中国科学: 信息科学 2017 年 第 47 卷 第 6 期: 715–735
SCIENTIA SINICA Informationis
论文

云环境中面向服务软件的演化部署优化方法

李琳[1,2], 应时[1,2*], 董波[1,2], 王蕊[1,2]

1. 武汉大学软件工程国家重点实验室, 武汉 430072
2. 武汉大学计算机学院, 武汉 430072
* 通信作者. E-mail: yingshi@whu.edu.cn

收稿日期: 2016–11–08; 接受日期: 2017–01–13; 网络出版日期: 2017–06–06
国家高技术研究发展计划 (863) (批准号: 2012AA011204) 和国家自然科学基金 (批准号: 61373038、61672392) 资助项目

摘要 针对现有的部署优化方法在求解云环境中面向服务软件的部署优化问题时, 无法处理服务实例和虚拟机实例的伸缩以及无法保障求解质量等问题, 本文提出了一种新的部署优化方法. 该方法以提高面向服务软件的运行性能和降低运行成本为目标构建问题模型, 并设计了一种基于遗传算法的 MGA-DO 算法对其进行求解. MGA-DO 算法采用基于组的编码方式对软件的部署方案进行编码, 然后结合基于组的单点交叉操作, 实现了在优化过程中对服务实例和虚拟机实例的伸缩. 此外, 该算法引入现有的部署优化经验, 设计了多种局部搜索规则, 以进一步提高算法的求解性能. 最后, 一系列模拟实验表明, 相比现有的算法, MGA-DO 算法在求解所研究的问题时表现出了更好的性能.

关键词 云计算, 面向服务的软件, 性能, 成本, 部署优化, 遗传算法

1 引言

云计算是随着 IT 技术的快速发展而出现的一种新的计算范式 [1], 它允许用户按照实际需求租用相关计算资源, 满足了人们对效用计算的长期渴求 [2], 因此受到了工业界和学术界的广泛关注. 随着云计算技术的逐渐成熟, 越来越多的企业和组织开始将其分布式软件部署到云环境中, 以利用云平台所带来的便利条件.

面向服务的体系架构以服务作为基本的构造单元, 通过动态组合不同的服务提供复杂的业务流程, 具有可重用、松耦合、易扩展等特点 [3], 已成为当前分布式软件的主要架构模式 [4]. 当将面向服务的软件部署到云环境中时, 软件供应商通常会将软件的每一个服务实例化为一个或多个实例, 并向云平台租用一定数量的虚拟机实例来部署这些服务实例, 部署所需的服务实例数和虚拟机实例数可根据需要进行伸缩. 对于给定的面向服务软件和云平台, 通常存在多种不同的部署方案. 部署方案的选择对

引用格式: 李琳, 应时, 董波, 等. 云环境中面向服务软件的演化部署优化方法. 中国科学: 信息科学, 2017, 47: 715–735, doi: 10.1360/N112016-00200
Li L, Ying S, Dong B, et al. Evolutionary deployment optimization for service-oriented software in cloud (in Chinese). Sci Sin Inform, 2017, 47: 715–735, doi: 10.1360/N112016-00200

软件的运行性能和运行成本具有重大影响[5], 为追求利润最大化, 软件供应商需要为软件选择具有较低运行成本和较高运行性能的部署方案. 然而, 巨大的选择空间和冲突的优化目标导致这一问题成为典型的 NP 难题[6], 如果采用人工方法进行搜寻, 将会耗费大量的时间、人力和成本.

为云环境中面向服务软件寻找最优部署方案实际上就是寻找服务实例与虚拟机实例之间的最优分配关系, 属于典型的组合优化问题. 对于这类问题, 通常可以采用元启发式算法自动搜索解空间, 提高问题的求解效率. 遗传算法[7] 是模拟自然进化过程, 搜索最优解的一种典型的元启发式算法, 且已经被证明是求解复杂组合优化问题的最有效方法之一[8]. 目前, 大多数部署优化算法都是基于遗传算法进行设计的[9,10]. 然而, 这些算法通常都假定组合对象数目确定, 无法处理本文问题中服务实例数和虚拟机实例数的伸缩特性. 此外, 现有的算法通常只是对元启发式算法的简单应用, 优化过程具有较强的随机性, 优化结果质量波动较大. 为解决上述问题, 本文基于遗传算法提出了一种新的软件部署优化算法, 旨在快速、稳定地为云环境中面向服务的软件找到权衡性能和成本的最优部署方案.

2 相关工作

软件部署优化问题一直是软件工程领域研究的热点, 学术界和工业界已涌现出大量的高水平研究成果. Jayasinghe 等[11] 提出的基于结构化约束敏感的软件部署方法, 通过将数据中心和应用服务模型作为问题的输入, 并针对需求约束、通讯约束和可用性约束, 采用 4 种近似值算法以分层的方式实现大规模的软件部署与优化过程. Malek 等[12] 提出的可扩展的部署优化框架, 通过构建分布式软件系统的服务质量权衡模型, 改进相应的部署优化算法, 使其能够根据用户 QoS 偏好找到最合适的软件部署方案. White 等[13] 提出的混合型空间部署优化算法 Scatter, 通过将启发式装箱算法和粒子群演化算法相结合, 以最小化分布在无线网络上实时系统的能量消耗为目标, 能够根据当前网络链接和处理器节点上能量消耗的变化, 自动导出满足系统各项约束的最优部署拓扑结构. 张晓薇等[14] 提出的软件体系结构驱动的部署优化方法, 通过利用体系结构驱动的方法构建优化模型, 并综合利用贪心算法和混合整数非线性规划算法对其进行求解, 满足了网络化移动应用在不同应用场景下的部署方案生成需求. 然而, 这些方法都假定部署的组件数和可用的硬件节点数都是确定的, 无法解决本文问题中, 服务实例和虚拟机实例的伸缩问题.

针对云环境下的软件部署问题, Yusoh 等[15~17] 基于遗传算法提出了一系列优化方法, 用于解决组合 SaaS 应用的放置、构件聚类和可扩展性问题. 赵秀涛等[18] 通过将云环境下 SBS 的资源分配问题转化为服务选择问题, 提出了一种基于服务选取的 SBS 资源优化分配模型, 并设计了一种混合遗传算法来求解该模型. 孟凡超等[19] 针对 SaaS 构件优化放置问题, 以提高云资源利用率, 降低云资源使用成本为目标, 提出了一种基于混合遗传模拟退火算法的 SaaS 构件优化放置方法. 然而, 这些方法只考虑了虚拟机实例的伸缩特性, 而没有考虑服务实例的伸缩特性.

也有少数研究综合考虑了服务实例和虚拟机实例的伸缩特性. Frey 等[20] 基于遗传算法和 CDOsim 工具, 提出了一种基于模拟的遗传算法 CDO-Xplore, 支持传统软件向云环境迁移时, 对软件的部署架构和运行时的重配置规则进行优化. 然而, 该方法需要事先获得软件在整个运行过程中的负载情况. Wada 等[21] 针对云环境中面向服务应用部署优化问题, 提出了一种多目标遗传算法 E^3-R, 为应用寻找一组既能满足应用的 SLAs, 又能平衡多个相互冲突的 QoS 目标的 Pareto 最优部署配置. 然而, 该方法的优化过程存在较强的随机性, 导致优化效果无法得到有效的保障.

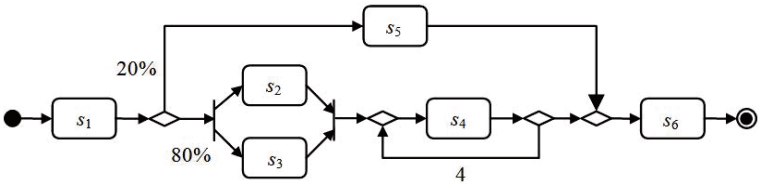

图 1 一个面向服务软件示例

Figure 1 An example for service-oriented software

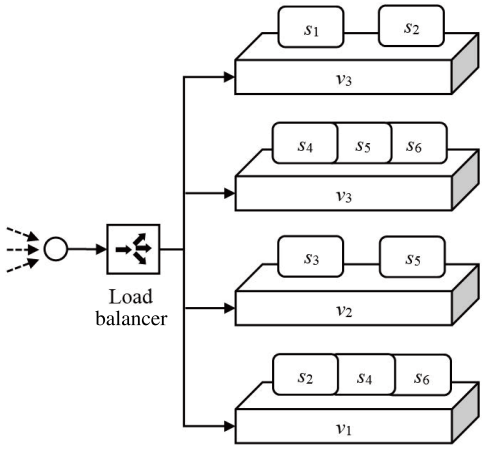

图 2 一个部署方案示例

Figure 2 An example for deployment architecture

3 问题模型

3.1 面向服务软件的部署方案

面向服务的软件可以看成由一组服务按照一定的逻辑交互关系 (包括顺序、选择、并发和循环) 组合而成的一个应用. 图 1 显示了一个面向服务软件示例. 该软件由 $s_1 \sim s_6$ 6 个服务组成, 并从服务 s_1 开始执行, 在其执行完以后, 将以 20% 的概率执行服务 s_5, 然后再执行服务 s_6; 或在服务 s_1 执行完以后以 80% 的概率并发执行服务 s_2 和 s_3, 等待服务 s_2 和 s_3 都执行完后, 重复执行 4 次服务 s_4, 再执行服务 s_6.

云平台为软件服务供应商提供了多种类型的虚拟机实例用以部署软件应用. 软件服务供应商在部署面向服务的软件时, 通常会遵循以下规则, 即一个服务实例只能被部署在一个虚拟机实例上, 一个虚拟机实例可以同时部署多个服务实例, 但同一个服务的多个实例必须被部署在不同的虚拟机实例上, 它们之间通过负载均衡器实现请求的调度.

图 2 给出了图 1 中的面向服务软件的一种可能的部署方案. 在该部署方案中, 服务 s_2, s_4, s_5, s_6 分别被实例化为两个服务实例, 而服务 s_1, s_3 则分别被实例化为一个服务实例, 且服务 s_1 和 s_2 分别有一个实例被部署在类型为 v_3 的一个虚拟机实例上, 服务 s_4, s_5, s_6 分别有一个实例被部署在另一个类型为 v_3 的虚拟机实例上, 服务 s_3 和 s_5 分别有一个实例被部署在类型为 v_2 的一个虚拟机实例上, 服务 s_2, s_4 和 s_6 分别有一个实例被部署在类型为 v_1 的一个虚拟机实例上.

