第38卷　第24期
Vol.38　No.24

计　算　机　工　程
Computer Engineering

2012年12月
December 2012

• 云计算专题 •　文章编号：1000—3428(2012)24—0027—05　文献标识码：A　中图分类号：N945

基于云计算的并行测试方案设计与实现

黄晓玲，陈桂林，赵生慧

(滁州学院计算机与信息工程学院，安徽 滁州 239000)

摘　要：目前对软件测试用例的需求在以指数级增长，导致测试资源相对不足、测试成本高、测试用例执行效率低等问题更加突出。为解决上述问题，设计一个基于云计算的并行测试方案，采用有限状态机定义测试对象及测试过程中的状态迁移，借鉴随机路线的思想，提出一个并行测试用例生成算法，在此基础上给出基于 MapReduce 模型和云计算平台的并行测试脚本。实验结果表明，与顺序执行测试序列相比，该方案的加速比可达 20，测试效率有明显提高。

关键词：云计算；云测试；并行测试；有限状态机；MapReduce 模型

Design and Implementation of Parallel Testing Schema Based on Cloud Computing

HUANG Xiao-ling, CHEN Gui-lin, ZHAO Sheng-hui

(College of Computer and Information Engineering, Chuzhou University, Chuzhou 239000, China)

【Abstract】 With the demands for software test cases in exponential growth, the problems such as relative shortage of test resources, high cost of testing and low efficiency execute of test cases are increasingly outstanding. To solve the above problems, a parallel test schema based on cloud computing is proposed, which uses Finite State Machine(FSM) to define test object and state transition in testing process. Parallel test case generation algorithm based on random routes idea is proposed, and a parallel test script based on the MapReduce and cloud computing platform is designed. Experimental results show that compared with test sequence of executing sequential, the acceleration of schema is up to 20, so the test efficiency is significantly improved.

【Key words】 cloud computing; cloud testing; parallel testing; Finite State Machine(FSM); MapReduce model

DOI: 10.3969/j.issn.1000-3428.2012.24.007

1　概述

随着网络及 Internet 的普及，大量的软件已经从传统的单机或者 C/S 环境转移到基于网络的分布式环境中，这一类软件一般都具有规模大、复杂性高等特点。与传统的单机或者 C/S 环境中的软件相比，软件中出现错误和缺陷的概率也明显增加。软件测试是发现软件缺陷、提高软件质量的重要途径。在进行软件测试时，需要根据软件的应用背景及功能设计相应的测试环境及测试用例。在传统的测试环境中，由于受资源等因素的约束，为了提高测试效率，一般均通过精简测试用例的方法来缩短运行时间，但是测试用例的精简降低了发现缺陷的概率，影响了测试质量及软件质量。为了解决这一问题，并行测试技术被引入到软件测试领域[1]。并行测试的基本思想是构建集群环境，通过并发执行测试任务来提高测试执行效率，同时为分布式软件构建一个与实际应用背景高度一致的测试环境。但是，构建集群需要根据测试需求的峰值配置物理设备，导致了较高的成本开销及相对较低的资源使用率，也增加了管理与维护的复杂度。

云计算是最近几年刚刚兴起的一种网络计算模式，它能够根据需求以服务的方式向用户提供动态弹性可伸缩的资源[2]，用户根据使用情况付费，从而降低了用户的拥有与管理成本。

云计算的这些特点为构建高效低成本的测试环境提供了新的途径，但同时也提出了新的挑战。在云计算环境中，传统的测试用例不能直接应用，需要根据云计算的特点对测试用例进行并行化处理。为解决这一问题，同时提高软件测试效率，本文提出一个基于云计算的软件测试方案，给出并行测试用例生成算法和基于 MapReduce 架构的测试用例并行化脚本，并实现了一个原型系统。

2　相关工作

将并行技术引入到测试领域，能够实现多个任务的同时测试，从而显著提高测试效率与质量。已有的并行测试

基金项目：安徽省自然科学基金资助项目(11040606M152)；滁州学院科研基金资助项目(2011kj015B)

