	1281	8310	0.154
60	100	2230	5401	0.412	3442	8503	0.404
80	180	7569	7220	1.048	13314	12809	1.039
100	310	19341	10002	1.933	21773	16731	1.301
130	420	35987	20075	1.792	43517	30722	1.416
170	686	59811	28676	2.085	82973	40362	2.055

TABLE 2. Comparison of estimated execution time using PD relation with its actual execution time

5.1 Future Work

We plan to extend and improve this work as follows: Our aim is to propose an algorithm, which attempts to improve the speed-up as much as possible in the distribution environments by reordering instructions at the compilation time. Therefore, it attempts will be made to increase distance between the caller points to its use point using the techniques known as instructions scheduling, for increase concurrent time of caller and callee methods as much as possible.

References

[1] J. AL-Jaroodi, N. Mahamad, H. Jiang, D. Swanson, "JOPI: a Java object passing interface", Concurrency Comput. Pract. Exp., Volume 17, pp. 775–795, 2005.

[2] S. Parsa, and V. Khalilpoor, "Automatic Distribution of Sequential Code Using JavaSymphony Middleware", SOFSEM 2006, LNCS 3831, pp. 440 – 450, 2006.

[3] L. S. Georgios, and D. K. Helen, "Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques", Simulation Modelling Practice and Theory, Volume 19, Issue 1, pp. 540-552, 2011.

[4] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience", Concurrency and Computation: Practice and Experience, Volume 17, No. 2-4, pp. 323-356, 2005.

[5] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid Cluster", Proceedings. First IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 35 – 36, 2001.

[6] B. Nitzberg, J. M. Schopf, J. P. Jones, "PBS Pro: Grid computing and scheduling attributes Grid resource management", pp. 183 – 190, 2004, Kluwer Academic Publishers Norwell, MA, USA.

[7] S. Zhou, J. Wang, X. Zheng, P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems", Software—Practice & Experience Volume 23 Issue 12, pp. 1305 – 1336, 1993.

[8] F. Berman, "Adaptive computing on the Grid using AppLeS, Parallel and Distributed Systems", IEEE Transactions on, Volume 14 , Issue 4 pp. 369 – 382, 2003.

[9] F. Berman, "New grid scheduling and rescheduling methods in the GrADS project", International Journal of Parallel Programming - Special issue: The next generation software program archive, Volume 33, Issue 2, pp. 209 – 229, 2005.

[10] Oriented Grid and Utility Computing (Wiley Series on Parallel and Distributed Computing), Editors Rajkumar Buyya and Kris Bubendorfer, ISBN-13: 978-0470287682.

[11] O. Bushehrian, "Automatic actor-based program partitioning", Journal of Zhejiang University-SCIENCE C (Computers & Electronics), 11(1), pp 45-55, 2010. [doi: 10.1631/jzus.C0910096]

[12] A. Isazadeh, J. Karimpour, I. Elgedawy, H. Izadkhah, "An Analytical Model for Source Code Distributability Verification", Springer Journal of Zhejiang University-SCIENCE C, Vol. 15, Issue 2, pp 126-138, 2014.

[13] A. Isazadeh, I. Elgedawy, J. Karimpour, H. Izadkhah, "An Analytical Security Model for Existing Software Systems", to appear in Applied Mathematics & Information Science, Vol. 8, Issue 2, pp 691-702, 2014.

[14] U. N. Bhat, "Elements of Applied Stochastic Processes", second ed. John Wiley & Sons, Inc, 1984.

[15] K. S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications", John Wiley and Sons, 2001.

[16] M. Schoeberl, "A time predictable Java processor", Proc. Conf. Design, Automation and Test in Europe, Germany, pp. 800–805, 2006

[17] C. A. Healy, M. Sjodin, D. B. Whalley, "Bounding loop iterations for timing analysis", Proc. IEEE Real-Time Technology and Applications Symp., pp. 12–21, 1998.

[18] R. Maani, S. Parsa, "An Algorithm to Improve Parallelism in Distributes Systems Using Asynchronous Calls", 7th Int. Conf. on Parallel Processing and Applied Math, p.312-317, 2007.

[19] T. Fahringer, A. Jugravu, "JavaSymphony: new directives to control and synchronize locality, parallelism, and load balancing for cluster and GRID-computing", Proc. Joint ACM Java Grande – ISCOPE 2002 Conf., Seattle, Washington.

[20] P. A. Jensen and J. F. Bard, "Operations Research Models and Methods", published by John Wiley and Sons, 2003.