网格环境中基于SLA的本地任务调度算法*)

曾万聃 常桂然 戴 勃 于振雷
(东北大学计算中心 沈阳 110004)

摘 要 在服务网格中为用户提供满足SLA(service level agreement)的服务,是实现网格"非凡的服务质量"的一个重要的研究问题。本文提出了网格本地资源基于任务SLA的调度算法,给出了算法的数学模型和描述。在基于Java的网格环境调度模拟器中对算法进行了验证,该算法能实现满足用户SLA约束的调度,为满足全局的服务质量水平提供本地调度支持,对提高网格服务质量水平具有实际意义。

关键词 网格,服务质量,调度,资源分派,SLA(service level agreement)

A Local Task Scheduling Algorithm Based on SLA in Grid Environment

ZENG Wan-Dan CHANG Gui-Ran DAI Bo YU Zhen-Lei
(Computing Center, Northeastern University, Shenyang 110004)

Abstract It is an important research area to supply users with the service of SLA in achieving extraordinary Grid QoS. A local task scheduling algorithm based on SLA for single task on grid node is proposed in this paper. Its mathematical model and the process are given. The experiment is taken in Java-based Grid simulated environment. The result shows that this algorithm can satisfy the user's requirement with SLA constraints and gives the local support for the global QoS and it has strong practical significance to improve the QoS for Grid services.

Keywords Grid, QoS(quality of service), Scheduling, Resource allocation, SLA(service level agreement)

1 简介

网格计算是解决科学、工程和经济领域大型应用中的问题的一门新技术[1]。网格是继万维网之后出现的一种新型网络计算平台,目的是为用户提供一种全面共享包括网页在内的各种资源的基础设施[2]。网格技术使用户无论在何时何地都能透明地访问计算和存储资源,并保证一定的服务质量成为可能[3]。网格计算已经成为资源密集型科学应用的主流架构,也将成为商业应用的未来模型。网格使计算资源共享和动态分派成为可能,极大地提高了分布式数据的访问能力,促进了操作的灵活性和合作性,服务提供者能够高效地调整本地策略来满足变化的需求[1,2]。

网格为用户提供了透明的统一的接口,用户不必考虑提供服务或者与之协作的资源的状态。但是由于网格资源的异构性、复杂性、动态性和自治性,网格资源可以随时加入网格资源或者收回正在提供服务的资源,这给网格的调度带来了新的挑战。在网格环境下的调度成为网格研究中的一个重要领域。

2 网格调度算法研究背景

在过去的网格研究中,调度算法主要有两类:基于性能的调度和基于经济的调度。在过去的网格系统中,大多数都是基于性能的调度,这些调度以最小化全部执行时间或性能作为调度目标对作业和资源进行匹配。文[6]利用贪婪算法优化网格调度的性能,贪婪算法循环地给每个作业分派资源来达到最好的性能,调度决策是基于本地作业的信息,不考虑未调度的作业。文[8,9]中的遗传算法考虑了全局的性能,文[7]中利用模拟退火算法来提高网格调度性能。另一个方面,基于经济的调度将资源价格也作为一个优化指标[6,7]。

3 基于SLA的网格资源分派的提出

SLA(service level agreement)是服务提供者和用户间的协议,定义用户可接受的QoS参数等级,通常用与某满意的服务级别和与之对应的代价表述[1]。大型网格是由地理上分布的成千上万的构件组成的复杂系统,在这样复杂的环境中确保一定的服务质量成为一大挑战,因为全局的服务级别保证(service-level agreements)绝大部分取决于本地的[3]调度策略。

网格中许多用户共享同一个资源的时候,如何在网格环境中监视并确保SLAs,是网格环境中一个重要挑战。网格环境定义的一个关键部分就是提供非凡的服务质量[3],应用级别的SLAs必须映射到资源级别的SLAs。当与底层资源协商SLAs时,网格全局资源调度器必须意识到,不同服务提供者可以为不同代价提供不同级别的服务质量,因此本地的资源分派方法对实现全局的SLA至关重要。这样,在网格环境中一个最优化问题就是:如何选择服务策略,取得用最小代价获得全局SLAs[1]。

