Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

An analytical model for source code distributability verification

Ayaz ISAZADEH[1], Jaber KARIMPOUR[1], Islam ELGEDAWY[2], Habib IZADKHAH[‡1]
([1]*Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran*)
([2]*Department of Computer Engineering, Middle East Technical University, Northern Cyprus Campus, Mersin 10, Turkey*)
E-mail: isazadeh@tabrizu.ac.ir; karimpour@tabrizu.ac.ir; elgedawy@metu.edu.tr; izadkhah@tabrizu.ac.ir

Received Mar. 18, 2013; Revision accepted July 22, 2013; Crosschecked Nov. 18, 2013

Abstract: One way to speed up the execution of sequential programs is to divide them into concurrent segments and execute such segments in a parallel manner over a distributed computing environment. We argue that the execution speedup primarily depends on the concurrency degree between the identified segments as well as communication overhead between the segments. To guarantee the best speedup, we have to obtain the maximum possible concurrency degree between the identified segments, taking communication overhead into consideration. Existing code distributor and multi-threading approaches do not fulfill such requirements; hence, they cannot provide expected distributability gains in advance. To overcome such limitations, we propose a novel approach for verifying the distributability of sequential object-oriented programs. The proposed approach enables users to see the maximum speedup gains before the actual distributability implementations, as it computes an objective function which is used to measure different distribution values from the same program, taking into consideration both remote and sequential calls. Experimental results showed that the proposed approach successfully determines the distributability of different real-life software applications compared with their real-life sequential and distributed implementations.

Key words: Code distributability, Synchronous calls, Asynchronous calls, Distributed software systems, Source code
doi:10.1631/jzus.C1300066 **Document code:** A **CLC number:** TP31

1 Introduction

Distributed systems and multi-processor machines are used nowadays to speed up execution of existing software programs (Al-Jaroodi *et al*., 2005). However, transforming sequential programs into distributed programs is a very challenging task, as we need to divide the software programs into concurrent segments (or clusters) that could work in a parallel manner (i.e., the clustering problem). Such a clustering problem is known as an NP-complete problem (Zhang *et al*., 2010). Moreover, we have to take into consideration the network latency between remote segments (clusters). For example, Fig. 1 shows a distributed architecture with three clusters, each of which has several classes. Each cluster should be deployed on different workstations.

The calls between two classes within the same distributed segment (cluster) will be accomplished via synchronous calls (e.g., C_1, C_2, C_5–C_9 calls), as through the synchronous call two classes are executed in sequential manner on the machine. On the other hand, the calls between two classes within the different distributed segments (different clusters) will be accomplished via an asynchronous call (e.g., C_3, C_4, C_{10}, C_{11} calls), as we do not want to block the caller classes for a long time due to network latency. Hence, communication overhead is crucial in determining the distributability of sequential programs.

Of course, doing such a challenging process manually is very tedious and error-prone; hence, we argue that such a process must be done automatically to avoid potential errors and to shorten the time for transforming sequential programs into distributed multi-threaded programs. To be able to automatically identify such clusters, we argue that an object-oriented approach (Deb *et al*., 2006) should be used,

‡ Corresponding author

as the code of a program can be easily encapsulated into objects, which can be categorized into individual clusters using clustering methods such as the ones discussed in Bushehrian (2010), where each cluster could be deployed at a number of distinct remote locations and communicate by exchanging messages (e.g., DCOM objects) over the network.

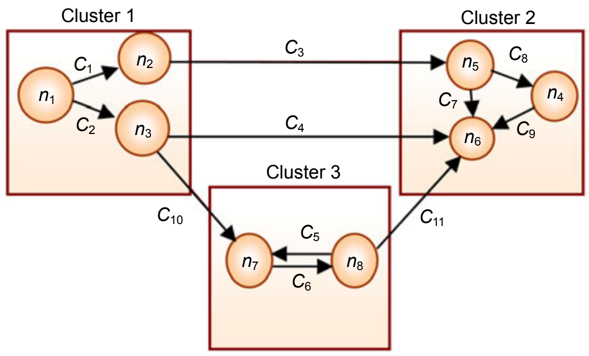


Fig. 1 A sample distributed architecture

Application clustering in an object-oriented environment is aimed to break up the functionality of an application into the distinct set of objects that can operate concurrently. We argue that the execution speedup achieved depends mainly on the concurrency degree between identified clusters. Hence, to achieve the best speedup, we have to obtain the maximum possible concurrency degree between identified clusters. Unfortunately, existing code distribution tools (Zhou *et al.*, 1993; Gentzsch, 2001; Berman *et al.*, 2003; Nitzberg *et al.*, 2004; Berman *et al.*, 2005; Thain *et al.*, 2005; Buyya and Abramson, 2009) do not check for the maximum possible concurrency degree between identified clusters. To overcome such limitations, in this paper we propose a novel approach for verifying the distributability of sequential object-oriented programs. We use a mathematical relationship, indicated by general time estimation (GTE), to estimate the distribution of a program based on the concurrency level obtained from asynchronous remote calls, considering the communication delays such as network latency. Then we compute the distribution speedup and search for the distribution values that maximize the achieved speedup. After determining the speedup, we can determine whether a source code is distributable or not. To compute the GTE relationship, we traverse the source code and determine possible call destinations. This is done by generating a call dependency graph (CDG), whose vertices represent all the system classes and whose edges show the dependencies between these classes. Unlike existing approaches for generating CDG, the proposed approach considers the impact of implicit calls. Note that traditional multi-threading approaches do not necessarily consider the communication overhead between entities when determining thread concurrency, as they focus mainly on system functionality. Hence, they cannot predict the final perceived performance of the system. On the other hand, the proposed approach considers both system functionality and perceived performance. Experimental results showed that the proposed approach successfully determines the distributability of different real-life software applications, compared with their real-life sequential and distributed implementations.

The main advantage of our approach is that we can determine the possibility of distributability of the source code before its parallelization, to see if it is indeed distributable, because the code may not be distributable in the first place. If it is determined that the distributor is, in this case, multi-threading, parallel programming or clustering methods can be used in the next step for distribution. Otherwise, if the program cannot be distributed, knowing this ahead of time will save a lot of time. Therefore, the overhead for determining the distributability would be worthwhile.

2 Related works

Nowadays, most distributed and multi-processor systems use task graph scheduling methods for distribution (deployment). The objective in these methods is task scheduling on distributed systems to minimize the completion time of the last task. The scheduling techniques are typically categorized into two classes, homogeneous and heterogeneous. In the homogeneous type, the processing power of all processors is considered the same. In the heterogeneous type, processors have different processing powers. CONDOR (Thain *et al.*, 2005), SGE (Gentzsch, 2001), PBS (Nitzberg *et al.*, 2004), LSF (Zhou *et al.*, 1993), AppleS (Berman *et al.*, 2003), GrADS (Berman *et al.*, 2005), and Nimrod/G (Buyya and Abramson, 2009) are the most famous scheduling systems. None of these schedulers, however, can specify whether an offered application program has the potential of becoming parallelized, or whether speedup can be achieved in case of parallelization.