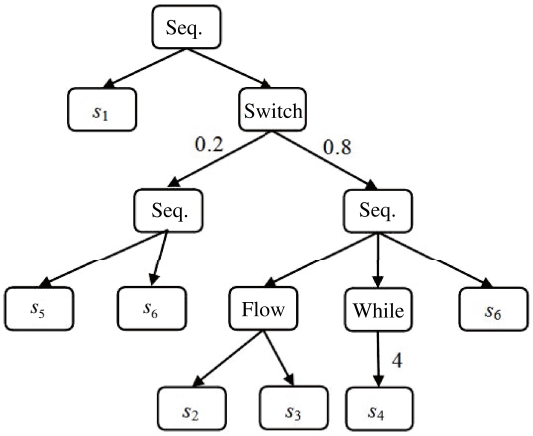

图 3　一个面向服务软件的交互关系树

Figure 3　An inter-relationship tree for service-oriented software

3.2　部署优化模型

本文所研究的问题实际上就是根据面向服务软件各服务的资源需求以及当前云环境所提供的资源, 确定服务实例到虚拟机实例的一种分配关系, 使得面向服务软件具有最优的运行性能和运行成本. 在云计算环境中, 资源通常包含多种类型, 如 CPU、内存、硬盘、带宽等. 为简化问题, 本文只考虑 CPU 和内存两类资源, 并假设硬盘和带宽资源充足, 服务之间的通信所带来的性能代价可忽略不计. 面向服务软件各服务对资源的需求一方面与服务本身有关, 另一方面与服务的负载有关, 负载越高, 服务对资源的需求也就越大. 负载通常是指用户对软件的并发请求, 包括并发请求的数量和请求的事务规模. 为简单起见, 本文假设对同一个服务的请求具有相同的事务规模, 因此, 服务的负载可定义为单位时间内所有用户对组件的并发请求总数. 此外, 本文还假设服务对资源的需求与服务处理的请求数呈正比.

根据上述分析, 下面给出本文问题所涉及的一些概念的基本定义.

定义1 (面向服务软件)　面向服务软件可定义为一个四元组 SOS=$\{S, \mathrm{TR}, \mathrm{SR}, \mathrm{IR}\}$, 其中,

(1) $S = \{s_i | 1 \leqslant i \leqslant \mathrm{sn}\}$ 表示组成面向服务软件的服务集, 其中, s_i 表示第 i 个服务, sn 为服务的个数;

(2) $\mathrm{TR} : S \to \mathbb{R}$ 为服务的时间需求函数, 指定每个服务在处理单个请求时对具有单位计算能力的 CPU 资源的占用时间;

(3) $\mathrm{SR} : S \to \mathbb{R}$ 为服务的空间需求函数, 指定每个服务在处理单个请求时所需的内存空间;

(4) $\mathrm{IR} = \{N, E\}$ 表示服务集 S 中的服务之间的交互关系, 为一个树形结构 [22], 其中 N 为节点集, N 中的内部节点标记为交互关系, 叶子节点标记为服务, E 为边集, 反映了节点之间的关系. 图 3 给出了图 1 中面向服务软件服务之间交互关系的树形结构图. 其中, seq. 表示顺序关系, switch 表示选择关系, flow 表示并发关系, while 表示循环关系.

定义2 (组件负载)　组件负载可以用函数 $\mathrm{RA} : S \to \mathbb{N}$ 进行定义, 它指定单位时间内用户对各服务的并发请求数.

定义3 (云平台)　云平台可定义为一个四元组 CPF=$\{V, \mathrm{CP}, \mathrm{AS}, \mathrm{RC}\}$, 其中,

(1) $V = \{v_j | 1 \leqslant j \leqslant \mathrm{vn}\}$ 为云平台可以提供的虚拟机类型集, 其中, v_j 表示第 j 种虚拟机类型, vn

为虚拟机类型数;

(2) CP : $V \to \mathbb{R}$ 为计算能力函数, 指定每一类虚拟机的 CPU 计算能力;

(3) AS : $V \to \mathbb{R}$ 为可用空间函数, 指定每一类虚拟机的可用内存空间;

(4) RC : $V \to \mathbb{R}$ 为成本函数, 指定每一类虚拟机的一个实例的租用成本.

定义4 (部署方案) 云环境中面向服务软件的部署方案可以定义为一个三元组 CDS= $\{N, \mathrm{DV}, \mathrm{DS}\}$, 其中,

(1) $N = \{n_k | 1 \leqslant k \leqslant \mathrm{nn}\}$ 为部署方案所包含的部署节点集, 其中, n_k 表示第 k 个部署节点, nn 为部署节点数;

(2) DV : $N \times V \to \{0, 1\}$ 为虚拟机实例类型函数, 指定每个部署节点的虚拟机实例的类型, 其中, $\mathrm{DV}(n_k, v_j) = 1$ 表示部署节点 n_k 的虚拟机实例类型为 v_j, $\mathrm{DV}(n_k, v_j) = 0$ 表示部署节点 n_k 的虚拟机实例类型不为 v_j;

(3) DS : $N \times S \to \{0, 1\}$ 为服务实例类型函数, 其中, $\mathrm{DS}(n_k, s_i) = 1$ 表示部署节点 n_k 上部署了服务 s_i 的一个实例, $\mathrm{DS}(n_k, s_i) = 0$ 表示部署节点 n_k 上没有部署服务 s_i 的实例.

从上面的定义中可知, nn, $\mathrm{DV}(n_k, v_j)$ 和 $\mathrm{DS}(n_k, s_i)$ 的值决定了一个特定的部署方案, 本文的问题就转化为求解性能和成本最优时 nn, $\mathrm{DV}(n_k, v_j)$ 和 $\mathrm{DS}(n_k, s_i)$ 的值. 性能通常为多个指标的综合体现, 本文主要考虑 3 个最常见的性能指标, 即面向服务软件端到端的响应时间, 端到端的吞吐量以及虚拟机实例的最大利用率. 其中, 端到端的响应时间表示用户从调用面向服务软件到收到其返回结果所消耗的时间, 它直接影响着用户对面向服务软件的满意程度; 端到端的吞吐量表示面向服务软件在单位时间内处理的请求数, 它体现了面向服务软件的处理能力; 虚拟机实例的最大利用率是指部署所涉及的各虚拟机实例的 CPU 利用率的最大值, 它体现了面向服务软件遭遇瓶颈的可能性. 云环境中面向服务软件的运行成本通常也由多方面因素构成, 为简单起见, 本文主要考虑租用虚拟机实例所需支付的成本.

面向服务软件端到端的响应时间和吞吐量需要根据它所包含的服务之间的交互关系, 聚合各服务的响应时间和吞吐量来获得. 由于各服务的响应时间、吞吐量、各虚拟机实例的利用率以及响应时间和吞吐量的聚合函数不能使用简单的数学公式进行表示, 为了方便描述本文的目标函数, 本节假设 $R(s_i)$ 和 $T(s_i)$ 分别为服务 s_i 的响应时间和吞吐量, $U(n_k)$ 为部署节点 n_k 的虚拟机实例的 CPU 利用率 (为简单起见, 后文都将其简称为部署节点 n_k 的利用率), $\prod_R$ 和 $\prod_T$ 分别为响应时间和吞吐量的聚合函数, R 和 T 分别为面向服务软件端到端的响应时间和吞吐量; U 为部署方案中所有虚拟机实例的利用率的最大值; C 为所有硬件主机的总运行成本, 则本文的优化目标可以表示为

$$\min R = \prod_{i=1}^{\mathrm{sn}} R(s_i), \tag{1}$$

$$\max T = \prod_{i=1}^{\mathrm{sn}} T(s_i), \tag{2}$$

$$\min U = \max_{n_k \in N} U(n_k), \tag{3}$$

$$\min C = \sum_{t=1}^{\mathrm{vn}} \left(\mathrm{RC}\,(v_j) \times \sum_{k=1}^{\mathrm{nn}} \mathrm{DV}(n_k, v_j) \right), \tag{4}$$

其中, $R(s_i)$, $T(s_i)$ 和 $U(n_k)$ 均为 nn, $\mathrm{DV}(n_k, v_j)$ 和 $\mathrm{DS}(n_k, s_i)$ 的函数, 它们以及聚合函数 $\prod_R$ 和 $\prod_T$ 的计算方法将在 3.3 小节给出.

一个可行的部署方案通常需要满足一些特定的约束. 首先, 就软件部署本身而言, 存在一些特定的约束, 如每个服务必须至少部署一个实例, 每个服务实例必须只能部署在一个虚拟机实例上等; 其次, 从运行角度而言, 可行的部署方案需要保证部署在一个虚拟机实例上的服务实例对资源的需求不能大于其本身提供的资源, 以便软件能够正常运行; 最后, 由于业务、技术、可靠性和拿权等方面的需求, 还可能存在一些其他的约束, 本文主要考虑位置约束和同位约束两种. 其中, 位置约束主要用于约束各类型服务实例与各类型虚拟机实例之间的位置关系, 同位约束主要用于约束任意两种类型的服务实例之间的位置关系, 它们分别使用函数 $\mathrm{LC}: S\times V\to\{0,1\}$ 和 $\mathrm{CL}: S\times S\to\{-1,0,1\}$ 进行表示. $\mathrm{LC}(s_i,v_j)$=1 表示服务 s_i 的实例可以部署在 v_j 类型的虚拟机实例上; $\mathrm{LC}(s_i,v_j)=0$ 表示服务 s_i 的实例不能部署在 v_j 类型的虚拟机实例上; $\mathrm{CL}(s_i,s_j)=1$ 表示服务 s_i 的实例和服务 s_j 的实例必须被部署在同一个虚拟机实例上; $\mathrm{CL}(s_i,s_j)=-1$ 表示服务 s_i 的实例和服务 s_j 的实例不能被部署在同一个虚拟机实例上; $\mathrm{CL}(s_i,s_j)=0$ 表示服务 s_i 的实例和服务 s_j 之间不存在位置约束. 基于上述描述, 要想构建一个可行的部署方案, 变量 nn, $\mathrm{DV}(n_k,v_j)$ 和 $\mathrm{DS}(n_k,s_i)$ 的值必须满足以下条件:

$$\sum_{k=1}^{\mathrm{nn}}\mathrm{DS}(n_k,s_i)\geqslant 1,\quad \forall i=1,\ldots,\mathrm{sn}, \tag{5}$$