本文提出了一种适用于网格本地基于用户单个任务SLA的网格资源分派算法,给出了算法数学模型、算法流程和描述,并在模拟环境中对算法进行了模拟,证明算法的有效性。

4 基于SLA的网格资源分派算法

4.1 算法的数学模型

这里采用基于SLA的网格本地调度模型[4],模型的前提是一个任务可以拆分成相互独立的若干个子任务,在满足这

*)高等学校博士学科点专项科研基金(20030145017)。曾万聃 博士研究生,研究方向:网格计算;常桂然 教授,博士生导师。

个条件的前提下，讨论单个用户任务在资源上的拆分调度。设一个应用需要 NC 百万个 CPU 轮询时间，任务结束时间最多 T_{max} 个时间单位，计算代价不能超过 C_{max}。假设最多有 3 个计算资源可以被分派，分别是 $si(i=1,2,3)$，表示资源 i 的速度是每秒 si 百万个 CPU 时间(millions of cycles/sec)，ci 代表使用资源 i 每秒的代价。

将该资源上的任务分成 3 个相互独立的并行任务。3 个计算资源有以下几种可能分派方法，(1)，(2)，(3)，(1,2)，(1,3)，(2,3)，(1,2,3)。

T 代表相应分派方法的执行时间，C 代表代价。一些分派方法是不能够满足全局的 SLA T_{max} 或者不能满足全局最大的代价 C_{max} 的。只有同时满足如下(1)式和(2)式的资源 i 才能够被使用：

$$T=\frac{NC}{s_i}\leqslant T_{max}\quad i=1,2,3 \tag{1}$$

$$C=c_i*\frac{NC}{s_i}\leqslant C_{max}\quad i=1,2,3 \tag{2}$$

式(1)说明在资源 i 上的执行时间不能超过全局 SLA T_{max}。式(2)表明代价的约束。当资源 i 和资源 j 被使用时，使用的时间和代价约束如下：

$$T=\max\{\frac{NC_i}{s_i},\frac{NC_j}{s_j}\}\leqslant T_{max} \tag{3}$$

$$C=c_i*\frac{NC_i}{s_i}+c_j*\frac{NC_j}{s_j}\leqslant C_{max} \tag{4}$$

$$NC_i+NC_j=NC \tag{5}$$

如果 3 个资源都用，表示如下：

$$T=\max\left\{\frac{NC_1}{s_1},\frac{NC_2}{s_2},\frac{NC_3}{s_3}\right\}\leqslant T_{max} \tag{6}$$

$$C=\sum_{i=1}^{3}c_i*\frac{NC_i}{s_i}\leqslant C_{max} \tag{7}$$

$$\sum_{i=1}^{3}NC_i=NC \tag{8}$$

如果忽略代价约束，则选择方法为与资源速度相应的最小执行时间，即

$$N_i=NC*\frac{s_i}{\sum_{j=1}^{3}s_j} \tag{9}$$

如果有 n 个资源，且每个计算任务可以被调度到从 1 到 n 的任何资源上。这样，可能的资源分派方法有：

$$\sum_{k=1}^{n}\binom{n}{k}-1=2^n-1 \tag{10}$$

所以本地的资源分派就是在以上可能中寻找两种用户要求的目标：第一，寻找满足最小代价的分派方法，并满足 $T\leqslant T_{max}$；同样，第二个目标是寻找最小执行时间的资源分派方法，并满足 $C\leqslant C_{max}$。

4.2 算法的描述

4.2.1 对用户代价要求最优化算法(cost_opt)