To specify whether the source code is distributable or not, we require an analysis tool to determine the actual destinations of method calls from the source code. For this purpose, we generate the call dependency graph (CDG). Currently, to create the CDG from an object-oriented source code, most tools will use: (1) for C++ programs, Acacia (Chen *et al.*, 1997), Columbus (Ferenc *et al.*, 2004), and Reveal (Matzko *et al.*, 2002); (2) for Java programs, Chava (Korn *et al.*, 1999); (3) for most object oriented programs, NDepend (www.ndepend.com), Understand (www.scitools.com), Bauhaus (Raza *et al.*, 2006), and Imagix-4D (Koskinen and Lehmonen, 2010). The main problem of these algorithms is that the implicit calls are not considered in design. Hence, these algorithms could not construct the CDG precisely when the source code includes implicit calls. In fact, a dependency graph that has been produced is pessimistic and has a number of edges (some of them are not needed) that consequently have a reverse effect on the concurrency results due to the many edges. Fig. 2 shows the pseudo code which does not include an implicit call, while Fig. 3 shows an implicit call in the pseudo code.

In Fig. 2, the declared type for variable *a* is class *A* and *a* instantiated of class *A*. Thus, call destination *a*.method1() is considered class *A*. In Fig. 3, the declared type of *a* is class *A* but *a* instantiated of class *B*. Thus, call destination *a*.method1() should be considered class *B*, not class *A*. In such cases, existing algorithms consider both classes as call destinations. In fact, a dependence graph is produced pessimistically. This kind of call is called an implicit call. Fig. 4 shows a CDG generated for Fig. 3 by Chava, NDepend, Understand, and Bauhaus algorithms. This graph is constructed pessimistically. An appropriate CDG for Fig. 3 is as shown in Fig. 5. Algorithms used for constructing a CDG act pessimistically. These algorithms construct the call graph conservatively and do not eliminate any probable call from the graph. As a result, the call graph obtained will have many edges and thus a negative impact on concurrency.

The main contribution of our method is two-fold. First, it provides a new mathematical relationship for estimating sequential and distributed execution times, extracted automatically from the object oriented source code, to specify whether a program is suitable for parallelization through studying the types of synchronous and asynchronous method calls inside

```
Class A {
          Public void method1() { print ("This is A"); }
           }
Class B extends A {
          Public void method1() { print ("This is B"); }
          }
Class C extends A {
          Public void method1() { print ("This is C"); }
          }
Class Main {
Public void method2() {
    A a;
    a=new A();
    a.method1();
                              }
}
```

Fig. 2 Sample pseudocode without an implicit call

```
Class A { … }
Class B extends A { … }
Class C {
          public void  m1() {
                    A a;
                    a=new B();
                    a.method1();
                    }
          } // Class c
Class main_Class {
          public static void main() {
                    A a=new A();
                    a.m();
                    C c=new C();
                    c.m1();
                    } // main
          } // main_Class
```

Fig. 3 Sample pseudocode including an implicit call

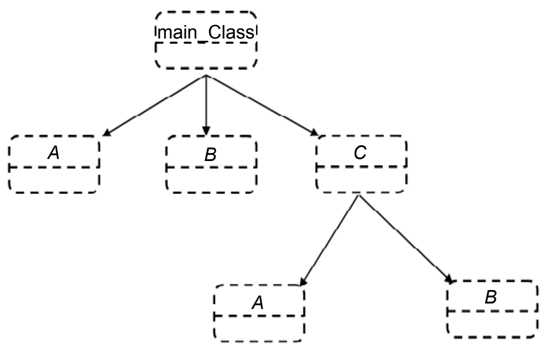


Fig. 4 The call dependency graph generated for Fig. 3 by Chava, NDepend, Understand, and Bauhaus algorithms

the source code. Second, it presents a new algorithm to determine the actual destinations of calls from the source code.

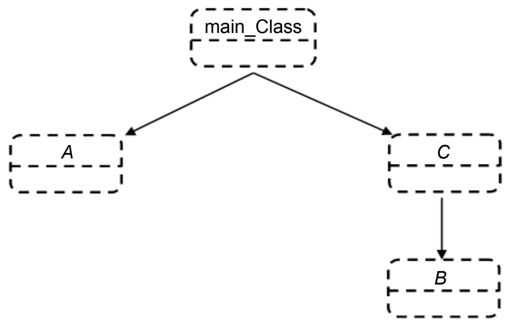


Fig. 5 An appropriate call dependency graph for Fig. 3

3 Source code analysis and CDG generation

In this section, we propose an algorithm to determine the actual destination of calls from source code, taking into consideration the relationship types between classes. In general, there are two basic criteria in the relationship between classes:

1. Type of interaction: determine the ways in which the two classes communicate with each other.

Aggregation: aggregation is a relationship between two classes, best described as a ‘has-a’ and ‘whole/part’ relationship.

Class-method: in this case, class D is a parameter of method m_c of class C.

Method-method: in this case, method m_d of a class D directly invokes a method m_c of a class C, or a method m_d receives via a parameter a pointer to m_c, thereby invoking m_c indirectly.

2. Type of relation: determine the ways in which the two classes are related to each other.

Inheritance: in this case, class D inherits attributes and behavior of class C, or vice versa.

Friendship: in this case, a friend class has access to the private and protected members of the class.

Other relationships between classes C and D are interface and abstract.

Determining the actual destination of calls has a great effect on the quality of the proposed method since these calls are used to specify whether the source code is distributable or not. The actual destinations of the calls are determined at the run-time, which makes them more difficult to recognize, as we have to predict the behavior of the system. We refer to these calls as implicit calls. To precisely determine the destination of a call, we have to consider both explicit and implicit calls.

Definition 1 (Destination of a method call) The destination of a method call is identified as follows:

1. If within a class, the class is defined as the class-attribute, then the class will be the calling destination of the main class.

2. If within a class, the class is defined as the class-method, then the class will be the calling destination of the main class.

3. If a call, such as o, has a declared class type C, the possible destination run-time of o, i.e., Destination(o), includes C and all sub-classes of C.

4. If a call of o has a declared interface I, the possible destination run-time of o, i.e., Destination(o), includes: (1) the set of all classes that implement I or a sub-interface of I, which we call implements(I); (2) all subclasses of implements(I).