作者简介：黄晓玲(1984－)，女，助教、硕士，主研方向：云计算，软件测试；陈桂林、赵生慧，教授

收稿日期：2012-05-03　**修回日期：**2012-07-11　**E-mail：**hxl1984007@gmail.com

技术主要用于硬件测试，其作用也仅限于加速硬件的测试过程[3-4]。到目前为止，用于提高软件系统测试效率的研究相对较少。文献[5]介绍了使用并行和分布式技术测试计算机软件系统的优势，提出了一个并行测试框架，但没有给出具体的测试方案与实现方法。文献[6]提出了一个针对分布式软件系统的并行测试设计方案，也没有给出实现细节。

在基于云计算的测试研究方面，文献[7]提出根据被测试程序的特征和所执行的测试类型确定测试何时适合迁移到云环境中，但并没有涉及具体的实施方案。文献[8]设计了一个基于云计算的 D-cloud 平台用于测试大规模软件的可靠性。平台使用 QEMU 对每个节点进行硬件错误注入，云平台进行各节点的管理，但没有给出实验结果和分析。

在云测试方面，与并行测试相结合的研究较少。YETI 是一个自动化随机测试工具，由于面临着性能问题，因此文献[9]提出在云平台上部署同时执行测试用例的环境来提高效率，但未说明测试用例是如何并行的。文献[10]针对 Cloud9 测试工具在传统运行环境下出现的内存和 CPU 资源不足的问题，提出利用云计算资源对分支代码进行并行测试，提高了测试效率，但没有给出并行测试用例的具体执行过程。因此，本文提出一个包括测试用例设计到执行完整过程的并行测试方案。

3 并行测试用例生成

3.1 基于有限状态机的测试对象描述

有限状态机(Finite State Machine, FSM)有成熟的理论基础[11]，被广泛应用于软件描述和软件测试。它可以准确描述测试对象的动态行为，易于生成测试用例及对测试结果进行分析，保证测试覆盖的充分性。

基于 FSM 的测试模型假设软件在某个时刻总处于某个状态，并且当前状态决定了软件可能的输入，而且从该状态向其他状态的迁移决定于当前的输入。有限状态机模型特别适用于把测试数据表达为输入序列的测试方法，并能利用图的遍历算法自动产生输入序列。本文在文献[11]的基础上，给出测试对象、测试对象状态迁移以及状态迁移图的定义。

定义 1 测试对象

假设节点的初始状态和目标状态集合一致，测试对象定义为 $M=(S, A, \delta)$，其中，S 是非空的测试对象的有限状态集合，$S_i(1\leqslant i\leqslant n)\subseteq S$；A 是窗口控件动作组成的有限集合；$\delta$：$S\times A\rightarrow S$ 是状态转换函数，表示测试对象由于控件动作而引起的状态转换。

定义 2 测试对象状态迁移

Stransition=(souce,a,target)

其中，a 是发生的一个控件动作；source 是 a 发生前的测试对象状态；target 是 a 发生后的测试对象状态。

定义 3 状态迁移图

$G=<V, E>$

其中，V 代表节点，每个 $v\in V$ 呈现当前对象状态；$E\subseteq V\times V$ 代表连接节点的有向边；$(v_i, v_j)\in E$ 代表触发状态 v_i 到 v_j 的事件。

例如，以百度 8 个超链接之间的跳转是否合法为测试对象，如图 1 所示。选取其中 3 个状态，根据上述定义得出状态迁移图，如图 2 所示，其中，S_0 表示“新闻”；S_1 表示“知道”；S_2 表示“地图”；I_0 表示触发“新闻”超链接到达 S_0 状态；I_3 表示触发“知道”超链接到达 S_1 状态；I_1-I_2、I_5-I_8 与 I_3 的功能类似。

图 1 含有 8 个超链接的 Web 页面

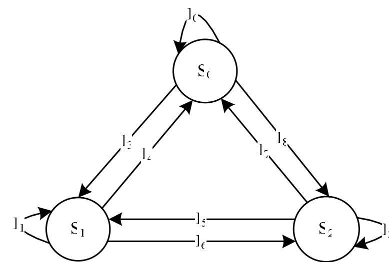

图 2 状态迁移图

3.2 基于随机路线的并行测试用例生成算法

并行测试用例是指将测试对象的测试用例划分为若干相对独立的测试序列，通过各测试子序列的并行执行实现。由于测试对象的并行处理能力取决于系统所采用算法的并行性，因此从理论上说，采用高效的算法能提高并行测试效率。