$$\mathrm{nn}\geqslant 1, \tag{6}$$

$$\sum_{i=1}^{\mathrm{sn}}\mathrm{DS}(n_k,s_i)\geqslant 1,\quad \forall k=1,\ldots,\mathrm{nn}, \tag{7}$$

$$\sum_{i=1}^{\mathrm{sn}}\mathrm{DS}(n_k,s_i)\times\mathrm{TR}(s_i)\times\mathrm{RA}(s_i)\leqslant\sum_{j=1}^{\mathrm{vn}}\mathrm{DV}(n_k,v_j)\times\mathrm{CP}(v_j),\quad \forall k=1,\ldots,\mathrm{nn}, \tag{8}$$

$$\sum_{i=1}^{\mathrm{sn}}\mathrm{DS}(n_k,s_i)\times\mathrm{SR}(s_i)\times\mathrm{RA}(s_i)\leqslant\sum_{j=1}^{\mathrm{vn}}\mathrm{DV}(n_k,v_j)\times\mathrm{AS}(v_j),\quad \forall k=1,\ldots,\mathrm{nn}, \tag{9}$$

$$\sum_{k=1}^{\mathrm{nn}}\mathrm{DV}(n_k,v_j)\times\mathrm{DS}(n_k,s_i)=0,\quad \text{if } \mathrm{LC}(s_i,v_j)=0, \tag{10}$$

$$\mathrm{DS}(n_k,s_i)=\mathrm{DS}(n_k,s_j),\quad \forall i=1,\ldots,\mathrm{sn},\quad \text{if } \mathrm{CL}(s_i,s_j)=1, \tag{11}$$

$$\mathrm{DS}(n_k,s_i)\neq\mathrm{DS}(n_k,s_j),\quad \forall i=1,\ldots,\mathrm{sn},\quad \text{if } \mathrm{CL}(s_i,s_j)=-1. \tag{12}$$

式 (5) 描述每个服务必须至少部署一个实例; 式 (6) 描述一个部署方案至少有一个部署节点; 式 (7) 描述每个虚拟机实例上至少部署了一个服务实例; 式 (8) 描述了部署在一个虚拟机实例上的服务实例运行所需的计算能力不能超过该虚拟机实例所拥有的; 式 (9) 描述了部署在一个虚拟机实例上的服务实例运行所需的内存空间不能超过该虚拟机实例可用的内存空间; 式 (10) 描述了位置约束; 式 (11) 和 (12) 描述了同位约束.

3.3 性能指标估算方法

本节主要介绍上一节目标函数中 $R(s_i)$, $T(s_i)$ 和 $U(n_k)$ 的计算方法以及聚合函数 $\prod_R$ 和 $\prod_T$ 的表示方法. 排队论是用于估算分布式系统性能最常见的一种方法, 且已经在大量的研究工作中得到了有效应用 [23, 24]. 本文利用排队论来估算各服务的响应时间、吞吐量和各硬件节点的利用率.

由于每个服务被实例化为多个实例进行部署, 因此, 其响应时间和吞吐量的值由该服务对应的服务实例的响应时间和吞吐量聚合而来. 假设部署节点 n_k 的虚拟机实例类型为 v_j (即 $\mathrm{DV}(n_k,v_j)=1$), 服务 s_i 有一个实例部署在部署节点 n_k 的虚拟机实例上, $\mathrm{RA}'(n_k,s_i)$ 为分配给该实例的负载, 也可以

理解为单位时间内到达该服务实例的请求数, 它可以根据服务 s_i 的负载 $\mathrm{RA}(s_i)$ 和负载均衡器所采用的负载均衡算法获得. 本文假设负载均衡器采用循环负载均衡算法, 即将一个服务的负载平均分配到该服务的各个实例上, 因此可得

$$\mathrm{RA}'(n_k, s_i) = \frac{\mathrm{RA}(s_i)}{\sum_{k=1}^{\mathrm{nn}} \mathrm{DS}(n_k, s_i)}, \tag{13}$$

其中, $\sum_{k=1}^{\mathrm{nn}} \mathrm{DS}(n_k, s_i)$ 用于计算服务 s_i 的实例数.

由于服务实例的负载为开放型负载, 当负载较大, 超出资源的承受能力时, 到达的服务请求将会逐渐堆积, 而无法得到及时处理. 因此, 本文只考虑负载处于资源承受能力之内的情况. 在这种情况下, 当系统处于平衡状态时, 服务实例的吞吐量等于服务实例的请求到达速率, 即

$$T'(n_k, s_i) = \mathrm{RA}'(n_k, s_i). \tag{14}$$

本文将每个虚拟机实例建模为一个 M/M/1 (Poisson 到达、负指数分布的服务时间、单服务中心) 队列, 根据排队理论, 服务实例的响应时间等于该实例所分配的资源对请求的处理率 (即单位时间内可以处理的请求数) 与该实例的请求到达率之间的差值的倒数. 据此可得, 部署节点 n_k 上服务 s_i 的实例的响应时间 $R'(n_k, s_i)$ 为

$$R'(n_k, s_i) = \frac{1}{\mu_{k,i} - \mathrm{RA}'(n_k, s_i)}, \tag{15}$$

其中, $\mu_{k,i}$ 为部署节点 n_k 上服务 s_i 的实例的平均请求处理率, 等于服务实例的可用资源量与它处理单个请求所需的资源量的比值, 即

$$\mu_{k,i} = \frac{\mathrm{CP}(v_j) \times (1 - U(n_k) + U'(n_k, s_i))}{\mathrm{TR}(s_i)}, \tag{16}$$

其中, $U(n_k)$ 为部署节点 n_k 的虚拟机实例的利用率; $U'(n_k, s_i)$ 为部署节点 n_k 上服务 s_i 的实例对该部署节点的虚拟机实例的利用率; $\mathrm{CP}(v_j)$ 为类型为 v_j 的虚拟机实例的 CPU 计算能力; $\mathrm{TR}(s_i)$ 为服务 s_i 在处理单个请求时, 对具有单位计算能力的 CPU 资源的时间需求; $\mathrm{CP}(v_j) \times (1 - U(n_k) + U'(n_k, s_i))$ 为部署节点 n_k 的虚拟机实例单位时间内可供服务 s_i 的实例所使用的资源量.

$U'(n_k, s_i)$ 的值可根据效用法则 [25] 进行计算, 即资源利用率等于吞吐量乘以服务时间:

$$U'(n_k, s_i) = \frac{\mathrm{RA}'(n_k, s_i) \times \mathrm{TR}(s_i)}{\mathrm{CP}(v_j)}. \tag{17}$$

根据式 (14) 和 (17) 可得部署节点 n_k 的虚拟机实例的 CPU 利用率 $U(n_k)$ 为

$$\begin{aligned} U(n_k) &= \sum_{s_i \in \{s_x \| \mathrm{DS}(n_k, s_i) = 1\}} U'(n_k, s_i) \\ &= \sum_{s_i \in \{s_x \| \mathrm{DS}(n_k, s_i) = 1\}} \frac{\mathrm{RA}'(n_k, s_i) \times \mathrm{TR}(s_i)}{\mathrm{CP}(v_j)}. \end{aligned} \tag{18}$$

根据式 (15)∼(17) 可得部署节点 n_k 上服务 s_i 的响应时间

$$R'(n_k, s_i) = \frac{\mathrm{TR}(s_i)}{\mathrm{CP}(v_j) \times (1 - U_{n_k})}. \tag{19}$$

本文以图 4 给出的简单示例详细说明如何计算各服务实例的响应时间、吞吐量以及各虚拟机实例的利用率. 该图给出了服务 s_1 和 s_2 分别有一个实例在部署节点 n_1 上运行的一种场景, 其中, 服务

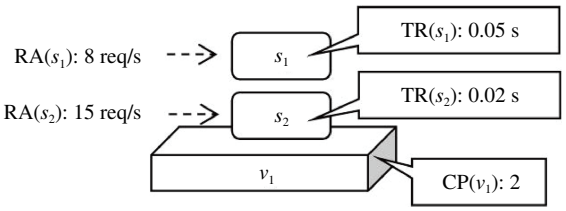

图 4　服务运行场景示例

Figure 4　An example for running scenario of services

s_1 和 s_2 在处理单个请求时, 对具有单位计算能力的 CPU 资源的时间需求 TR(s_1) 和 TR(s_2) 分别为 0.05 s 和 0.02 s, 部署节点的虚拟机实例类型为 v_1, 其计算能力 CP(v_1) 为 2. 在该场景中 (见图 4), 服务 s_1 和 s_2 的请求到达率 RA(s_1) 和 RA(s_2) 分别为 8 req/s 和 15 req/s, 根据式 (18) 可得部署节点 n_1 的虚拟机实例的利用率 $U(n_1) = 0.05 \div 2 \times 8 + 0.02 \div 2 \times 15 = 35\%$; 根据式 (14) 可得, 这两个服务实例的吞吐量 $T'(n_1, s_1)$ 和 $T'(n_1, s_2)$ 等于它们的请求到达率, 即分别为 8 req/s 和 15 req/s; 根据式 (19) 可得部署节点 n_1 上服务 s_1 的实例的响应时间 $R'(n_1, s_1) = 0.05 \div 2 \div (1 - 0.35) \approx 0.038$, 部署节点 n_1 上服务 s_2 的实例的响应时间 $R'(n_1, s_2) = 0.02 \div 2 \div (1 - 0.35) \approx 0.015$.