The main aim is to precisely identify a set of reaching variables to o in each call, like $o.m()$. This set is called Receiving-types(o). The proposed algorithm uses a graph to perform this action. For example, we say type A reaches variable o if once at least there would be one path in the program run to be started by an object of type A (e.g., as v=new A()), and then this chain of assignment would be

$$x_1=v,\ x_2=x_1,\ \ldots,\ x_n=x_{n-1},\ o=x_n. \qquad (1)$$

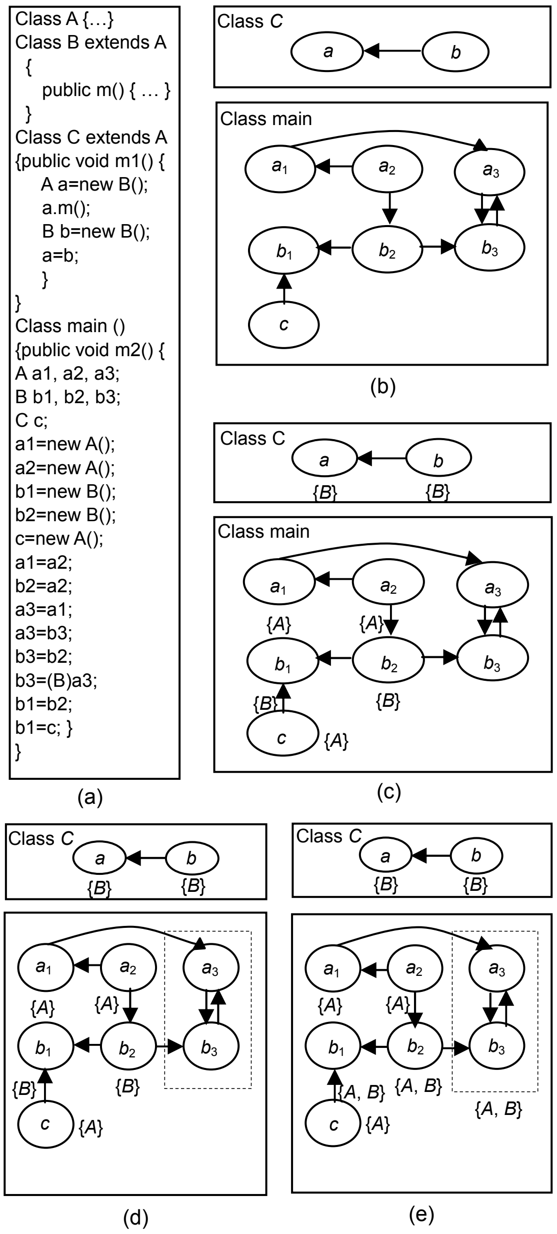
Given a program P, the destination of a call is determined using Algorithm 1, considering the explicit and implicit method analysis (EIMA). We denote this algorithm as the EIMA algorithm.

In Fig. 6a, we provide the important parts of an example program. Figs. 6b–6e show steps 1–4 in Algorithm 1 for code in Fig. 6a. Fig. 6b shows construction of the graph based on assignments in code. Fig. 6c shows the initial assigned values, while Fig. 6d shows the removal of cycles from the graph and Fig. 6e shows the propagation of the types. It is obvious that nodes a_3 and b_3, which are in a same cycle, are converted to a united node before propagation. After calculating the Receiving-types(o) set for each call using Algorithm 1, the actual destination of each call is determined using Eq. (2).

Algorithm 1 Determining the destination of a call

```
Step 1: Graph construction, in which nodes show variables and
        each edge as $a \rightarrow b$ shows an assignment as $b=a$.
  Step 1.1: Nodes are created as follows:
    1. For each field $f$ (where $f$ has a reference to a class) in
       class $C$ into namespace NS, create a new node labeled
       NS.$C$.$f$.
    // This condition occurs when a class is defined as
    // static class or aggregation occurs
    2. For each method $m$ in class $C$ into namespace NS,
       create a new node labeled NS.$C$.$m$.
  Step 1.2: Edges are added as follows:
       For each statement of form lvariable=rvariable or
       lvariable=($C$)rvariable, where lvariable and rvariable
       must be an ordinary, field or array reference, add a
       directed edge from the rvariable node to the lvariable
       node.
Step 2: Initialize the graph, in which all assignments would be
        searched as lvariable=new type and type would be
        placed as the initial value in Receiving-types(lvariable)
        set.
Step 3: Remove all cycles from the graph and generate a new
        directed graph without cycles. To remove cycles, the
        nodes are those that are located in a cycle to be con-
        verted into a node. Receiving-types(lvariable) of this
        node would be obtained from the union of nodes.
Step 4: Compute the Receiving-types($o$) set for each call
        through propagation of types in the graph.
Step 5: After the above steps, the actual destination of each
        call, EIMA($o$), would be obtained by the following
        relationship:
```

$$\text{EIMA}(o)=\text{Destination}(o)\cap\text{Receiving-types}(o). \quad (2)$$



```
Class A {…}
Class B extends A
 {
    public m() { … }
 }
Class C extends A
{public void m1() {
    A a=new B();
    a.m();
    B b=new B();
    a=b;
    }
}
Class main ()
{public void m2() {
A a1, a2, a3;
B b1, b2, b3;
C c;
a1=new A();
a2=new A();
b1=new B();
b2=new B();
c=new A();
a1=a2;
b2=a2;
a3=a1;
a3=b3;
b3=b2;
b3=(B)a3;
b1=b2;
b1=c; }
}
```

Fig. 6 Computing the Receiving-types(o) set for each call
(a) Important parts of an example program; (b) Graph construction based on assignments in code; (c) Initial assigned values; (d) Removal of cycles from the graph; (e) Type propagation in the graph

To generate our proposed GTE relationship, we analyze the program using Algorithm 2 and then extract the necessary facts from it and save them within a file named intermediate code. The intermediate code shows clear concepts and facts of methods in the program and does not include unnecessary details, and in fact, shows a summarized model of the program that we need to analyze. The intermediate code generated should include the required structure to construct our relationship. Each method of the program is specified through ordered elements in the intermediate code. These elements can include one of these three types: call statements, non-call statements, and synchronization points, which are shown by Call, Some_computation, and Use, respectively. Call$_i$ indicates an indicator to the other method which is called through the current method. The actual destinations of the calls are determined using Algorithm 1. Use$_i$ indicates the first use point of Call$_i$ in the method. In this study, we use the Def-Use chain method (which is used in a super compilers technique), to determine Use$_i$ related to Call$_i$. A Def-Use chain (Zima and Chapman, 1991) can be constructed to find the first locations within the program code where the values affected by a remote method call are required.