本文采用 GraphML[12]描述图 2，得出状态和状态序列，命名为 init.xml。其中，<node>代表图 2 中的节点；<edge>指连接节点的状态迁移，通过节点属性 source 和 target，获得迁移后的目标状态。对 init.xml 具体描述如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<graph id='G'   edgedefault='directed'>
  <node id ="S0"/>
  <node id ="S1"/>
      <node id ="S2"/>
…
  <edge id="I0" source="S0" target="S0" />
  <edge id="I1" source="S1" target="S1"/>
  <edge id="I2" source="S2" target="S2" />
  <edge id="I3" source="S0" target="S1"/>
…
</graph>
```

基于随机路线的并行测试用例生成算法的主要思想为：从 init.xml 文件的<node>标记随机选取一个状态 srcState 作为初始状态，从该状态出发，从可能的<edge>中随机选取一个迁移路径 srcTransition，执行触发事件

triggerEvent，到达目标状态 tarState。将 srcTransition 记录到 XML 文件中。通过使用变量 time 控制运行时间，设当前系统时间为 currentTime，计算结束时间为 end。测试用例生成过程一直循环持续到停止时间为止。算法伪代码如下：

输入 init.xml

输出 TestCase.xml

```
1.初始化
end=time+currentTime;
//读取 init.xml 文件
parseURI("init.xml");
//根据 init.xml 中(source, target)获得迁移事件
triggerEvent ();
2.产生测试序列
While (System.currentTimeMillis() <= end){
if (srcState. triggerEvent ()!= null) {
//从当前状态出发的<edge>，随机产生边作为路径
srcTransition=Random(srcState.transitionSize());
//获得并触发当前路径的状态迁移事件
get(srcTransition). triggerEvent ();
//将目标状态赋给当前状态
srcState = get(srcTransition).tarState;
//输出当前路径到 TestCase.xml 中;
output(srcTransition,TestCase.xml)
}
else
//从<node>节点中任取一个节点作为初始状态
srcState =Random(state);
}
```

基于随机路线的并行测试序列生成算法的相关文献较少，只有文献[13]提出了静态随机并行测试序列生成算法，其时间复杂性为 $O(n^2)$。而本文算法的时间复杂度为 $O(n)$，对于给定的任意序列，算法总可以在线性时间内求得可行解，具有较高的测试效率。

根据上述算法得出的并行测试用例 TestCase.xml 如下所示：

```
<?xml version="1.0" ?>
<Navigation>
<State>
<transition>I0</transition>
<transition>I3</transition>
<transition>I6</transition>
</State>
<State>
<transition>I6</transition>
<transition>I7</transition>
<transition>I0</transition>
</State>
…
</Navigation>
```

其中，<State>用来标识独立测试用例，如 I_0-I_3-I_6 表示从 S_0 到 S_2 的一条测试路径。

4　基于云环境的测试用例并行化实现

4.1　并行测试脚本设计

本文采用 MapReduce 设计测试脚本。MapReduce 是并行计算模型，它将复杂的运行于大规模集群上的并行计算过程抽象到 Map 和 Reduce 这 2 个函数。Map 接受一组数据并将其转换为一个<key, value>对列表，输入域中的每个元素对应一个<key, value>对。Reduce 接受 Map 函数生成的列表，然后根据它们的 key 汇总<key, value>对列表，形成最终结果。

4.1.1　测试用例处理

本文基于 MapReduce 的<key,value>为每条测试用例前面加入执行命令。根据 3.2 节 TestCase.xml 中的<State>标记读出并行测试用例，在每条测试用例前面加入运行测试工具及脚本命令，最终生成.txt 格式的测试文件。测试文件每行命令格式如下：

<TestCase ID><运行工具命令 TestScript TestCase>

4.1.2　脚本设计

基于 MapReduce 的并行脚本设计思想如下：Map 阶段主要抽取出唯一标识的测试用例编号，调用测试工具和脚本执行测试用例，并返回各条测试用例结果。Reduce 阶段主要对 Map 阶段的测试结果进行合并。对成功运行的测试用例进行统计，否则，返回相应的测试日志。测试结果形式为：

<TestCaseID><TestResult>

脚本如下所示：