得到了各服务实例的响应时间和吞吐量, 就可以聚合得到对应服务的响应时间和吞吐量, 它们可以表示为

$$R(s_i) = \sum_{k=1}^{\mathrm{nn}} \left(\frac{\mathrm{RA}'(n_k, s_i)}{\mathrm{RA}(s_i)} \times R'(n_k, s_i) \right), \tag{20}$$

$$T(s_i) = \sum_{k=1}^{\mathrm{nn}} T'(n_k, s_i). \tag{21}$$

得到了各服务的响应时间和利用率, 则可以根据前面定义的服务交互关系的树型结构, 采用聚合函数 $\prod_R$ 和 $\prod_T$ 计算面向服务软件端到端的响应时间和吞吐量. 聚合函数 $\prod_R$ 和 $\prod_T$ 可根据前面定义的树形结构, 分别用递归函数 RF(w) 和 TF(w) 表示, 即

$$\mathrm{RF}(w) = \begin{cases} \max\limits_{w' \in \mathrm{CN}(w)} \mathrm{RF}(w'), & w \in \overline{F}, \\ \sum\limits_{w' \in \mathrm{CN}(w)} \left(\mathrm{EV}(w') \times \mathrm{RF}(w') \right), & w \notin \overline{F}, \end{cases} \tag{22}$$

$$\mathrm{TF}(w) = \begin{cases} \max\limits_{w' \in \mathrm{CN}(w)} \mathrm{TF}(w'), & w \in \overline{\mathrm{SW}}, \\ \sum\limits_{w' \in \mathrm{CN}(w)} \mathrm{TF}(w'), & w \notin \overline{\mathrm{SW}}, \end{cases} \tag{23}$$

其中, w 表示关系树中的节点, 当 w 为叶子节点 (即为表示服务实例的节点) 时, RF(w) 和 TF(w) 分别为 w 表示的服务的响应时间 $R(w)$ 和吞吐量 $T(w)$; 当 w 为根节点时, RF(w) 和 TF(w) 分别表示面向服务软件端到端的响应时间 R 和吞吐量 T; CN(w) 为节点 w 的子节点集; EV(w) 为结构树中以 w 为终节点的边的值; $\overline{F}$ 为结构树中标记为 "flow" 的节点集; $\overline{\mathrm{SW}}$ 为结构树中标记为 "switch" 的节点集.

4　求解算法

本文采用一种改进的多目标遗传算法 MGA-DO (multi-objective genetic algorithm for deployment optimization) 求解云环境中面向服务软件部署优化问题. 该算法针对面向服务软件的部署特征, 设计

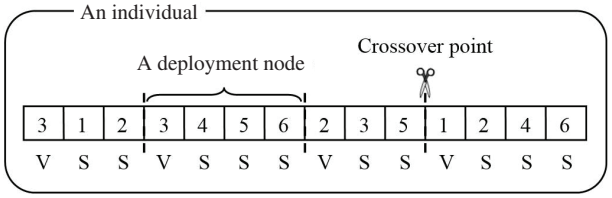

图 5 一个部署方案编码示例

Figure 5 An example for the encoding of deployment

了一种新的编码方案以及对应的遗传操作, 同时将遗传算法与现有的部署优化经验相结合, 设计了 5 种局部搜索规则, 通过提高算法的局部搜索能力提高算法的求解效率.

4.1 编码方案

本文把一个虚拟机实例和部署在该虚拟机实例上的服务实例称为一个部署节点, 一个部署方案由多个部署节点组成. 鉴于一个服务实例只能部署在一个虚拟机实例上, 而一个虚拟机实例可以部署多个不同服务的实例, MGA-DO 算法按照部署节点对部署方案进行分组编码, 每个部署节点为一个基因序列, 该基因序列的第一个基因表示该部署节点的虚拟机实例的类型, 其他基因表示该部署节点上的服务实例对应的服务编号. 图 5 显示了云环境中面向服务软件的一个部署方案的编码示例, 该部署方案包含 4 个部署节点, 其中第 2 个部署节点 3456 表示编号为 4, 5, 6 的服务分别有一个实例被部署在类型为 v_3 的一个虚拟机实例上.

4.2 适应度函数

MGA-DO 算法借鉴了 NSGA-II [26] 算法和 E^3-R [21] 算法中关于适应度函数的相关定义, 设计了一种新的适应度值的计算方法. 下面先给出一些相关概念, 然后介绍如何计算种群中个体的适应度值.

定义5 (支配) 假设存在两个个体 p_i 和 p_j, 当 p_i 的所有目标值都优于或等于 p_j, 且至少存在一个目标值优于 p_j 时, 称 p_i 支配 p_j.

定义6 (支配等级) 假设存在一个种群 P, 对于个体 $p_i \in P$, 如果 p_i 不被 P 中的其他任何个体支配, 则 p_i 的支配等级 $\mathrm{DR}(p_i) = 1$, 为最高支配等级; 其他个体的支配等级为种群中支配该个体的所有个体中, 支配等级最低的个体的支配等级加 1, 即, 如果种群 P 中支配个体 p_i 的所有个体中, 支配等级最低的个体的支配等级为 j, 则 p_i 的支配等级为 $j+1$.

定义7 (支配值) 假设存在一个种群 P, 对个体 $p_i \in P$, p_i 的支配值 $\mathrm{DV}(p_i)$ 为该种群中低于或等于该个体支配等级的个体 (包括该个体) 的总数.

图 6 显示了支配等级和支配值的一个例子. 由于目标维数过多时, 无法用图直观的表示, 因此本示例只选择吞吐量和响应时间两维目标. 本示例显示了 9 个个体的支配关系, 其中个体 p_1, p_2, p_4 的支配等级为 1, 因为没有任何其他个体支配它们, 它们的支配值为种群中的个体总数 9; 个体 p_3, p_5, p_6 的支配等级为 2, 因为只有等级为 1 的个体支配它们, 它们的支配值为种群数减去支配等级为 1 的个体数, 即 $9-3=6$; 类似地, 个体 p_1, p_2, p_4 的支配等级为 3, 支配值为种群数减去支配等级为 1 的个体数和支配等级为 2 的个体数, 即 $9-3-3=3$.

当两个个体具有不同的支配等级时, 更倾向保留支配等级较高 (DR 值较小) 的个体; 当两个个体具有相同的支配等级时, 更倾向于保留拥挤距离较高的个体 [26].

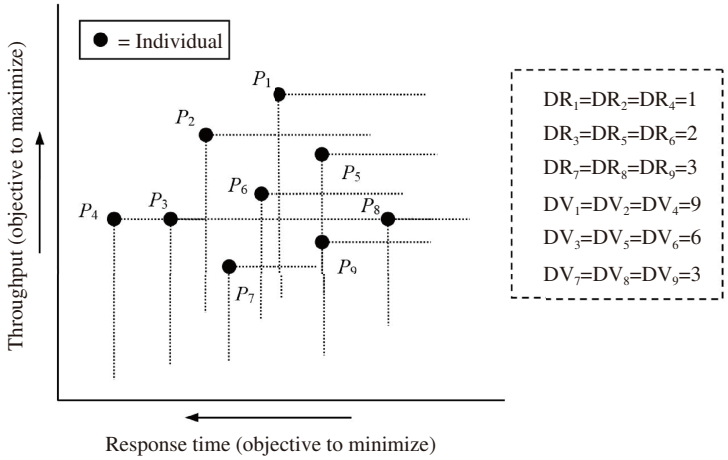

图 6　支配排序示例

Figure 6　An example for domination ranking

定义8 (非支配集)　种群 P 中支配等级相同的个体组成的集合称为一个非支配集.

定义9 (拥挤距离)[26]　种群 P 中个体 p_i 的拥挤距离为该个体的非支配集中, 在每个目标上与该个体相邻的两个个体到该个体的平均距离之和.

定义10 (稀疏值)　种群 P 中个体 p_i 的稀疏值 $\mathrm{SV}(p_i)$ 为该个体所在的非支配集中, 拥挤距离小于该个体拥挤距离的个体数.

MGA-DO 算法定义个体 p_i 的适应度为该个体的支配值与其稀疏值之和, 即

$$\mathrm{FV}(p_i) = \mathrm{DV}(p_i) + \mathrm{SV}(p_i). \tag{24}$$

4.3　选择与交叉算子

(1) 选择算子. MGA-DO 使用二进制锦标赛法进行父类个体的选择. 该方法首先从当前种群中随机选择两个个体, 然后对比这两个个体的适应度值, 将适应度值大的个体选作父类个体. 算法中的所有父类个体都使用该方法从当前种群中获得.

(2) 交叉算子. 由于本文采用的编码方式为可变长度的编码方式, 且编码相对复杂, 为了确保交叉操作尽可能地产生可行个体, MGA-DO 对交叉操作进行了限制. 首先, MGA-DO 根据部署节点来划分交叉点 (crossover point) (如图 5 中的虚线位置), 即交叉操作是以部署节点作为基本的操作单元; 其次, MGA-DO 的交叉算子采用单点交叉方式, 在用于执行交叉操作的两个父类个体的每个个体中分别随机选取一个交叉点 (即两个交叉点的位置可以不同), 然后对交叉点前或后的部分基因进行交换. 图 7 显示了交叉算子的一个例子.

从图 7 可以看出, 执行交叉算子时, 可能会导致新产生的部署方案不包含某一服务的任何实例 (如图 7 中执行交叉算子产生的子个体 A′ 中不存在服务 s_3 的任何实例), 从而使得新生成的部署方案不可行. 因此, 需要在交叉算子执行完以后对生成的不可行的部署方案进行修正. 采取的修正措施是, 随机选取一个部署节点, 并在该部署节点上添加服务 s_3 的一个实例 (如图 8 所示).

4.4　局部搜索规则

遗传算法通常采用随机变异操作进行局部搜索, 搜索能力差, 容易产生 “早熟” 收敛问题 [26]. 目

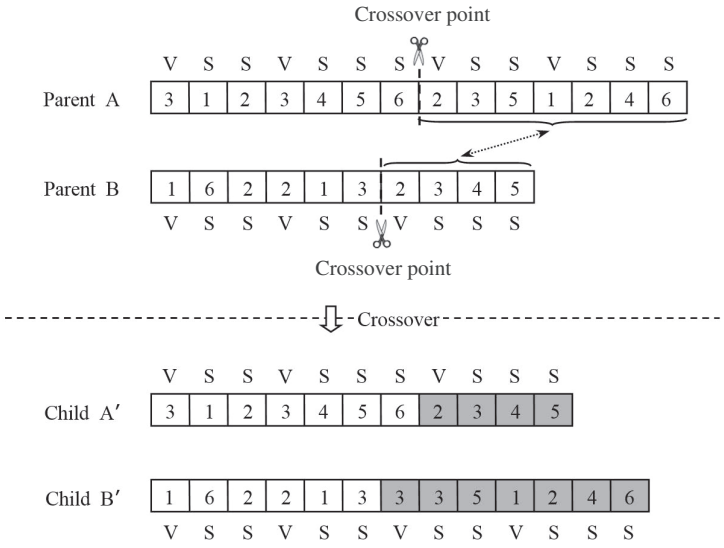

图 7 交叉算子示例

Figure 7 An example for crossover operator

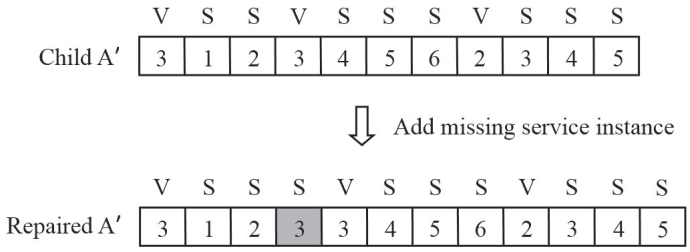

图 8 修复不可行的部署方案

Figure 8 Repairing infeasible deployments

前, 在部署优化领域, 已有一些经验可用于优化软件的性能和成本. 因此, MGA-DO 算法结合这些优化经验, 针对不同的优化目标, 设计了多种局部搜索规则, 以提高算法的局部搜索能力. 本文在 3.3 小节指出, 只有当存在部署节点过载的情况时, 软件的吞吐量才会发生变化, 否则, 其吞吐量将等于软件当前的请求达到速率, 这表明吞吐量的优化可以转化为对最大利用率的优化. 因此, 本文只针对响应时间、最大利用率和成本这 3 个目标制定相应的搜索规则.