Some_computation$_i$ indicates the estimated execution time of sequence of the instructions in the method between Call$_i$ and Use$_i$ without considering the call statement. We have used the WCET method (Schoeberl and Pedersen, 2006) to estimate execution time of instructions and loops. Algorithm 2 shows how to transform the source code into the intermediate code. For example, if the code in Fig. 7 is given as the input to Algorithm 2, then the intermediate code generated will be as shown in Fig. 8.

Algorithm 2 Transforming source code to intermediate code

```
For each package in source code
  For each class into a package
    For each method into each class, do the following:
      1. Write ("Method" + className + methodName).
      2. For each call, determine the actual destination of
         the call using Algorithm 1 and write ("Call" +
         destination className + destination method-
         Name).
      3. Determine the execution time of instructions us-
         ing the WCET method (Schoeberl and Pedersen,
         2006) as the following:
      - the execution time for the total instruction from
          the start point of the method to the first call,
      - the execution time between the call and the first
          point of its usage, and
      - the execution time between the usage point up to
          the next call and/or end of the method.
      4. For each call, determine where a call is used using
         the Def-Use algorithm (Zima and Chapman,
         1991) and write ("Use" + destination className +
         destination methodName).
    End
  End
End
```

```
package NS1
{
  class B extends ClassFormatError, ThreadDeath
  {  A a;
     C c;
     public M( )
       {  a=new A( ); c=new C( );
          Some_computation;
          y=a.m();
          Some_computation; z=c.n(); print(z);
          Some_computation;
          While (y!=0) { … }
       }
  }
}
```

Fig. 7 Sample source code

```
Method B.M
  Begin method
    Some_computation
    Call A.m
    Some_computation
    Call C.n
    Use C.n
    Some_computation
    Use A.m
EndMethod B.M
```

Fig. 8 The intermediate code generated for Fig. 7

4 Source code distributability verification

The transformation of synchronous local calls into asynchronous remote calls can have negative effects on the program execution speed (Parsa and Khalilpoor, 2006), as when there are many calls between two entities (e.g., classes), the network traffic increases and as a result, the efficiency of the distributed system decreases due to communication overhead. Therefore, before starting the process of source code distribution, we should determine whether its distribution could cause speedup or not. To achieve this goal, we propose general time estimation (GTE) relationship to measure the values of different distributions of source code from its intermediate code. Two modes are considered for each call, asynchronous and sequential. Concurrency is achieved through asynchronous calls between the distributed segments. To estimate the concurrency level generated in a program by a distribution, the execution time of the program should be estimated with respect to the way in which it is distributed. The execution time of all instructions, except the nested calls, can be computed using the available methods proposed by, for example, Healy *et al.* (1998) and Schoeberl (2006). The available methods cannot be applied easily to calculate the execution time of the nested calls because the execution time of a caller method is dependent on the fact that a caller method is executed in a synchronous or asynchronous manner with a callee method. For example, in Fig. 9, at time t_1, the current method (caller method) will continue to work in a non-stop manner until the use point of the results of a callee method is reached. This point is shown by *S*. As shown in Fig. 9, the level of concurrency in executing the caller and callee methods depends on the time interval between the call point and the use point. The difficulty lies in the estimation of this interval time. As shown in Fig. 10, there may be other calls within the distance between the call point and use point, and the execution of these calls can be either synchronous or asynchronous. The distributed segments connect to each other asynchronously; i.e., one segment continues to work after calling a method from a remote location (other distributed segments) and waits for a call response only when it requires that response. We call these points 'synchronization points' (Maani and Parsa, 2007).

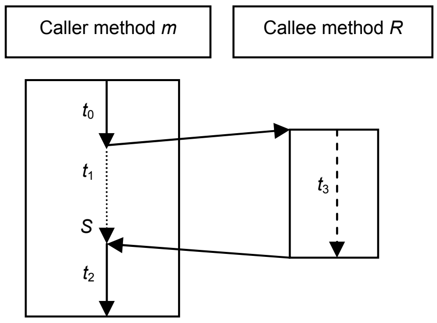


Fig. 9 Calls between methods

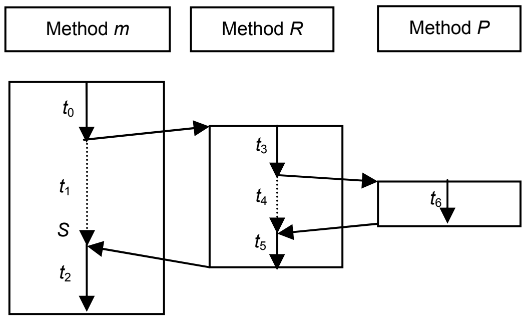


Fig. 10 Number of nested calls

4.1 Estimated execution time for sequential execution

In sequential execution, all methods are executed on the same processor. P_i represents the processing power of processor i in terms of the number of cycles per byte, given as one of the inputs. In Fig. 9, if method m and method R are executed serially (or synchronously), the estimated execution time will be as shown in Eq. (3). The time t_i indicates the program instruction times, estimated by the WCET algorithm (Schoeberl and Pedersen, 2006). WCET does not consider a specific processor processing power in estimating the execution time of instructions. Therefore, to calculate the real execution time of instructions, it should be divided by the estimated execution time (i.e., t_i) to processing power of processor i in terms of the number of cycles per byte (i.e., P_i).

$$\mathrm{GTE}_m^{\mathrm{seq}}=\frac{t_0}{P_i}+\frac{t_3}{P_i}+\frac{t_1}{P_i}+\frac{t_2}{P_i}. \tag{3}$$

Also, Eq. (3) can be written in recursive form as the following:

$$\begin{cases}\mathrm{GTE}_m^{\mathrm{seq}}=\dfrac{t_0}{P_i}+\mathrm{GTE}_R^{\mathrm{seq}}+\dfrac{t_1}{P_i}+\dfrac{t_2}{P_i},\\ \mathrm{GTE}_R^{\mathrm{seq}}=\dfrac{t_3}{P_i}.\end{cases} \tag{4}$$

Since the aim is to compare sequential time to parallel time, in Eq. (4), P_i is related to the most powerful processor for heterogeneous processors and to a power of one for homogeneous processors. Notice that, in Fig. 10 the depth of the nested calls is 2. The estimated time of the sequential execution for Fig. 10 will be as follows:

$$\begin{aligned}\mathrm{GTE}_m^{\mathrm{seq}}&=\frac{t_0}{P_i}+\frac{t_3}{P_i}+\frac{t_6}{P_i}+\frac{t_4}{P_i}+\frac{t_5}{P_i}+\frac{t_1}{P_i}+\frac{t_2}{P_i}\\&=\frac{1}{P_i}(t_0+t_3+t_6+t_4+t_5+t_1+t_2).\end{aligned} \tag{5}$$