该算法是在时间约束条件下寻求代价最优的算法。算法流程如下：

1 将主机上的各个处理单元按照执行每兆指令所需的代价进行升序排列。
2 执行如下循环，循环变量 i 从零开始递增：
  2.1 将任务放在前 $i+1$ 个处理器上执行，并保证分配在这些处理器上的任务量刚好可以让这些处理器同时结束工作；
  2.2 检查在目前的分配情况下是否可以在规定的时间限制(deadline)之内完成任务。如果是，转 3；如果不能完成任务，转 2。
3 将任务分配在前 i 个处理器上，并使前 i 个 cpu 上的任务刚好在 deadline 指定的时间内完成。剩下的任务分配给第 $i+1$ 个处理器。因为经过了第二步的判断，因此在这里，第 $i+1$ 个 cpu 能够在用户执行的 deadline 之前完成为其分配的任务。
4 将在各个 cpu 上的任务分配量作为结果返回给调用者。

算法的类 Java 的算法描述如下：

```
1  Collections.sort(host.getCPUS(),new PPSComparator());
2  for(i=0;i<host.getCPUS().size();i++){
   2.1  totalMIPS+=((CPU)host.getCPUS().get(i)).getMIPS
        ();
   2.2      if(task.getLength()/totalMIPS<=task.getDeadline
            ()){
              succ=true;
              break;
            }
        }
3  if(succ){
          Snippet[]snippets=new Snippet[i+1];
          double totallen=0;
          for (int j=0;j<=i;j++){
            CPU acpu=(CPU)host.getCPUS().get(j);
            if(j! =i) {
              snippets[j]=new Snippet(acpu.getMIPS()*
              task.getDeadline(),acpu);
                  totallen+=acpu.getMIPS()*task.getDead-
              line();
            }else{
              snippets[j]=new Snippet(task.getLength()-to-
              tallen,acpu);
            }
          }
        } else{
          snippets=null;
        }
4  return res;
```

4.2.2 对用户时间要求最优化算法(time_opt)

在预算因素允许的情况下最大限度地优化时间。

1 将主机上的各个处理单元按照执行每兆指令所需的代价进行升序排列。

2 执行如下循环，循环变量 i 从处理器的最后一个所对应的下标开始递减：

2.1 将任务放在前 $i+1$ 个处理器上执行，并保证分配在这些处理器上的任务量刚好可以让这些处理器同时结束工作；

2.2 判断在目前的分配情况下，是否可以在规定的代价限制之内完成任务。如果是，转 3；如果不能完成任务，转 2。

3 将任务分配到前 i 个处理器上，使其刚好同时完成各自的任务。剩下的任务分配给第 $i+1$ 个处理器。在这一过程中，应该保证 $i+1$ 个处理器的总代价与用户指定的预算相等。

4 将在各个 cpu 上的任务分配量作为结果返回给调用者。

算法的类 Java 描述如下：

```
1  Collections.sort(host.getCPUS(),new PPSComparator());
2  for(i=host.getCPUS().size()-1;i>=0;i--){
   2.1      totalMIPS=getTotalMIPS(host.getCPUS(),i);
            totalTime=task.getLength()/totalMIPS;
            totalCost=getTotalCost(host.getCPUS(),i,total-
            Time);
   2.2      if(totalCost<=task.getBudget()){
              success=true;
              break;
            }
      }
3  if(success){
       if(i==(host.getCPUS().size()-1)){
         Snippet[] snippets=new Snippet[i+1];
         for(int j=0;j<=i;j++){
           CPU acpu=(CPU)host.getCPUS().get(j);
           snippets[j]=new Snippet(acpu.getMIPS()*total-
           Time,acpu);
         }
       }else{
         double k=0;
         k=calK(i); //k 为第 i+1 个处理器执行 1 兆指令的代
         价与将这 1 兆指令平均分配给前 i 个处理器来执行的代
         价的差值
         double lenIP1=0;
         lenIP1=(task.getBudget()-totalCost)/k;
         Snippet[] snippets=new Snippet[i+2];
       snippets[i+1]=new Snippet(lenIP1,((CPU)host.getC-
```

```
            PUS().get(i+1)));
            double newLen=task.getLength()-lenIP1;
            double newTime=newLen/getTotalMIPS(host.getCPUS
            (),i);
            for(int j=0;j<=i;j++){
                CPU acpu=(CPU)host.getCPUS().get(j);
                snippets[j]=new Snippet(acpu.getMIPS()*new-
                Time,acpu);
            }
        }
    } else{
        res=null;
    }
4   return res;
```