(1) 优化响应时间. 软件的响应时间可以通过为其分配更多的计算资源来进行优化, 因此, 针对这一目标设计的搜索规则如下:

(i) 租用一个新的虚拟机实例, 用于替换当前部署方案下, 计算能力低于该新虚拟机实例计算能力的部署节点的虚拟机实例. 假设 n_k 为当前部署方案下的一个部署节点, 其虚拟机实例类型为 v_j, v_y 为可租用的一个虚拟机类型, $\mathrm{CP}(v_j)$ 和 $\mathrm{CP}(v_y)$ 分别为 v_j 和 v_y 类型的虚拟机实例的计算能力; 如果 $\mathrm{CP}(v_j) < \mathrm{CP}(v_y)$, 则租用一个 v_y 类型的虚拟机实例, 并将部署节点 n_k 上的所有服务实例都迁移到该虚拟机实例上, 然后释放空闲虚拟机实例.

(ii) 租用一个新的虚拟机实例, 并从当前部署方案下的部署节点上迁移部分服务实例到该虚拟机实例上, 或在该虚拟机实例上部署新的服务实例, 以形成新的部署节点.

(2) 优化最大利用率. 虚拟机实例的利用率主要与负载有关, 降低最大利用率可以通过降低具有最高利用率的虚拟机实例上的负载来实现. 因此, 针对这一目标设计的搜索规则如下:

(i) 将利用率最高的 (源) 部署节点上一个服务实例迁移到利用率最低的 (目的) 部署节点上, 并保证迁移后目的部署节点的利用率不高于迁移前源部署节点的利用率. 假设 n_k 和 n_t 为当前部署方案下利用率最高和最低的两个部署节点, 它们的虚拟机实例类型分别为 v_j 和 v_y, 利用率分别为 $U(n_k)$ 和 $U(n_t)$, s_i 为部署节点 n_k 上一个服务实例的类型, $U'(n_k, s_i)$ 为服务 s_i 的实例对部署节点 n_k 的虚拟机实例的利用率; 如果 $U(n_k) > U(n_t) + U'(n_k, s_i) \times \mathrm{CP}(v_j) \div \mathrm{CP}(v_y)$, 则将部署节点 n_k 上服务 s_i 的实例迁移到部署节点 n_t 上.

(ii) 交换利用率最高的和最低的两个部署节点上的两个服务实例的位置, 要求用于交换的利用率最高的部署节点上的服务实例对资源的占用量要高于利用率最低的部署节点上的服务实例对资源的占用量, 且保证交换后的最大利用率不高于交换前的最大利用率. 假设 n_k 和 n_t 为当前部署方案下利用率最高和最低的两个部署节点, 它们的虚拟机实例类型分别为 v_j 和 v_y, 利用率分别为 $U(n_k)$ 和 $U(n_t)$, s_i 和 s_m 分别为部署节点 n_k 和 n_t 上的一个服务实例的类型, $U'(n_k, s_i)$ 和 $U'(n_t, s_m)$ 分别为 s_i 和 s_m 的实例对其所在部署节点的虚拟机实例的利用率, 如果 $U'(n_k, s_i) \times \mathrm{CP}(v_j) > U'(n_t, s_m) \times \mathrm{CP}(v_y)$, $U(n_k) > U(n_t) + U'(n_k, s_i) \times \mathrm{CP}(v_j) \div \mathrm{CP}(v_y)$, 则将部署节点 n_k 上服务 s_i 的实例迁移到部署节点 n_t 上, 并将部署节点 n_t 上服务 s_m 的实例迁移到部署节点 n_k 上.

(iii) 在利用率最低的部署节点上增加一个新的服务实例, 要求该服务实例所属的服务类型与利用率较高的部署节点上占用资源较多的服务实例所属的服务类型相同, 且保证增加后的最大利用率不高于增加前的最大利用率. 假设 n_k 和 n_t 为当前部署方案下利用率最高和最低的两个部署节点, 它们的虚拟机实例类型分别为 v_j 和 v_y, 利用率分别为 $U(n_k)$ 和 $U(n_t)$, s_i 为部署节点 n_k 上占用资源最多的服务实例的类型, 如果 $U(n_k) > U(n_t) + U'(n_k, s_i) \times \mathrm{CP}(v_j) \times (1 - 1 \div (\sum_{k=1}^{\mathrm{nn}} \mathrm{DS}(n_k, s_i) + 1))$, 则创建服务 s_i 的一个新实例, 并将其部署在部署节点 n_t 上.

(iv) 从利用率最高的部署节点上删除一个服务实例, 要求删除的服务实例所属的服务有实例部署在其他部署节点上, 且保证删除后的最大利用率不高于当前的最大利用率.

(v) 租用一个新的虚拟机实例, 用于替换利用率最高的部署节点的虚拟机实例, 要求新虚拟机实例比利用率最高的部署节点的虚拟机实例拥有更多的计算资源. 假设 n_k 为当前部署方案下利用率最高的部署节点, 其虚拟机实例类型为 v_j, v_y 为可租用的虚拟机类型, $\mathrm{CP}(v_t)$ 和 $\mathrm{CP}(v_y)$ 分别为 v_t 和 v_y 类型的虚拟机实例的计算能力; 如果 $\mathrm{CP}(v_t) < \mathrm{CP}(v_y)$, 则租用 v_y 类型的一个虚拟机实例, 并将部署节点 n_k 上的所有服务实例都迁移到该虚拟机实例上, 并释放空闲虚拟机实例.

(vi) 租用一个新的虚拟机实例, 并从利用率最高的部署节点开始, 将部分服务实例迁移到该虚拟机实例上, 或在该虚拟机实例上部署新的服务实例, 且要求新服务实例所属的服务类型与利用率最高的虚拟机上的服务实例所属的服务类型一致, 且保证操作后新虚拟机实例的利用率不大于操作前的最大利用率.

(3) 优化成本. 本文的部署成本为部署所需的虚拟机实例的租用成本, 可以通过租用更廉价的虚拟机实例或减少租用的虚拟机实例的数量来降低软件的部署成本. 因此, 针对这一目标设计的搜索规则如下:

(i) 租用一个新的虚拟机实例, 用于替换利用率最低的部署节点的虚拟机实例, 要求新虚拟机实例的价格低于利用率最低的部署节点的虚拟机实例的价格, 且新虚拟机实例拥有足够的资源放置利用率最低的部署节点上的所有服务实例.

(ii) 将当前利用率最低的部署节点上的所有服务实例迁移到其他部署节点上, 并释放该部署节点,

算法 1 演化过程

$P^0 \leftarrow$ A population of popsize individuals generated by random;
AssignFitnessValue(P^0);
$Q^0 \leftarrow$ A new population generated by applying local search on P^0;
AssignFitnessValue(Q^0);
$P^1 \leftarrow$ Top pn of $P^0 \cup Q^0$ in terms of their fitness values;
$g \leftarrow 1$
while $g < g_{\max}$ **do**
  $Q^g \leftarrow \emptyset$;
  while $|Q^g| \leqslant$ Popsize **do**
    $p_a, p_b \leftarrow$ Randomly selected two individuals from P^g;
    $P_\alpha \leftarrow$ Either p_a or p_b with a higher fitness value;
    $p_a, p_b \leftarrow$ Randomly selected two individuals from P^g;
    $p_\beta \leftarrow$ Either p_a or p_b with a higher fitness value;
    $r \leftarrow$ A random generated number distributed uniformly in [0,1];
    if $p_c \geqslant r$ **then**
      $q_\alpha, q_\beta \leftarrow$ Crossover(p_α, p_β);
    else
      $q_\alpha, q_\beta \leftarrow P_\alpha, P_\beta$;
    end if
    Add q_α to Q^g if Q^g does not contain q_α;
    Add q_β to Q^g if Q^g does not contain q_β;
  end while
  AssignFitnessValue(Q^g);
  $M^g \leftarrow$ A new population generated by applying local search on Q^g;
  AssignFitnessValue(M^g);
  $P^{g+1} \leftarrow$ Top pn of $P^g \cup Q^g \cup M^g$ in terms of their fitness values;
  $g \leftarrow g + 1$;
end while

要求其他部署节点拥有足够的资源能够放置这些服务实例.

在种群中存在两类个体: 可行个体和不可行个体, 针对不同的个体, 采用不同的方式应用局部搜索规则. 对于可行个体, 首先随机选择一个优化目标, 然后在针对该目标的规则中随机选择一个规则进行执行; 而对于不可行个体, 则根据导致其不可行的目标, 选择相应的规则进行执行.

4.5 算法描述

算法 1 给出了 MGA-DO 算法的优化过程. 其中, P^g 为第 g 代的种群, Q^g 为第 g 代生成的子个体群, p_c 为交叉率, pn 为种群规模. 在算法开始时, MGA-DO 采用随机生成的方式生成初始种群, 即首先根据成本约束和性能约束得到可行部署方案的部署节点范围; 然后随机生成个体的部署节点数 nn, 对于每个部署节点, 每个服务以 0.5 的概率在该部署节点上部署一个实例; 最后检查生成的部署方案, 如果存在没有实例被部署的服务, 则在现有的部署节点中, 随机选择一个部署节点, 并在该部署节点上部署该服务的一个实例. 在后续的繁殖过程中, 种群 P^g 在每一代繁殖出 N 个不同的子个体, 形成子个体群 Q^g, 然后对子个体群 Q^g 应用局部搜索策略, 形成新的种群 M^g, 最后从种群 P^g, Q^g 和 M^g 的合集中选出 pn 个适应度值最高的个体作为下一代的种群 P^{g+1}. 生成子个体群 Q^g 的具体过程如下: 首先, 使用二进制锦标赛法选择从种群 P^g 中选取两个父个体 p_α 和 p_β; 其次, 以概率 p_c 对 p_α 和 p_β

表 1　虚拟机参数设置

Table 1　Parameter setting of VMs

VM type	CP	AS	VC
High	4	30	50
Mid	2.4	20	30
Low	1.5	10	10

执行交叉操作繁殖出两个子个体 Q_α 和 Q_β; 最后, 重复上述操作, 直到产生 N 个不同的子个体.