We can rewrite the above relationship for Fig. 10 in recursive form (like Eq. (6)) and expand it for the nested call with any depth.

$$\begin{cases}\mathrm{GTE}_m^{\mathrm{seq}}=\dfrac{t_0}{P_i}+\mathrm{GTE}_R^{\mathrm{seq}}+\dfrac{t_1}{P_i}+\dfrac{t_2}{P_i},\\ \mathrm{GTE}_R^{\mathrm{seq}}=\dfrac{t_3}{P_i}+\mathrm{GTE}_P^{\mathrm{seq}}+\dfrac{t_4}{P_i}+\dfrac{t_5}{P_i},\\ \mathrm{GTE}_P^{\mathrm{seq}}=\dfrac{t_6}{P_i}.\end{cases} \tag{6}$$

Generally, for the sequential (or synchronous) call, the estimated execution time relationship for the cases in which the processors power is specified is

$$\mathrm{GTE}_m^{\mathrm{seq}}=\sum\frac{t_i}{P_i}+\mathrm{GTE}_R^{\mathrm{seq}}. \tag{7}$$

4.2 Estimated execution time for asynchronous mode

In this section, we will calculate the estimated execution time when methods are executed in parallel (asynchronously) and have specified the number and the power of the processors. See again Fig. 9. If method m and method R are executed asynchronously, the program estimation time is expressed as

$$\begin{aligned}\mathrm{GTE}_m^{\mathrm{asynch}} &= \frac{t_0}{P_i} + I_{\mathrm{init}} + \frac{t_1}{P_i} \\ &\quad + \max\left(\frac{t_3}{P_j} - \frac{t_1}{P_i} + C_t + I_{\mathrm{init}}, 0\right) + \frac{t_2}{P_i}, \\ &\quad i \neq j,\ i, j = 1, 2, \ldots, \text{number of processors,}\end{aligned} \tag{8}$$

where P_i represents the processing power of processor i, C_t is the communication time, and I_{init} indicates program preparation time for remote calling. The values of C_t's could be extracted from a latency matrix between processors and given as one of the inputs. Since the aim is parallel execution, the processor of method R should not be the same as the processor of method m. The aim is to find the processors that minimize $\mathrm{GTE}_m^{\mathrm{asynch}}$.

See again Fig. 10 in which the depth of the nested calls is 2. We can write the estimated execution time in recursive form for parallel execution:

$$\begin{cases} \mathrm{GTE}_m^{\mathrm{asynch}} = \dfrac{t_0}{P_i} + I_{\mathrm{init}} + \dfrac{t_1}{P_i} \\ \qquad + \max\left(\mathrm{GTE}_R^{\mathrm{asynch}} - \dfrac{t_1}{P_i} + C_t + I_{\mathrm{init}}, 0\right) + \dfrac{t_2}{P_i}, \\ \mathrm{GTE}_R^{\mathrm{asynch}} = \dfrac{t_3}{P_j} + I_{\mathrm{init}} + \dfrac{t_4}{P_j} \\ \qquad + \max\left(\mathrm{GTE}_P^{\mathrm{asynch}} - \dfrac{t_4}{P_j} + C_t + I_{\mathrm{init}}, 0\right) + \dfrac{t_5}{P_j}, \\ \mathrm{GTE}_P^{\mathrm{asynch}} = \dfrac{t_6}{P_k}. \end{cases} \tag{9}$$

The aim is to find the processors that can minimize $\mathrm{GTE}_m^{\mathrm{asynch}}$. In general, when the number and power of processors are known, the estimated time relationships for the parallel (or asynchronous) execution are

$$\begin{aligned}\mathrm{GTE}_m^{\mathrm{asynch}} &= \sum \frac{t_i}{P_i} + \sum I_{\mathrm{init}_i} + \max(\mathrm{GTE}_R^{\mathrm{asynch}} \\ &\quad - t_i / P_i + C_t + I_{\mathrm{init}}, 0).\end{aligned} \tag{10}$$

4.3 General time estimation relation

For Fig. 9, the following relationship will be obtained if the estimated times for the asynchronous and sequential execution are combined:

$$\begin{cases} \mathrm{GTE}_m = \dfrac{t_0}{P_i} + a_1 \cdot \mathrm{GTE}_R + \dfrac{t_1}{P_i} + (1 - a_1)\Bigg[I_{\mathrm{init}} \\ \qquad + \max\left(\mathrm{GTE}_R - \dfrac{t_1}{P_i} + C_t + I_{\mathrm{init}}, 0\right)\Bigg] + \dfrac{t_2}{P_i}, \\ \mathrm{GTE}_R = \dfrac{t_3}{P_j}. \end{cases} \tag{11}$$

Also, for Fig. 10, Eq. (12) will be obtained if the estimated times for the asynchronous and sequential executions are combined:

$$\begin{cases} \mathrm{GTE}_m = \dfrac{t_0}{P_i} + a_1 \cdot \mathrm{GTE}_R + \dfrac{t_1}{P_i} + (1 - a_1)\Bigg[I_{\mathrm{init}} \\ \qquad + \max\left(\mathrm{GTE}_R - \dfrac{t_1}{P_i} + C_t + I_{\mathrm{init}}, 0\right)\Bigg] + \dfrac{t_2}{P_i}, \\ \mathrm{GTE}_R = \dfrac{t_3}{P_j} + a_2 \cdot \mathrm{GTE}_P + \dfrac{t_4}{P_j} + (1 - a_2)\Bigg[I_{\mathrm{init}} \\ \qquad + \max\left(\mathrm{GTE}_P - \dfrac{t_4}{P_j} + C_t + I_{\mathrm{init}}, 0\right)\Bigg] + \dfrac{t_5}{P_j}, \\ \mathrm{GTE}_P = \dfrac{t_6}{P_k}. \end{cases} \tag{12}$$

The purpose of this combination is that there is a need to consider the synchronous and asynchronous modes for each call, to determine which one is causing the speedup. Considering Eqs. (7) and (10), the general mathematical formula of a GTE relationship is written as

$$\begin{aligned}\mathrm{GTE}_m &= \sum \frac{t_i}{P_i} + \sum a_i \cdot \mathrm{GTE}_{I_i} + \sum (1 - a_i)\Bigg[I_{\mathrm{init}_i} \\ &\quad + \max\left((\mathrm{GTE}_{I_i} + C_t) - \frac{t_i}{P_i} + I_{\mathrm{init}}, 0\right)\Bigg].\end{aligned} \tag{13}$$

Also, if the user cannot determine the power or the number of processors, we can delete P_i from the above relationships and therefore Eq. (13) can be written as follows:

$$\begin{aligned}\mathrm{GTE}_m = &\sum t_i + \sum a_i \cdot \mathrm{GTE}_{I_i} + \sum (1-a_i)\big[I_{\mathrm{init}_i} \\ &+ \max((\mathrm{GTE}_{I_i} + C_t) - t_i + I_{\mathrm{init}}, 0)\big].\end{aligned} \tag{14}$$