```
//Map 阶段
public static class MapClass extends
        MapReduceBase implements Mapper{
private Text TestCaseID, TestLog;
public void map(LongWritable key, Text
value, OutputCollector output){
    String line = value.toString();   //读取文本每行内容，调用
//测试工具和脚本执行
    StringTokenizer itr = new StringTokenizer(line);
//找到 TestCaseID
TestCaseID= itr.firstTokens();
TestLog=itr.lastTokens();     //启动测试工具，执行脚本输出测
//试用例的结果
output.collect(TestCaseID, TestLog);
        }
}
//Reduce 阶段
```

```
public static class Reduce extends
MapReduceBase    implements Reducer{
      public void reduce(Text key, Iterator values,
  OutputCollector output) {
          int sum=0;
          while(values.hasNext()){
            if(values.equals("pass"))
//统计测试结果成功个数
                    sum+=((IntWritable)values.next();
                  output.collect("成功测试用例数",
new IntWritable(sum));
              else
                    output.collect(key, values); }
}
}
```

4.2 并行测试的环境构建与测试流程

Hadoop[14]是Apache组织的开源项目，其框架中最核心的设计是HDFS和MapReduce。以Hadoop平台为设计背景，结合云计算技术给出并行测试用例的执行环境，如图3所示。

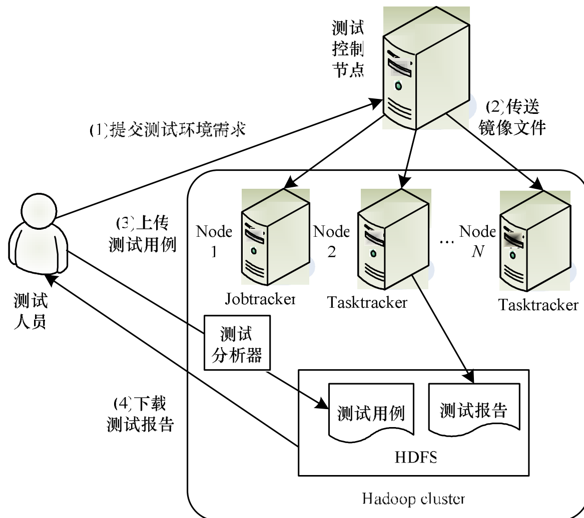

图3 基于云的并行测试执行环境

在图3中，测试控制节点对虚拟节点、镜像文件等进行管理，对于用户申请的环境配置需求，根据当前虚拟节点资源使用情况，决定是否接受任务。现有的IaaS平台都可实现该功能，如OpenStack[15]。测试分析器是根据4.1节中测试用例处理设计的一个具体实现，是运行在服务器端的工具，实现对上传的测试用例进行预处理后再保存到HDFS中。测试报告是运行MapReduce并行脚本后返回的运行结果，测试结果形式为：

<TestCaseID><TestResult>

本文选择OpenStack作为测试控制节点。它提供Web界面形式对镜像文件进行管理，用户可以根据需要自己定制或上传镜像文件。镜像文件包括操作系统和软件，格式为.img。当控制节点发送包含Hadoop的镜像文件给各个测试子节点时，就搭建了一个Hadoop集群环境。

测试流程大体如下：

(1)测试人员提交所需的软、硬件测试环境给控制节点OpenStack。

(2)在OpenStack下构建Hadoop集群测试环境。

(3)基于3.2节算法生成的测试用例需要经过测试分析器的处理后，再上传到HDFS中。

(4)执行4.1.2节的测试脚本驱动被测系统运行测试用例，并将生成测试结果保存在HDFS中。

(5)通过HDFS下载测试报告，对测试中的问题进行评价和定位。

其中，JobTracker作为主控节点，用于调度和管理TaskTracker；TaskTracker负责执行JobTracker分配的任务，并将测试结果存放在HDFS中。

5 实验与结果分析