5　实验分析

5.1　实验数据与环境

本实验首先通过搜索空间分析算法显示了所求解问题的困难程度; 然后利用 MGA-DO 算法与蛮力搜索算法对 5 个规模较小的实验案例进行求解, 并从处理时间和搜索空间两个方面进行对比分析, 以证明 MGA-DO 算法的有效性; 最后设计了 3 种不同规模与初始设置的模拟案例 (记为 Case1, Case2 和 Case3), 利用所提出的 MGA-DO 算法与最近提出的 E^3-R 算法 [21] 及 NSGA-II [26] 算法对其进行求解, 并从搜索空间与有效性、目标值、处理时间和 hypervolume 4 个方面对比这 3 种算法, 以证明 MGA-DO 算法的优越性. 用于对比 MGA-DO, E^3-R 和 NSGA-II算法的 3 个模拟案例的服务数分别为 8, 12 和 15, 部署这些服务所能接受的最大成本分别为 \$500, \$990 和 \$2500; 部署可租用的虚拟机分高、中、低 3 类, 各虚拟机实例的利用率上限值为 0.8. 表 1 给出了各种类型的虚拟机的相关属性参数.

实验中与服务相关的参数同样由指定范围内随机生成的数据进行填充, 各参数的范围如下: $\mathrm{TR} \in (1, 20]$, $\mathrm{SR} \in [2, 8]$, $\mathrm{RA} \in [0.01, 0.1]$; 服务之间的交互关系根据概率随机指定, 指定为顺序、选择、并发和不交互这 4 种交互关系的概率分别为 0.25; 其他一些参数值在以上这些参数值确定后再确定.

MGA-DO 算法中的需要确定的参数主要包括种群规模 pn、交叉率 p_c 和变异率 p_m, 这些参数值的设定会对算法的性能和质量造成较大影响, 因此, 需要对这些参数进行分析, 以确定其最佳的取值. 对于种群规模而言, 通常设置的值越大, 种群的多样性越高, 也就越有利于找到最优的解, 然而, 较大的种群规模也会导致算法的运行时间较长, 大大降低了算法的性能. 为此, 本实验根据文献 [19, 21] 的实验结果, 设置 MGA-DO 算法的种群规模 $\mathrm{pn} = 100$.

交叉算子作为遗传算法的主要算子, 发挥着全局搜索的作用. 交叉概率的设置, 决定了算法的全局搜索能力. 通常来说, 较高的交叉概率会使算法具有较强的全局搜索能力, 能够有效降低算法陷入局部最优解的概率, 但同时也可能导致因搜索过多的不必要解空间而耗费大量的计算时间. 为了探索交叉率的最优取值, 本实验在 Case1, Case2 和 Case3 上进行了实验研究. 实验过程中, 交叉率的取值范围为 [0.1,1], 并以 0.1 为增量. 由于从这些案例得到的解集的 hypervolume 值不在相同的数量级上, 因此, 本实验根据文献 [27], 将它们按照以下公式转化为 R-values:

$$r_{i,j}^k = \frac{H_{i,j}^k}{10 \times G_{i,j}(\max_k(H_{i,j}^k))}, \tag{25}$$

其中, $H_{i,j}^k$ 为算法 MGA-DO 运行在案例 i 上的第 j 个参数取第 k 个值时, 所得解集的 hypervolume 值, 其中, i, j, k 为整数, i 的取值范围为 [1,3], j 的取值为 1, k 的取值范围为 [1,10]; 为 i, j 取固定值且 k

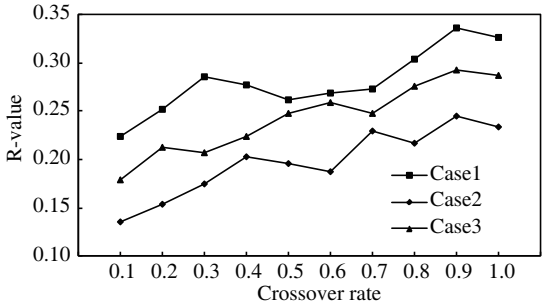

图 9 在 Case1, Case2 和 Case3 上对交叉概率 p_c 的分析结果

Figure 9 Analysis of the configuration for crossover rate on Case1, Case2, Case3

从 1 变化到 10 时, 所得的 10 个解集的 hypervolume 值中的最大值; $G_{i,j}(\max_k(H^k_{i,j}))$ 表示 $\max_k(H^k_{i,j})$ 的数量级, 如 0.02 的数量级为 0.01.

本实验针对交叉率的每个取值, 在每个案例上分别独立运行 30 次, 并将得到的解集的 hypervolume 值按照 (25) 式进行转换, 得到如图 9 所示的结果. 从该图中可以看出当 $p_c = 0.9$ 时, 算法 MGA-DO 能够在 Case1, Case2 和 Case3 上得到最大的 R-values.

实验的硬件平台环境为: CPU Inter Core(TM) 2 Duo 3.16 GHz, 内存 DDR3 4 GB, 硬盘 500 GB; 软件平台为: 操作系统 64bit Windows 7, 算法实现工具 MyEclipse 8.5, 编程语言为 Java.

5.2 搜索空间分析和有效性分析

对于优化问题来说, 问题的搜索空间是影响问题求解效率的一个重要因素, 因此, 有必要对其进行分析. 假设一个面向服务软件包含 sn 个服务; 云平台提供了 vn 种类型的虚拟机, 其中, v_t 类型的虚拟机实例的租用价格为 $\mathrm{RC}(v_t)$; 服务供应商可花费的最大成本为 $C_{\max}$; 则一个部署方案所包含的虚拟机实例的最大数为

$$\mathrm{vn}_{\max} = \frac{C_{\max}}{\min_{v_t \in V}(\mathrm{RC}(v_t))}. \tag{26}$$

云环境中面向服务软件的部署空间分析问题实际上是一个典型的排列组合问题. 根据前面的定义可知, 一个部署方案由 nn 个部署节点组成, 每个部署节点由一个虚拟机实例和一个或多个服务实例组成; 对于每个部署节点, 每个服务都有两种选择, 即在该部署节点上部署一个服务实例或不在该部署节点上部署服务实例; 同时, 每个部署节点上必须至少部署一个服务实例. 因此, 部署节点的类型数为 $\mathrm{vn} \times (2^{\mathrm{sn}} - 1)$. 由于一个部署方案中可能存在多个类型相同的部署节点, 即, 具有相同虚拟机实例类型和相同服务实例类型, 该问题可以转化成典型的放球问题. 即, 把 n 个球放到 m 个盒子里, n 个球无区别, m 个盒子有区别, 且允许有空盒, 求可能的方案数. 这里, 部署节点看作球, 节点类型看作盒子, 则由 nn 个部署节点组成的部署方案的数量为 $(\mathrm{nn} + \mathrm{vn} \times (2^{\mathrm{sn}} - 1) - 1)!/(\mathrm{nn}! \times (\mathrm{vn} \times (2^{\mathrm{sn}} - 1) - 1)!)$. 由于 nn 的取值范围为 $[1, \mathrm{vn}_{\max}]$ 中的整数, 因此, 上述面向服务软件的搜索空间为

$$\sum_{n=1}^{\mathrm{vn}_{\max}} (\mathrm{nn} + \mathrm{vn} \times (2^{\mathrm{sn}} - 1) - 1)!/(\mathrm{nn}! \times (\mathrm{vn} \times (2^{\mathrm{sn}} - 1) - 1)!). \tag{27}$$

假设在一个部署优化问题中, 面向服务软件包含 10 个服务, 云平台包含 6 种类型的虚拟机, 其最低的租用价格为 \$10, 部署面向服务软件可花费的总成本不能超过 \$200, 即 sn=10, vn=6, $\mathrm{vn}_{\max} = \frac{C\max}{\min_{v_t \in V}(\mathrm{RC}(v_t))} = \frac{200}{10} = 20$, 则该优化问题的搜索空间为 8.09×10^{41}, 由此可见, 即使对于一个中等规

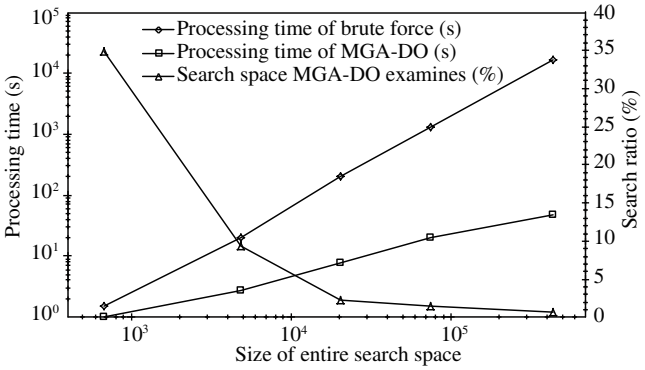

图 10　检查的搜索空间比率与处理时间

Figure 10　Ratio of search space examined and processing time

模的面向服务软件, 其部署优化问题的搜索空间也是非常巨大的, 搜索整个空间需要耗费大量的时间和精力.

部署优化问题是一个典型的 NP 难题, 蛮力搜索是唯一能够保证找到真正最优解的一种求解方法. 然而, 由于蛮力搜索需要搜索整个解空间, 因此只适用于求解规模较小的问题. MGA-DO 旨在通过搜索部分解空间, 快速寻找问题的近似最优解. 因此, 能够适用于求解大规模问题. 为证明 MGA-DO 算法的有效性, 本实验设置了 5 个不同规模的模拟案例 (它们的搜索空间在 679 与 4.36×10^5 之间逐渐递增), 并分别利用蛮力搜索和 MGA-DO 对它们进行求解, 以获得它们的真正最优解. 图 10 对比了这两种算法求解这些问题时的搜索空间和求解时间. 由于蛮力搜索方法需要搜索整个问题空间, 因此, 图 10 中没有给出蛮力搜索方法的搜索空间, 而只给出 MGA-DO 的搜索空间占整个搜索空间 (也即蛮力搜索方法的搜索空间) 的比率.