5 实验

网格环境调度模拟器 GridSim 模拟了一个网格计算环境，包括网格环境下的用户、异构的资源、应用、资源 broker 和调度器等网格环境下的类，可以对网格调度分两层不同的算法进行模拟。首先是任务到达资源 broker，对所有任务进行资源分派，确定任务执行的节点；然后，资源到达本地后对本地任务进行资源分派，调度执行。在 GridSim 中可以分别对这两个层次的调度加入新的调度算法，进行网格环境中调度算法的验证。我们对本文算法在本地调度层进行了 20 组实验，验证了算法的有效性。挑选两组不同资源配置下、不同任务需求的任务拆分情况和调度结果作为例子，说明利用本文的优化算法之后，能够取得用户 SLA 要求的服务质量的服务。

5.1 实验一

资源配置如表 1，本地资源包含 3 个 Sun Ultra os：Solaris 处理器，具体配置设置见表 1。

表 1 资源配置

处理器标号	MIPS	每秒花费代价
1	100	10
2	50	3
3	150	17

在这个资源上，分别设置了 3 个不同任务，并以 3 种不同的算法来测试调度的结果。NONE_OPT 行代表用自动调度机制得到的调度结果，Cost_OPT 行代表用 Cost_OPT 算法的调度结果，Time_OPT 代表用 Time_OPT 算法的调度结果。

(1)任务 1。长度=1000MIPS，SLA 约束：deadline=15s，budget=90。对该任务用不同算法在资源 1 上进行调度，结果如表 2。

表 2 调度结果 1

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1:1000	100	10	失败
Cost_OPT	cpu1:250,cpu2:750	70	15	成功
Time_OPT	cpu1:583,cpu2:292,cpu3:125	90	5.84	成功

(2)任务 2。长度=1000MIPS，SLA 约束：deadline=50s，budget=200。对该任务用不同算法在资源 1 上进行调度，结果如表 3。

表 3 调度结果 2

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1:1000	100	10	成功
Cost_OPT	cpu2:1000	60	20	成功
Time_OPT	cpu1:333,cpu2:167,cpu3:500	00	3.33	成功

(3)任务 3。长度=1000MIPS，SLA 约束：deadline=9s，budget=90。对该任务用不同算法在资源 1 上进行调度，结果如表 4。

表 4 调度结果 3

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1:1000	100	10	失败
Cost_OPT	cpu1:550,cpu2:450	82	9	成功
Time_OPT	cpu1:583,cpu2:292,cpu3:125	90	5.83	成功

5.2 实验二

资源配置如表 5，本地资源包含 4 个 Sun Ultra os：Solaris 处理器，具体配置设置见表 5。

表 5 资源配置 2

处理器标号	MIPS	每秒花费代价
1	200	每秒花费=20
2	250	每秒花费=24
3	150	每秒花费=14
4	100	每秒花费=5

在这个资源上，分别设置了 3 个不同任务，并以 3 种不同的算法来测试调度的结果。

(1)任务 1。长度=2500MIPS，SLA 约束：deadline=30s，budget=250。对该任务用不同算法在资源 2 上进行调度，结果如表 6。

表 6 调度结果 4

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1:2500	250	12.5	成功
Cost_OPT	cpu4:2500	125	25	成功
Time_OPT	cpu1:714.3,cpu2:892.86, cpu3:535.71,cpu4:357.14	225	3.57	成功