为了检验并行测试方案的有效性，以图1所示的百度8个超链接之间的跳转是否合法为测试对象，生成独立测试用例1 520条。

实验环境使用diablo版本的OpenStack，搭建在20台服务器上，每台服务器配置为2×E5504型号CPU、8 GB内存、2×146GSAS型号磁盘、2×千兆网卡，支持全虚拟化技术。服务器操作系统为ubuntu11.10。在OpenStack平台上申请虚拟机节点。每个节点操作系统为Ubuntu Natty 11.04，Hadoop版本采用0.20.0，测试工具为RFT7.0。

实验1 测试执行时间比较

分别在单机和基于云的集群环境下统计执行测试用例所需时间。在单机环境下，测试用例执行时间 $Ts \approx 750$ min。图4表明在不同节点数量下，在基于云的集群环境下所需的执行时间。其中，x轴是基于云的集群中节点数；y轴是执行测试用例所需的时间。当虚拟机节点数达到100时，运行并行测试用例时间 Tp=35 min，加速比 $\alpha = Ts/Tp \approx 20$。

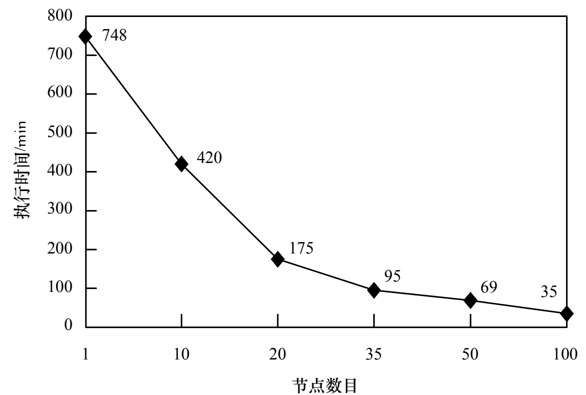

图4 基于云的集群环境下测试执行时间

实验2 CPU和内存利用率比较

首先申请10个物理节点，每台机器独立安装测试工具RFT，并为每台机器平均分配152条测试用例。当手工

同时运行 10 台机器的测试脚本来驱动测试用例执行时，在节点运行良好的情况下，花费时间约为 383 min。

当虚拟机节点数为 10 个时，运行测试用例时间大约为 420 min。虽然在 10 个节点下独立节点的运行时间比云环境运行时间少，但云环境节点的平均负载相对于独立节点在 CPU、内存利用率方面均衡得多。如图 5 和图 6 所示，云环境下 CPU 利用率始终维持在 35.4%左右，内存利用率大约为 32.7%，而独立节点的 CPU 和内存利用率上下浮动比例较大。当子节点因 CPU 或内存利用率过高发生死机时，云环境节点相对于独立节点环境不会造成测试用例遗漏或无法执行，提高了测试环境的健壮性。

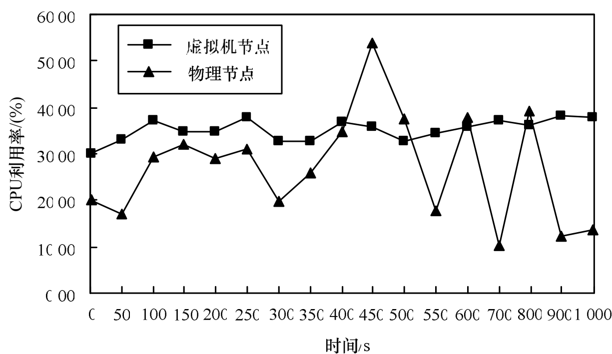

图 5　CPU 利用率对比

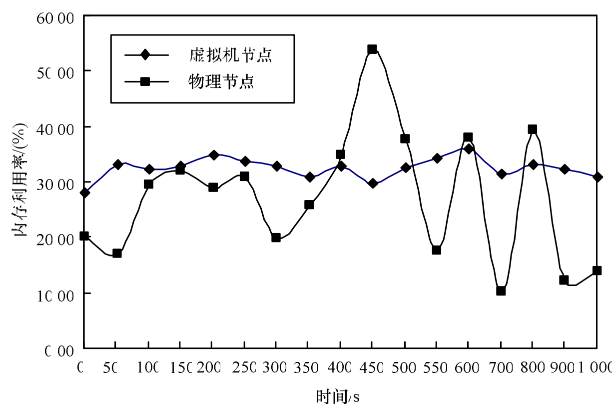

图 6　内存利用率对比

在现有并行测试的研究中，实验环境主要搭建在基于物理节点的集群中。如文献[6]给出物理节点集群下的实验结果，顺序执行测试序列花费 12 h 52 min，并行执行测试序列花费 1 h 41 min，加速比为 7.7。文献[16]在 Hadoop 环境下进行实验，物理节点数为 100 时，并行测试所需时间为 4 min，而在单机环境下，顺序执行测试所需时间为 69 min，加速比为 17.25。本文的实验环境搭建在云环境下，测试结果加速比为 20，相比于非云平台，明显提高了测试效率和测试健壮性。

6　结束语

本文结合云计算技术，提出了基于随机路线的并行测试用例生成算法和基于 MapReduce 并行化脚本实现的并行测试方案，并设计实现了原型系统。实验结果证明，基于云平台的并行测试方案减少了测试用例执行时间，加速了测试过程，降低了测试环境成本，提高了测试质量。目前本文方案适用于系统测试阶段，后期的工作是对其进行扩展，使其适用于整个软件测试过程。

参考文献

[1] Parallel Testing[EB/OL]. [2012-04-10]. http://www.blurtit.com/q191888.html.

[2] Armbrust M. Above the Clouds: A Berkeley View of Cloud Computing[R]. Berkeley, USA: Electrical Engineering and Computer Sciences University of California, Tech. Rep.: UCB/EECS-2009-28, 2009.

[3] Karimi F, Irrinki S, Crosby T, et al. Parallel Testing of Multi-port Static Random Access Memories[J]. Microelectronics Journal, 2003, 34(1): 3-21.