In the above relationship, depending on the call to be synchronous or asynchronous, the value of a_i is considered as 1 or 0, respectively. The aim is to determine a_i and P_i, to minimize GTE_m. In Eqs. (13) and (14), the communication time from performing the remote and asynchronous calls 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). However, there may be cycles in the CDG, resulting from direct or indirect recursive calls. Assuming that I_i is an invocation to a method in the cycle (and I_i itself is not in the cycle) and the estimated number of recursions is n_i, then the GTE of I_i should be multiplied by n_i and the back edge of the recursion should be removed from the call graph. In this case, Eq. (13) will be modified as follows:

$$\begin{aligned}\mathrm{GTE}_m = &\sum \frac{t_i}{P_i} + \sum a_i n_i \cdot \mathrm{GTE}_{I_i} + \sum (1-a_i)\left[I_{\mathrm{init}_i}\right. \\ &\left.+ \max\left((\mathrm{GTE}_{I_i} + C_t) - \frac{t_i}{P_i}, 0\right)\right].\end{aligned} \tag{15}$$

An invocation I_i or a synchronization point S_i may be located within a loop statement. Therefore, to consider the impact of loop iterations on time estimation, Eq. (13) may be modified as follows:

$$\begin{aligned}\mathrm{GTE}_m = &\sum \frac{t_i}{P_i} + \sum a_i n_i m_i \cdot \mathrm{GTE}_{I_i} + \sum (1-a_i) \\ &\cdot \left[I_{\mathrm{init}_i} + \max\left((\mathrm{GTE}_{I_i} + C_t) - \frac{t_i}{P_i}, 0\right)\right].\end{aligned} \tag{16}$$

In Fig. 11, as calls I_1–I_5 are synchronous or asynchronous, S_1–S_4 are synchronization points (use point) for the values returned from calls I_1–I_5, and T_1–T_5 are the execution times of non-call statements. In Fig. 12, a_1–a_5 are considered as 0 or 1. The above relationship is aimed to determine a_1–a_5 in a way to minimize GTE (A.main). We use the simplex optimization method in operation research to determine the binary values of a_i. After determining a_i and specifying the estimated time of parallel execution, the sequential execution time of the program is calculated as well. Finally, the speedup is calculated by dividing the sequential time into parallel estimation time. If this value is larger than one, the parallel execution of the program is faster than the sequential execution. In fact, the source code can be paralleled. For the relations in Fig. 11, the times of calls I_1–I_5 are 60, 60, 30, 20, and 30 s respectively, and T_1–T_5 are 30, 35, 32, 50, and 43 s respectively. Also, the communication overhead in a remote call is considered random communication costs. These communication costs are distributed uniformly within (1, 5) s. Table 1 shows the distributed, sequential, and speedup execution times.

```
Class A {
 void main(string[] arg) {
    int r1, r2, r3;
    B b=new B(); C c=new C(); D d=new D();
    r1=b.m();                   // I1
    r2=c.n();                   // I2
    for(i=0; i<n; i++) {
       r3=d.p();                // I3
    }
   While (r2==1) { … }         // S2
      // some statements: T1
   If (r1>r2 && r1>r3) { … }   // S1 and S3
      // some statements: T2
   }
} // class
Class B {
    static int m() {
      // some statements: T3
} }   // Class
Class C {
  int m1() {
  int r3;
  D d=new D();
  for (i=0; i<n; i++)
     r3=d.p();                    // I4
     Print(r3);                   // S4
} }  // Class
Class D {
  int p() {
   int r4;  A a=new A();
   r4=a.main();                   // I5
     // some statements: T4
} }  // Class
```

Fig. 11 Code of a sample program

Considering the program code in Fig. 11 and intermediate code generated for that in Fig. 12, $GTE_{A.main}$ can be written as Fig. 13.

Table 1 Distributed execution time, sequential execution time, and speedup for Fig. 11

Sequential time (s)	Distributed time (s)	Speedup
267	221	1.208

```
Method A.main
  Begin method
    Call B.m  // The actual destination of a call
              // is determined using Algorithm 1
    Call C.n
    Call D.p
    Use C.n
    Some_computation
    Use B.m  // determined using the Def-Use
             // chain method
    Use C.n
    Use D.p
EndMethod A.main
Method B.m
  Begin method
  Some_computation
EndMethod B.m
Method C.m1
  Call D.p
  Use D.p
EndMethod C.m1
Method D.p
  Call A.main
  Some_computation
EndMethod D.p
```

Fig. 12 Intermediate code for Fig. 11

```
GTE(A.main)=a1*GTE(B.m)+a2*GTE(C.n)+a3*GTE(D.p)
    +(1-a2)*T(S1)+t1+(1-a1)*T(S2)+(1-a3)*T(S3)+t2
  T(S1)=max((GTE(C.n)+Ci)-(a3*GTE(D.p)),0)
  T(S2)=max((GTE(B.m)+Ci)-(t1+(1-a2)*T(S1)
        +a3*GTE(D.p)+a2*GTE(C.n)),0)
  T(S3)=max((GTE(D.p)+Ci)-((1-a1)*T(S2)+t1
        +(1-a2)*T(S1)),0)
GTE(B.m)=t3
GTE(C.n)=a4*GTE(D.p)+(1-a4)*T(S4)
  T(S4)=max((GTE(D.p)+Ci),0)
GTE(D.p)=a5*GTE(A.main)+t4
```

Fig. 13 General time estimation relationships for Fig. 11

5 Evaluation results

To evaluate the proposed approach, we tested our approach against different types of software systems, and then compared the estimated results obtained against their real life executions. We used the well-known travelling salesman problem (TSP; http://www.adaptivebox.net/CILib/code/tspcodes_link.html) and two practical applications (robot control program and sparse matrix solver) to evaluate the proposed method. The sparse matrix solver is a sparse matrix solver of an electronic circuit simulation generated by the OSCAR FORTRAN compiler. This application is known to have a relatively high level of parallelism (Gotoda *et al.*, 2012). The robot control program is a Newton-Euler dynamic control calculation for the six-degree-of-freedom Stanford manipulator, which has a lower level of parallelism compared to the sparse matrix solver (Gotoda *et al.*, 2012). Our assessment method is such that we first extract the call graph using the method proposed in Section 3 and then extract GTE^{seq} and GTE^{asynch} from the call graph. Then we predict from the GTE relationship the estimated time of the parallel and sequential execution and calculate the speedup for them. Afterward, we distribute them on the network including the number of computers (specified as above) using the jDistributor tool and calculate the parallel and sequential execution time. 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 is the sequential execution time and T_P is the parallel execution time on P processors. We used a network including a number of personal computers (PC). Each PC has an Intel Core i7 920 (2.67 GHz) as the CPU, Intel Gigabit CT Desktop Adaptor EXPI9301CT as the network interface card, and 6.0 GB of memory. In Parsa and Khalilpoor (2006) a tool named jDistributor was provided for semi-automatic distribution of the sequential program on the homogeneous distributed systems. This tool distributes using Java Symphony middleware. The algorithm used in the jDistributor tool is a clustering method and its goal is to find an appropriate clustering for distribution. The results are shown in Tables 2 and 3.