从图 10 可以看出, MGA-DO 寻找最优部署方案所检查的搜索空间占整个搜索空间的比率非常小, 且问题规模越大, 比率越小. 例如, 在规模为 679 的问题中, 由 MGA-DO 检查的搜索空间占整个搜索空间的比率为 35.6%, 而在规模为 4.36×10^5 的问题中, 该比率为 0.7%. MGA-DO 在求解优化问题时只需检查一小部分搜索空间的优势, 使得其处理时间远远小于蛮力搜索, 即使求解大规模的优化问题也能在较短的时间内找到最优的部署方案. 例如, 在问题规模为 4.36×10^5 的问题中, MGA-DO 平均只用了 47 s 就找到了最优的部署方案, 而蛮力搜索得花费 4.59 个小时, 该结果表明 MGA-DO 能够有效地找到最优部署方案.

5.3　目标值分析

本节主要根据目标函数值对比 MGA-DO 算法与其他算法的性能. 表 2 显示了每个算法在 3 个不同规模的模拟案例上独立运行 30 次得到的解集的所有目标值的平均值和标准方差. 在每次运行过程中, 所有的对比算法在 Case1 上执行 30000 次评估, 在 Case2 上执行 40000 次评估, 在 Case 3 上执行 50000 次评估. 此外, 为了使对比更加全面, 本实验也记录了不使用本文提出的局部搜索, 而使用随机变异操作进行局部搜索的算法 MGA-DO/S 在 3 种不同案例上的实验结果.

根据结果可以看出, 在所有的 3 种规模的实验案例中, MGA-DO 算法都能够为各目标函数找到更优的值, 且即使不采用优化策略, 提出的 MGA-DO 算法仍然具有较好的性能. 目标函数值的标准方差 (STD) 显示了解集分布的集中程度, 它的值越小, 说明解集越集中, 表示算法的求解性能越稳定. 从表 2 中的数据可以看出, MGA-DO 算法的解集具有较小的标准方差, 这说明该算法得到的解集更稳定.

表 2 各算法在 3 个案例上运行所得到的目标值的平均值和方差

Table 2 Means and standard deviation of objective values obtained by different algorithms on 3 cases

Case	Algorithm	Response time (ms)	Max utility (%)	Cost ($)
Case1	MGA-DO	82.60 (3.351)	12.05 (0.319)	218.33 (3.16)
	MGA-DO/S	83.23 (3.823)	12.84 (0.491)	230.15 (3.67)
	E^3-R	83.41 (4.218)	13.19 (0.472)	237.07 (4.43)
	NSGA-II	86.51 (4.346)	14.42 (0.505)	246.24 (3.73)
Case2	MGA-DO	141.63 (4.073)	17.06 (0.364)	489.67 (5.53)
	MGA-DO/S	148.01 (4.579	20.28 (0.437)	536.79 (6.08)
	E^3-R	151.33 (4.861)	18.81 (0.475)	497.94 (6.72)
	NSGA-II	157.54 (4.326)	21.61 (0.532)	551.36 (7.09)
Case3	MGA-DO	344.49 (6.076)	18.87 (0.421)	1670.25 (7.25)
	MGA-DO/S	349.18 (6.711)	20.11 (0.506)	1743.72 (8.04)
	E^3-R	371.85 (7.327)	21.92 (0.479)	1767.59 (8.96)
	NSGA-II	385.02 (7.794)	23.56 (0.543)	1825.36 (9.37)

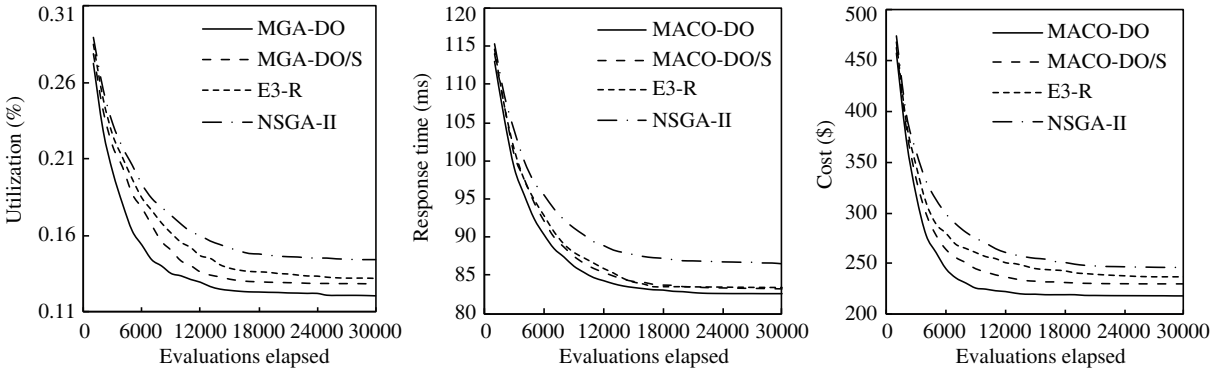

图 11 各算法在 Case1 上运行的演化曲线图

Figure 11 Evolutionary curves of different algorithms on Case1

为了进一步分析算法的收敛行为, 本节还给出了这些算法的演化曲线 (见图 11~13). 演化曲线上的值为 30 次独立运行过程中, 每隔 1000 次评估所记录的对应目标函数的最优值的平均值.

从图 11~13 可以看出, 在 Case1 和 Case2 上执行 1000 次评估以及在 Case3 上执行 2000 次评估之后, 所有的算法都能够找到可行的解, 这说明使用智能算法能够比使用随机搜索更快地找到可行解. 在找到可行解后, 这些可行解将继续演化以展现更广范围的权衡, 即更短的响应时间、更低的最大利用率和更低的成本. 从上述图中可以看出, 与 MGA-DO/S, E^3-R 以及 NSGA-II相比, MGA-DO 在整个优化过程中, 始终能够找出具有更广泛的权衡的部署方案. 即便是 MGA-DO/S, 也能在大多情况下找出具有更广泛权衡的部署方案, 这说明 MGA-DO 算法比现有的算法能够更有效地解决云环境中面向服务软件部署优化问题; 算法 MGA-DO 与 MGA-DO/S 的结果对比显示, 本文提出的局部搜索策略能够更进一步地提高算法的求解效率和精度.

5.4 处理时间分析

表 3 给出了每个算法处理上述 3 种不同规模案例 (即对 Case1 执行 30000 次评估、对 Case2 执

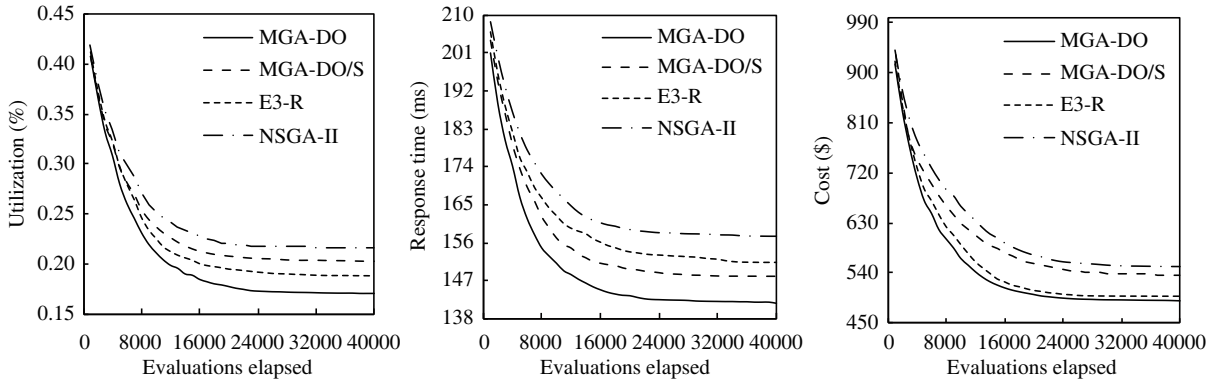

图 12　各算法在 Case2 上运行的演化曲线图

Figure 12　Evolutionary curves of different algorithms on Case2

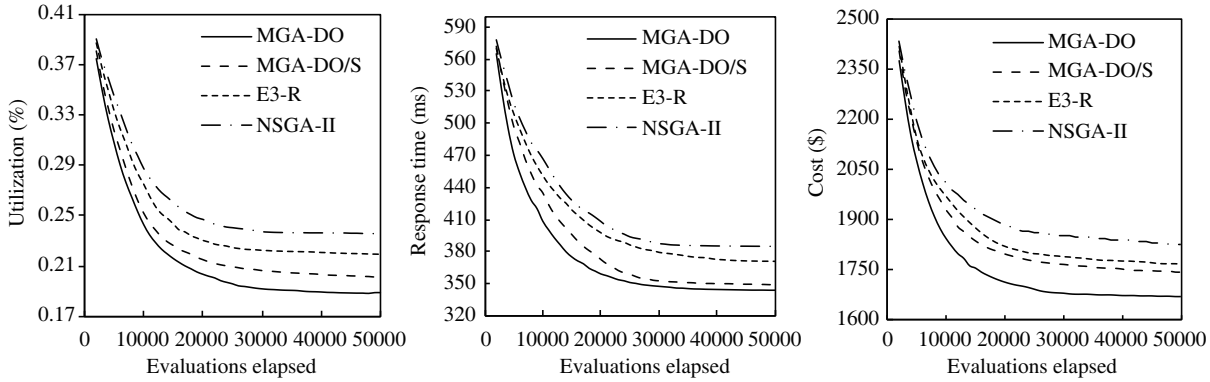

图 13　各算法在 Case3 上运行的演化曲线图

Figure 13　Evolutionary curves of different algorithms on Case3

行 40000 次评估和对 Case3 执行 50000 次评估) 的平均时间花费, 给出的结果是 30 次独立运行所花费的平均时间及其标准方差. 从表 3 中的数据可以看出, 各算法的处理时间相差不大, 因此, 如果以相同的评估次数作为算法的终止条件时, 最终能够获得较好目标函数值的算法具有较好的性能. 根据 5.3 小节中目标函数值的对比分析可知, MGA-DO 算法能够最终获得具有较好目标函数值的部署方案, 因此, 其相比其他算法具有更好的性能.

从表 3 中的数据还可以看出, MGA-DO 算法的处理时间最短, 这是因为本文提出的局部搜索策略只是对现有的部署方案进行局部调整, 因此, 在计算调整后的部署方案的性能时, 只需对受调整影响的部分服务的性能进行重新计算, 其他服务的性能则保持不变, 这在一定程度上降低了部署方案的性能估算时间.