[4] Kim W W. Parallel Testing Method by Partitioning Circuit Based on the Exhaustive Test[J]. Lecture Notes in Computer Science, 2004, 3044(2): 262-271.

[5] Starkloff E. Designing a Parallel, Distributed Test System[J]. IEEE Aerospace and Electronic Systems Magazine, 2005, 16(6): 3-6.

[6] Lastovetsky A. Parallel Testing of Distributed Software[J]. Information and Software Technology, 2009, 47(10): 657-662.

[7] Parveen T, Tilley S. When to Migrate Software Testing to the Cloud?[C]//Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation. Paris, France: IEEE Computer Society, 2010.

[8] Banzai T, Koizumi H, Kanbayashi R, et al. D-cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology[C]//Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. Melbourne, Australia: IEEE Computer Society, 2010.

[9] Oriol M, Ullah F. YETI on the Cloud[C]//Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation. Paris, France: IEEE Computer Society, 2010.

[10] Ciortea L, Zamfir C, Bucur S, et al. Cloud9: A Software Testing Service[J]. ACM SIGOPS Operating Systems Review, 2010, 43(4): 5-10.

[11] Chow T S. Testing Design Modeled by Finite State Machines[J]. IEEE Transactions on Software Engineering, 1978, 4(3): 178-187.

[12] GraphML Specification[EB/OL]. (2010-09-21). http://graphml.graphdrawing.org/specification.html.

[13] 许　明，胡雷刚. 并行测试任务可并行性分析研究[J]. 计算机工程, 2009, 35(4): 56-57.

[14] Hadoop[EB/OL]. [2012-03-20]. http://hadoop.apache.org/.

[15] OpenStack[EB/OL]. [2012-04-08]. http://www.openstack.org/.

[16] Parveen P, Tilley T. Towards a Distributed Execution Framework for JUnit Test Cases[C]//Proc. of IEEE International Conference on Software Maintenance. [S. l.]: IEEE Press, 2009: 425-428.

编辑　张　帆