(2)任务 2。长度：2500，SLA 约束：deadline=23s，budget=200。对该任务用不同算法在资源 2 上进行调度，结果见表 7。

表 7 调度结果 5

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1: 2500	250	12.5	失败
Cost_OPT	cpu3:200,cpu4:2300	133.67	23	成功
Time_OPT	cpu2:500,cpu3:1200,cpu4:800	200	8	成功

(3)任务 3。长度：2500，SLA 约束：deadline=10 budget=200。对该任务用不同算法在资源 2 上进行调度，结果见表 8。

表 8 调度结果 6

调度策略	任务拆分调度结果	总代价	总时间	备注
NONE_OPT	cpu1: 2500	250	12.5	失败
Cost_OPT	cpu3:1500,cpu4:1000	190	10	成功
Time_OPT	cpu2:500,cpu3:1200,cpu4:800	200	8	成功

结论 本算法能够实现网格环境中基于用户任务 SLA 约束的本地任务拆分调度，为全局服务质量水平提供本地 SLA 调度支持。但是由于试验是在理想环境下进行，在真实的环境中，还要考虑到数据传输的通信延迟、高层协调等等。但是本算法对于实现全局的服务质量水平提供本地调度的支持、满足网格任务 SLA、提高网格服务质量水平具有实际意

义。

参 考 文 献

1 Foster I, kesselman C. The Grid: Blueprint for a New Computing Infratructure[M]. Morgan Kaufmann, 2004
2 Douglis F, Foster I. The Grid Grows Up [J]. IEEE Internet Computing, 2003, 7(4): 24～26
3 Leff A, Rayfield J T, Dias D M. Service-Level Agreements and Commercial Grids [J]. IEEE Internet Computing, 2003, 7(4): 44～50
4 Menascé M A, Casalicchio E. QoS in Grid Computing. Journals of IEEE Internet Computing [J], July • August 2004
5 Chen Hanhua, Jin Hai, Mao Feng, et al. Q-GSM: A QoS Oriented Grid Service Management Framework
6 Raman R, Livny M, Solomon M. Matchmaking: Distributed resource management for high throughput computing [A]. Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (1998), Chicago, Illinois, July 28-31, 1998
7 Khan A Y. Experiments with Scheduling Using Simulated Annealing in a Grid Environment [A]. In: Proceedings of GRID 2002. Baltimore, MD, USA, 2002. 232～242
8 Abraham A, Buyya R, Nath B. Nature's Heuristics for Scheduling Jobs on Computational Grids [A]. In: Proceedings of the 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000), December 14-16, 2000, Cochin, India, 2000. 45～52
9 Martino V, Mililotti M. Scheduling in a Grid Computing Environment Using Genetic Algorithms [A]. In: Proceedings of International Parallel and Distributed Processing Symposium, 2002

(上接第 60 页)