Tables 2 and 3 show that the results obtained using the proposed method are in accordance with their real execution results. That is, when a speedup obtained from the proposed method is larger than 1, through the real execution, its distributed execution is faster than its sequential execution; if the speedup is less than 1, its distributed execution is slower than its sequential execution and in such a case the program should not be executed in distributed form. Indeed, we can specify (predict) based on the speedup obtained from the proposed method whether the source code is distributable or not.

Table 4 shows that the execution time of the robot control program on the network including three processors, will speed up compared with the sequential mode. As Gotoda *et al*. (2012) mentioned, this application has a lower level of parallelism. Therefore, its execution on the network including four and six processors will reduce the speedup compared to the sequential manner. This is due to the increased communication time between processors. In fact, its distributed execution is slower than its sequential execution.

Table 4 also shows that the distributed execution of a sparse matrix solver on the network including three, four, and six computers is faster than its sequential execution. However, in the network including four processors, speedup is more than the speedup of the network including six processors. This is due to the increased communication time between processors. GTE only shows whether a program can be paralleled or not, and provides no information about the way in which the distribution is performed or the type of clustering for the distribution.

Table 2 TSP comparison: estimated execution vs. real execution time using the jDistributor tool on the network including three computers

Number of nodes	Number of edges	Sequential time (ms)		Distributed time (ms)		Speedup	
		Estimated execution time*	Real execution time***	Estimated execution time**	Real execution time***	Estimated execution time	Real execution time***
20	40	475	573	5342	7357	0.089	0.078
40	81	981	1383	5532	7810	0.177	0.177
60	122	2870	3246	5901	8163	0.486	0.398
80	163	7560	11 214	7120	11 109	1.062	1.009
100	204	15 341	19 773	10 098	14 741	1.519	1.341
120	245	25 987	43 517	15 675	30 722	1.657	1.416
140	286	49 621	60 871	19 676	25 362	2.522	2.400

* Using Eq. (7); ** using Eq. (16); *** using the jDistributor tool

Table 3 Comparison of estimated execution time and real execution time using the jDistributor tool on the network including four computers between two famous applications

Application	Sequential time (ms)		Distributed time (ms)		Speedup	
	Estimated execution time*	Real execution time***	Estimated execution time**	Real execution time***	Estimated execution time	Real execution time***
Robot control program	475	573	542	728	0.876	0.787
Sparse matrix solver	981	1383	325	465	3.014	2.977

* Using Eq. (7); ** using Eq. (16); *** using the jDistributor tool

Table 4 Comparison of estimated execution time and speedup on the network including 3, 4, and 6 processors using the robot control program and the sparse matrix solver

Application	Estimated sequential time (ms)			Estimated distributed time (ms)			Speedup		
	3	4	6	3	4	6	3	4	6
Robot control program	475	475	475	237	542	1084	2.008	0.876	0.438
Sparse matrix solver	981	981	981	466	325	346	2.107	3.014	2.836

6 Conclusions

In this paper, we introduce a new approach for source code distributability verification. The approach enables users to determine the expected speedup gains before going into a real-life distributability implementation. We propose an analytical model for object oriented programs, which analyzes the programs statistically through studying the types of synchronous and asynchronous calls inside the source code, and we propose criteria that specify whether a program is suitable for parallelization on homogeneous or heterogeneous processors. Experimental results showed that the proposed approach successfully determines the distributability of different real-life software applications when compared with their real-life sequential and distributed implementations.

The following directions can be explored to extend and improve this work. First, study the effect of code instructions shift at compile time on program concurrency, and see if such shift operations could improve program distributability. Second, study the effect of near optimal partitioning on the analytical model from a performance viewpoint.

References

Al-Jaroodi, J., Mohamad, N., Jiang, H., *et al.*, 2005. JOPI: a Java object passing interface. *Concurr. Comput. Pract. Exp.*, **17**(7-8):775-795. [doi:10.1002/cpe.854]

Berman, F., Wolski, R., Casanova, H., *et al.*, 2003. Adaptive computing on the Grid using AppLeS. *IEEE Trans. Parall. Distr. Syst.*, **14**(4):369-382. [doi:10.1109/TPDS.2003.1195409]

Berman, F., Casanova, H., Chien, A., *et al.*, 2005. New Grid scheduling and rescheduling methods in the GrADS project. *Int. J. Parall. Program.*, **33**(2):209-229. [doi:10.1007/s10766-005-3584-4]

Bushehrian, O., 2010. Automatic actor-based program partitioning. *J. Zhejiang Univ.-Sci. C (Comput. & Electron.)*, **11**(1):45-55. [doi:10.1631/jzus.C0910096]

Buyya, R., Abramson, D., 2009. The Nimrod/G grid resource broker for economic-based scheduling. *In*: Buyya, R., Bubendorfer, K. (Eds.), Market-Oriented Grid and Utility Computing. John Wiley & Sons, Inc., Hoboken, NJ, USA, p.3-27. [doi:10.1002/9780455432.ch17]

Chen, Y., Gansner, E., Koutsofios, E., 1997. A C++ data model supporting reachability analysis and dead code detection. Proc. 6th European Software Engineering Conf. and the 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering, p.414-431. [doi:10.1007/3-540-63531-9_28]

Deb, D., Fuad, M., Oudshoom, M.J., 2006. Towards autonomic distribution of existing object oriented programs. Int. Conf. on Autonomic and Autonomous Systems, p.17. [doi:10.1109/ICAS.2006.61]

Ferenc, R., Beszedes, A., Gyimóthy, T., 2004. Extracting facts with Columbus from C++ code. Proc. 8th European Conf. on Software Maintenance and Reengineering, p.4-8.