5.5　Hypervolume 分析

本节对比 MGA-DO, MGA-DO/S, E^3-R 和 NSGA-II算法在 3 种不同规模的案例上运行得到的解集的超体积. 每个算法在每个案例上独立运行 30 次, 并将每次运行得到的非支配解集的目标值进行标准化, 然后计算标准化后的目标值与参考点 (1.0, 1.0, 1.0) 之间的超体积. 表 4 给出了每个算法得到的

表 3 各算法的处理时间

Table 3 Processing time of different algorithms

Algorithm	Processing time (s)		
	Case1	Case2	Case3
MGA-DO	148.3 (4.877)	376.7 (7.779)	884 (10.364)
MGA-DO/S	155.6 (4.553)	391.9 (6.381)	921 (10.576)
E^3-R	921 (10.576)	398.1 (6.593)	944.4 (9.724)
NSGA-II	198.3 (3.768)	451.5 (5.718)	1000.6 (8.973)

表 4 各算法 hypervolume 值对比 (最小/最大/平均)

Table 4 Comparison of the hypervolumes obtained by different algorithms (min/max/mean)

Algorithm	Case1	Case2	Case3
MGA-DO	0.0575/0.0716/0.0671	0.0454/0.0547/0.0511	0.0353/0.0467/0.0421
MGA-DO/S	0.0521/0.0672/0.0672	0.0429/0.0528/0.0484	0.0344/0.0458/0.0383
E^3-R	0.0527/0.0666/0.0573	0.0438/0.0531/0.0492	0.0378/0.0441/0.0395
NSGA-II	0.0475/0.0647/0.0533	0.0427/0.0503/0.0461	0.0332/0.0436/0.0372

hypervolume 值的最大值、最小值和平均值.

从表 4 中的数据可以看出, MGA-DO 算法在每个实验案例上获得的解集的最大、最小和平均 hypervolume 值均大于其他算法. 这说明从 hypervolume 指标来看, MGA-DO 算法在求解面向服务软件部署优化问题上也比其他算法有更好的性能; MGA-DO 算法与 MGA-DO/S 算法的 hypervolume 值的对比结果也显示了 MGA-DO 算法中的局部搜索规则比随机变异规则具有更好的效果.

6 结论

本文提出了一种用于求解云环境中面向服务软件部署优化问题的部署优化方法, 支持快速、稳定地为面向服务的软件在云环境中寻找具有高性能和低成本的最优部署方案. 该方法利用排队论获得面向服务软件在不同部署方案下的性能, 并基于此构建了问题的优化模型. 同时, 它还基于遗传算法设计了一种求解该模型的部署优化算法, 通过引入现有的部署优化经验, 设计 5 种局部搜索规则, 提高算法的求解性能. 实验结果表明, 本文提出的算法能够有效提高求解最优部署方案的效率, 且与现有的部署优化算法相比, 本文提出的优化算法能够获得具有更广权衡范围 (即更短的响应时间、更低的最大利用率和更低的成本) 的最优解.

参考文献

1 Liu T, Lu T, Wang W, et al. SDMS-O: a service deployment management system for optimization in clouds while guaranteeing users' QoS requirements. Future Gener Comput Syst, 2012, 28: 1100–1109

2 Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Commun ACM, 2010, 53: 50–58

3 Gu J, Luo J Z, Cao J X, et al. Performance modeling and analysis for composite service considering failure recovery. J Softw, 2013, 24: 696–714 [顾军, 罗军舟, 曹玖新, 等. 考虑失效恢复的组合服务性能建模与分析. 软件学报, 2013, 24: 696–714]

4 Mirandola R, Potena P, Scandurra P. Adaptation space exploration for service-oriented applications. Sci Comput Program, 2014, 80: 356–384

5 Jennings B, Stadler R. Resource management in clouds: survey and research challenges. J Netw Syst Manage, 2015, 23: 567–619

6 Canfora G, Di Penta M, Esposito R, et al. An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, Washington, 2005. 1069–1075

7 Osman I H, Kelly J P. Meta-heuristics: an overview. In: Meta-Heuristics. Berlin: Springer, 1996. 1–21

8 Laili Y J, Zhang L, Tao F, et al. Rotated neighbor learning-based auto-configured evolutionary algorithm. Sci China Inf Sci, 2016, 59: 052101

9 Aleti A, Buhnova B, Grunske L, et al. Software architecture optimization methods: a systematic literature review. IEEE Trans Softw Eng, 2013, 39: 658–683

10 Aleti A, Grunske L, Meedeniya I, et al. Let the ants deploy your software-an ACO based deployment optimisation strategy. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, 2009. 505–509

11 Jayasinghe D, Pu C, Eilam T. Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement. In: Proceedings of the IEEE International Conference on Services Computing, Washington, 2011. 72–79

12 Malek S, Medvidovic N, Mikic-Rakic M. An extensible framework for improving a distributed software system's deployment architecture. IEEE Trans Softw Eng, 2012, 38: 73–100

13 White J, Dougherty B, Thompson C, et al. ScatterD: spatial deployment optimization with hybrid heuristic/evolutionary algorithms. ACM Trans Auton Adap Syst, 2011, 6: 123–154

14 Zhang X W, Cao D G, Chen X Q, et al. Deployment solution optimization for mobile network application. J Softw, 2011, 22: 2866–2878 [张晓薇, 曹东刚, 陈向群, 等. 一种网络化移动应用部署方案优化方法. 软件学报, 2011, 22: 2866–2878]

15 Yusoh Z I M, Tang M. A cooperative coevolutionary algorithm for the composite SaaS placement problem in the cloud. In: Proceedings of International Conference on Neural Information Processing, Sydney, 2010. 618–625

16 Yusoh Z I M, Tang M. A penalty-based grouping genetic algorithm for multiple composite saas components clustering in cloud. In: Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC), Seoul, 2012. 1396–1401

17 Yusoh Z I M, Tang M. Composite saas placement and resource optimization in cloud computing using evolutionary algorithms. In: Proceedings of IEEE 5th International Conference on Cloud Computing (CLOUD), Hawaii, 2012. 590–597

18 Zhao X T, Zhang B, Zhang C S. Service selection based resource allocation for SBS in cloud environments. J Softw, 2015, 26: 867–885 [赵秀涛, 张斌, 张长胜. 一种基于服务选取的 SBS 云资源优化分配方法. 软件学报, 2015, 26: 867–885]

19 Meng F C, Chu D H, Li K Q, et al. Solving SaaS components optimization placement problem with hybird genetic and simulated annealing algorithm. J Softw, 2016, 27: 916–932 [孟凡超, 初佃辉, 李克秋, 等. 基于混合遗传模拟退火算法的 SaaS 构建优化放置方法. 软件学报, 2016, 27: 916–932]

20 Frey S, Fittkau F, Hasselbring W. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proceedings of the 2013 International Conference on Software Engineering, San Francisco, 2013. 512–521

21 Wada H, Suzuki J, Yamano Y, et al. Evolutionary deployment optimization for service-oriented clouds. Softw Pract Exper, 2011, 41: 469–493

22 Potena P. Optimization of adaptation plans for a service-oriented architecture with cost, reliability, availability and performance tradeoff. J Syst Softw, 2013, 86: 624–648

23 Khazaei H, Mišić J, Mišić V B. Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. IEEE Trans Parall Distr Syst, 2012, 23: 936–943

24 Vilaplana J, Solsona F, Teixidó I, et al. A queuing theory model for cloud computing. J Supercomput, 2014, 69: 492–507

25 Menasce D A, Dowdy L W, Almeida V A F. Performance by Design: Computer Capacity Planning by Example. Upper Saddle River: Prentice Hall, 2004

26 Deb K, Pratap A, Agarwal S, et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evolution Comput, 2002, 6: 182–197
27 Yin H, Zhang C, Zhang B, et al. A hybrid multiobjective discrete particle swarm optimization algorithm for a sla-aware service composition problem. Math Problems Eng, 2014, 2014: 1–14

Evolutionary deployment optimization for service-oriented software in cloud

Lin LI[1,2], Shi YING[1,2*], Bo DONG[1,2] & Rui WANG[1,2]

1. *State Key Lab of Software Engineering, Wuhan University, Wuhan* 430072*, China*;
2. *School of Computer Science, Wuhan University, Wuhan* 430072*, China*
* Corresponding author. E-mail: yingshi@whu.edu.cn

Abstract This paper proposes a new deployment optimization method, to address the drawbacks of existing deployment optimization methods, for optimizing the deployment architectures of service-oriented software in cloud. Examples of these drawbacks are the lack of scalability of service instances and virtual machine instances, and the inability to guarantee the solving quality. The method first constructs a deployment optimization model with the goals of improving the running performance and reducing the operation cost of service-oriented software. Next, a genetic-based algorithm MGA-DO is utilized for solving the model. The MGA-DO adopts a group-based encoding scheme to encode the deployment architectures of service-oriented software and combines this scheme with a group-based crossover operator to realize the scalability of service instances and virtual machine instances in the optimization process. Moreover, the MGA-DO utilizes the existing knowledge of deployment optimization to design five types of local search rules, to further improve the local search ability of the algorithm and accelerate the convergence speed. Finally, a series of simulations show that, compared with the existing algorithms, the MGA-DO algorithm performs better on solving the research problem.

Keywords cloud computing, service-oriented software, performance, cost, deployment optimization, genetic algorithm

Lin LI was born in 1988. She received her M.S. degree in computer science from Central China Normal University, Wuhan, China, in 2012. Currently, she is a Ph.D. candidate in the School of Computer Science, Wuhan University, China. Her research interests include service-oriented software architectures, cloud computing, and intelligence computing.

Shi YING was born in 1961. He received his Ph.D. degree from Wuhan University of Technology. Now he is a professor in the School of Computer Science, Wuhan University. His research interests include high-performance software development, semantic web technology, and software architecture and model.

Bo DONG was born in 1987. He received his M.S. degree in computer science from Central China Normal University, Wuhan, in 2013. Currently, he is a Ph.D. candidate in the School of Computer Science, Wuhan University. His research interests include cloud computing, service-oriented software deployment, and performance evaluation.

Rui WANG was born in 1989. She received her M.S. degree from the College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao, China, in 2014. Currently, she is a Ph.D. candidate in the School of Computer Science, Wuhan University. Her research interests include big data analysis, machine learning, and performance analysis.