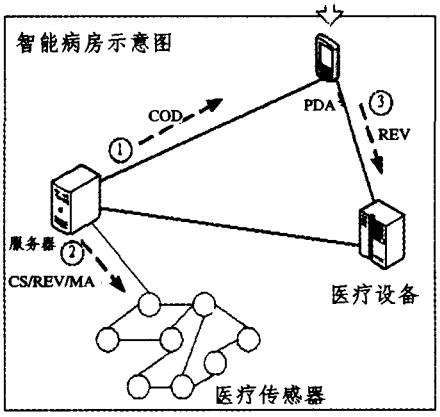

图 2 智能病房示例

5.2 模式选择策略

下面给出了用 Ponder 语言对模式进行选择与优化的部分描述，其中策略被直接实例化：

```
/*需要设定检测参数，完成一定功能的检查，如按照病人的呼吸、心跳、脉搏等确诊是否有呼吸道疾病，则采用 REV 模式，具体检查方法和需要数据在功能代码(know-how code)中描述*/
inst oblig CustomizedInfo{
  on SpecialCheck(MainDoctor)
  subject /HospitalDevices /RoomDevice
                           /Server
  target HospitalDevices /RoomDevice
                         /Sensor
  action REVParadigm(know-how code)
}
/*服务器在运行过程中，由于供电或带宽资源不足或系统故障出现不稳定状态，就采用 REV 或 MA 模式，使查询在传感器总体服务程序处本地完成，然后将结果传回服务器*/
inst oblig InstabilityServer{
  on InstabilityServer(RoomServer)
subject HospitalDevices /RoomDevice
                        /Server
  target /HospitalDevices /RoomDevice
                          /Sensor
  action MAInteraction(know-how code,evn state,execution state)
}
```

5.3 策略制定规则分析

在这个场景中，服务器与医疗传感器之间的通信可以采用 C/S、REV、MA 3 种方式。在网络和设备资源都充足的情况下，选择哪种方式更优化，可以做一个简单的分析比较。为了获取医疗传感器的值，服务器要与各个传感器进行交互。也就是说，这个场景属于多个网络节点的情况。根据上面的分析，假设有 n 个传感器需要通信，则

C/S 模式中：

$$B_{c/s}=(2h+B_{pa}+B_r)Qn \tag{11}$$

$$T_{c/s}=(2h+B_{pa}+B_r)Qn/\delta \tag{12}$$

REV 模式和 MA 模式按照公式(7)～(10)。

假设移动组件的大小远远大于指令头的大小。设 $h=20$ Byte，$B_{pa}=20$Byte，$B_r=150$ Byte，$Q=n=15$，$B_{code}=400$Byte，$B_{mac}=600$ Byte，$T_s=0.1$s，带宽 $\delta=10000$ Byte/s。可以将 3 种交互模式的网络流量和通信时间的值做一个柱状图做比较，如图 3 所示。

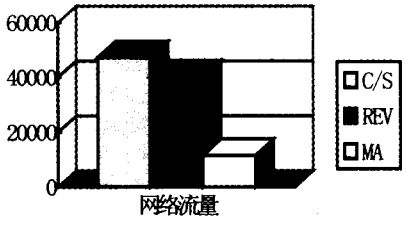

图 3(1) 3 种迁移模式的网络流量

由此可见，在网络中需要多个节点相互通信时，移动代理方式的通信流量和通信时间都是最佳的。远程评估方式次之，C/S 方式最差。在实际制定策略的时候，就可以类似上面的例子，按照所关注的指标量进行分析计算，配合以资源所有情况决定的定性分析，选择最佳交互模式。

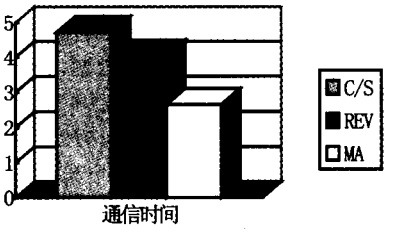

图 3(2) 三种迁移模式的通信时间

结论 本文分析了迁移模式集成的必要性和实现方法，提出了用策略实现的多种迁移模式集成模型。并对如何制定迁移规则进行了定性和定量的分析。集成模型不仅能够有效地满足普及计算动态多变环境的要求，还使得整个系统具有很强的灵活性和通用性。

参 考 文 献

1 Fuggetta A, Picco G P, Vigna G. Understanding code mobility. Software Engineering, IEEE Transactions on, 24(5)
2 Montanari R, Lupu E, Stefanelli C. Policy Based Dynamic Reconfiguration of Mobile Code Applications. PDF Computer, 2004, 37(7): 73～80
3 Montanari R, Tonti G, Stefanelli C. A policy-based mobile agent infrastructure. Proceedings, Applications and the Internet, 2003