Gentzsch, W., 2001. Sun Grid Engine: towards creating a compute power grid. Proc. 1st IEEE/ACM Int. Symp. on Cluster Computing and the Grid, p.35-36. [doi:10.1109/CCGRID.2001.923173]

Gotoda, S., Ito, M., Shibata, N., 2012. Task scheduling algorithm for multicore processor system for minimizing recovery time in case of single node fault. Proc. 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, p.260-267. [doi:10.1109/CCGrid.2012.23]

Healy, C.A., Sjodin, M., Rustagi, V., *et al.*, 1998. Bounding loop iterations for timing analysis. Proc. 4th IEEE Real-Time Technology and Applications Symp., p.12-21. [doi:10.1109/RTTAS.1998.683183]

Korn, J., Chen, Y., Koutsofios, E., 1999. Chava: reverse engineering and tracking of Java applets. Proc. 6th Working Conf. on Reverse Engineering, p.314-325. [doi:10.1109/WCRE.1999.806970]

Koskinen, J., Lehmonen, T., 2010. Analysis of ten reverse engineering tools. Advanced Techniques in Computing Sciences and Software Engineering, p.389-394. [doi:10.1007/978-90-481-3660-5_67]

Maani, R., Parsa, S., 2007. An algorithm to improve parallelism in distributes systems using asynchronous calls. Proc. 7th Int. Conf. on Parallel Processing and Applied Mathematics, p.49-58. [doi:10.1007/978-3-540-68111-3_6]

Matzko, S., Clarke, P.J., Gibbs, T.H., *et al.*, 2002. Reveal: a tool to reverse engineer class diagrams. Proc. 40th Int. Conf. on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, p.13-21.

Nitzberg, B., Schopf, J.M., Jones, J.P., 2004. PBS Pro: grid computing and scheduling attributes. *In*: Grid Resource Management. Kluwer Academic Publishers, Norwell, MA, USA, p.183-190. [doi:10.1007/978-1-4615-0509-9_13]

Parsa, S., Khalilpoor, V., 2006. Automatic distribution of sequential code using JavaSymphony middleware. Proc. 32nd Conf. on Current Trends in Theory and Practice of Computer Science, p.440-450. [doi:10.1007/11611257_42]

Raza, A., Vogel, G., Plödereder, E., 2006. Bauhaus—a tool suite for program analysis and reverse engineering. Proc. 11th Ada-Europe Int. Conf. on Reliable Software Technologies, p.71-83. [doi:10.1007/11767077_6]

Schoeberl, M., 2006. A time pedictable Java pocessor. Proc. Design, Automation and Test in Europe, p.1-6. [doi:10.1109/DATE.2006.244146]

Schoeberl, M., Pedersen, R., 2006. WCET analysis for a Java processor. Proc. 4th Int. Workshop on Java Technologies for Real-Time and Embedded Systems, p.202-211. [doi:10.1145/1167999.1168033]

Thain, D., Tannenbaum, T., Livny, M., 2005. Distributed computing in practice: the condor experience. *Concurr. Comput. Pract. Exp.*, **17**(2-4):323-356. [doi:10.1002/cpe.938]

Zhang, Q.F., Qiu, D.H., Tian, Q.B., *et al*., 2010. Object-oriented software architecture recovery using a new hybrid clustering algorithm. Proc. 7th Int. Conf. on Fuzzy Systems and Knowledge Discovery, p.2546-2550. [doi:10.1109/FSKD.2010.5569799]

Zhou, S., Zheng, X., Wang, J., *et al*., 1993. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exp.*, **23**(12):1305-1336. [doi:10.1002/spe.4380231203]

Zima, H., Chapman, B., 1991. Supercompilers for Parallel and Vector Computers (1st Ed.). Addison Wesley, MA, USA.

JOURNAL OF ZHEJIANG UNIVERSITY SCIENCE ABC

Google | Google
Search in this s | Search

Home | Current Issue | Online Submission | Readers Register | Contact Us

CONTENTS
Current Issue
Back Issue
❶ Articles in Press
Online First
Subscription
INSTR. FOR AUTHOR
Preparing Manuscript
Online Submission
Revision & Acceptance
❷ Cross Check
FOR REVIEWER
❸ Int'l Reviewer
Guidelines for Reviewer
ABOUT JZUS
Editorial Board >
e-Link
JZUS Events
Editor Paper
Thank You Card
Contact us
Publishing Service
Polishing & Checking
Advertisements
Conference Info
Publicizing

JZUS-A wins "The Chinese Government Award for Publishing" for Journals
浙江大学学报(英文版)A辑荣获"第二届中国出版政府奖":首届期刊奖

J. Zhejiang Univ.-SCI. A (Applied Physics & Engineering) IF=0.527(2012); 0.408(2011)

J. Zhejiang Univ.-SCI. B (Biomedicine & Biotechnology) IF=1.108(2012); 1.099(2011)

J. Zhejiang Univ.-SCI. C (Computers & Electronics) IF=0.297(2012); 0.308(2011)

❹ Highlights

A Fused empirical mode decomposition and wavelets for locating combined damage in a truss-type structure through vibration analysis
Structural health monitoring (SHM) is a relevant topic for civil systems and involves the monitoring, data processing and interpretation to evaluate the condition of a structure, in order to detect damage. In real structures, two...
DOI:10.1631/jzus.A1300030 Downloaded: 443 Clicked:960 Cited:0 Comments:0 <Full Text> <PPT> 454 ❺

B Menstrual blood-derived mesenchymal stem cells differentiate into functional hepatocyte-like cells
Orthotopic liver transplantation (OLT) is the only proven effective treatment for both end-stage and metabolic liver diseases. Hepatocyte transplantation is a promising alternative for OLT, but the lack of available donor livers h...
DOI:10.1631/jzus.B1300081 Downloaded: 396 Clicked:477 Cited:0 Comments:0 <Full Text>

C Underwater glider design based on dynamic model analysis and prototype development
Underwater gliders are efficient mobile sensor platforms that can be deployed for months at a time, traveling thousands of kilometers. Here, we describe our development of a coastal 200 m deep underwater glider, which can serve as...
DOI:10.1631/jzus.C1300001 Downloaded: 772 Clicked:901 Cited:0 Comments:0 <Full Text> <PPT> 502

Top 10 cited A B C (2011~2013) ❻
Evaluation of the heat tran...
Discrete element modelling ...
Experimental evaluation of...
Preliminary report of the 2...
Modified particle swarm opt...
more

Newest cited A B C (2011~2013) ❼
Modeling pollutant transpor...
Shear buckling strength of...
Physically-based approach t...
Information fusion diagnosi...
CFD analysis of a transfer ...
more

Top 10 DOIs Monthly ❽
Curvatures estimation on tr...
Be careful! Avoiding duplic...
Is alcohol consumption asso...
Diagnosis and treatment of ...
Dental erosion and severe t...
more

Newest 10 comments ❾
Physically-based approach t...
Separation of phosphatidylc...
Application of system dynam...
More general results on mix...
Effect of vertebrate hormon...
more

Top 10 downloads A B C ❿
Critical review in adsorpti...