Transforming Source Code to Mathematical Relations for Performance Evaluation

Habib Izadkhah

1Department 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

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

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:

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

Note that all the elements in a row of $P$ add up to 1 and each of the $P_{ij}$'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:

$$P = \begin{bmatrix} 1 & 0 \\ C & Q \end{bmatrix}.$$
starting from state $s_i$. Hence the inverse matrix $(I - Q)^{-1}$ exists. This is called the fundamental matrix $F$:

\[(3) \quad F = (I - Q)^{-1} = I + Q + Q^2 + Q^3 + \cdots \sum_{i=0}^{\infty} Q^i.\]

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

\[(4) \quad E[X_{i,j}] = m_{i,j},\]

$m_{i,j}$ 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 $F_D = [m^2_{i,j}]$ such that:

\[(5) \quad m^2_{i,j} = \begin{cases} m_{i,j} & \text{if } i = j \\ 0 & \text{otherwise}. \end{cases}\]

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:

\[(6) \quad \sigma^2 = F(2F_D - I) = F_2.\]

Hence:

\[(7) \quad \text{Var}[X_{i,j}] = \sigma^2_{i,j}.\]

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 $I$, 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](image)

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

\[(8) \quad P = \begin{bmatrix}
0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 \\
0 & 0.25 & 0.25 & 0.25 & 0.25 & 0 & 0 \\
0 & 0 & 0 & 0.5 & 0.5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

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:

\[(9) \quad P = \prod_{i=1}^{n} PD_i^{X_{1,i}}.\]

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:

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

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

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

Let $E[X_{1,i}] = m_{1,i}$ and $Var[X_{i,j}] = \sigma^2_{i,j}$, relation (11) may be written as:
(12) \[ E \left[ PD_{X_i}^{m_{i,i}} \right] = PD_{i}^{m_{i,i}} + \frac{1}{2} (PD_{i}^{m_{i,i}}) (\log PD_{i})^2 \sigma_{i,i}^2. \]

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

(13) \[ E[P] = \prod_{i=1}^{n} \left( PD_{i}^{m_{i,i}} + \frac{1}{2} (PD_{i}^{m_{i,i}}) (\log PD_{i})^2 \sigma_{i,i}^2 \right). \]

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 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

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

\[
P_{DA,m}^{asynch} = t_0 + t_1 + I_{init} + \max(PD_R^{asynch} - t_1 + C_t + I_{init}, 0) + t_2.
\]

\[
max(PD_R^{asynch} - t_1 + C_t + I_{init}, 0) + t_2.
\]

\[
P_{DR}^{asynch} = t_3 + t_4 + T\_init + \max(PD_P^{asynch} - t_4 + C_t + I_{init}, 0) + t_5.
\]

\[
P_{DP}^{asynch} = t_6.
\]

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

\[
P_{D_m}^{asynch} = \sum t_i + \sum I_{init} + \max(PD_R^{asynch} - t_i + C_t + I_{init}, 0).
\]

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:

\[
P_D = \sum t_i + \sum a_i * PD_{I_i} + \sum (1 - a_i) \times (I_{init} + \max((PD_{I_i} + C_t) - t_i + I_{init}, 0)).
\]

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 call 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:

\[
P_D = t_0 + a_1 * PD_R + t_1 + (1 - a_1) \times (I_{init} + \max(PD_R - t_1 + C_t + I_{init}, 0)) + t_2,
\]

\[
PD_R = t_3 + a_2 * PD_P + t_4 + (1 - a_2) \times (I_{init} + \max(PD_P - t_4 + C_t + I_{init}, 0)) + t_5,
\]

\[
GTE_P = t_6,
\]

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 Listing 2, \(PD_{A,m}\) can be written as (25).
\[
P D_{(A,m)} = T_1 + a_1 \cdot P D_{(B,m)} + (1 - a_1) \cdot T(S_1) + a_2 \cdot P D_{(C,n)} + a_3 \cdot P D_{(D,p)} + T_2 + (1 - a_2) \cdot T(S_2) + T_3 + a_4 \cdot P D_{(F,g)} + (1 - a_3) \cdot T(S_3) + (1 - a_4) \cdot T(S_4) + T_4,
\]

\[
P D_{(B,m)} = T_5, \quad P D_{(C,n)} = T_6, \quad P D_{(D,p)} = a_5 \cdot P D_{(D,p)} + (1 - a_5) \cdot T(S_5) + a_6 \cdot P D_{(F,g)} + (1 - a_6) \cdot T(S_6),
\]

\[
P D_{(F,g)} = T_7,
\]

\[
T(S_1) = \max(P D_{(B,m)} + 2t_{c_1}, 0),
\]

\[
T(S_2) = \max(T_2 + a_3 \cdot P D_{(D,p)} + 2t_{c_1}, 0),
\]

\[
T(S_3) = \max((P D_{(F,g)} + 2t_{c_1}) + T_3 + ((1 - a_2) \cdot T(S_2) + T_2), 0),
\]

\[
T(S_4) = \max((P D_{(F,g)} + 2t_{c_1}), 0),
\]

\[
T(S_5) = \max(P D_{(D,p)} + 2t_{c_1}, 0),
\]

\[
T(S_6) = \max((P D_{(F,g)} + 2t_{c_1}, 0).
\]

<table>
<thead>
<tr>
<th>Sequential Time (seconds)</th>
<th>Expected Distributed Time (seconds)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>380</td>
<td>261</td>
<td>1.455</td>
</tr>
</tbody>
</table>

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 \(P D_{(A,m)}, P D_{(B,m)}, P D_{(C,n)}, P D_{(D,p)}\) and \(P D_{(F,g)}\). We use the Dantzig’s simplex algorithm [20] to determine the binary values of \(a_1\) for synchronous call the value of \(a_1\) is considered as 1 and for asynchronous calls, the value of \(a_1\) is considered as 0. Simplex method is a popular algorithm for linear programming. Then, after determining \(P D\) 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 \(PD_{async}\) 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 execution 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 distribution 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.
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 increased concurrent time of caller and callee methods as much as possible.

References


