一种基于调度簇树的周期性分布实时任务调度算法*)

王小非[1] 方 明[2]

(华中科技大学计算机科学与技术学院 武汉 430074)[1] (武汉数字工程研究所产品研发部 武汉 430074)[2]

摘 要 本文针对现有的基于任务复制的静态调度算法在调度周期性分布实时任务时存在的缺点,提出了一种称之为调度簇树(SCT)的新的结构并研究了其特性,在此基础上给出了一种基于SCT树的周期性分布实时任务调度算法(SAS)。通过与OSA算法进行比较的实验结果表明,SAS算法可实现调度长度向上最接近分布实时任务周期,最大程度减少所需预留处理器数目,大大提高分布实时系统的处理器利用率,同时并不增加调度算法的复杂度。

关键词 SCT树,任务调度,DAG,任务复制,分布实时系统

A Scheduling Algorithm of Periodic Distributed Real-Time Tasks Based on Scheduled Clusters Tree

WANG Xiao-Fei[1] FANG Ming[2]

(College of Computer Soience and Technology, Huazhong University of Science and Technology, Wuhan 430074)[1]

(Department of Research & Development, Wuhan Digital and Engineering Institute, Wuhan 430074)[2]

Abstract To overcome the disadvantages of existing static scheduling algorithms based on tasks duplication when they are adopted in Real-time Distributed System, a Scheduling Algorithm of Periodic Distributed Real-Time Tasks is presented which we name it SAS algorithm. As the result of experiments conducted to compared with the scheduling by some typical algorithms (e.g. OSA and PPA algorithm) shows, SAS algorithm makes Scheduled Length approach the period of tasks, reduces the number of needed processors to the minimum, and improves the utilization of processors greatly; meanwhile, the complexity of the scheduling algorithm is not increased.

Keywords Scheduled Clusters Tree(SCT), Task scheduling, DAG, Task duplication, Distributed real-time tasks

1 引言

在混合型分布实时系统中,任务可分为周期性实时任务和非周期性任务两类。为保证周期性实时任务能在既定的周期内执行完成,一般采用处理器预留策略和静态调度算法;对于非周期性任务,则采用动态调度算法。本文的研究着眼于周期性实时任务的静态调度。

对静态调度算法的研究表明,基于任务复制的调度算法比无任务复制的调度算法具有较短的调度长度[1],因而更适用于分布实时系统。其中部分基于任务复制的调度算法在一定的限制条件下可得到调度长度上的最优调度,例如,1998年Darbha提出的TDS[2]算法和2002年Park提出的OSA[3]算法。在调度长度方面,TDS算法比在它以前出现的所有算法都优越,因为它通过任务复制减少了通信时间。而OSA算法实际上是一个扩展的TDS算法,因为它减少了算法的限制条件,增加了最优调度的适用范围。

当我们考虑采用OSA算法调度分布实时系统中的周期性实时任务时,还要解决两方面的问题:第一,OSA算法假设处理器数目无限,这与我们希望尽量减少预留处理器数目的初衷是不相符的;第二,当分布实时任务的周期已确定,盲目追求调度长度最短,会造成预留处理器资源的浪费。因此,分布实时系统中的周期性实时任务调度的目标,应以调度长度不超过且最接近任务周期为主要目标,以所需处理器最少为次要目标。

为减少任务调度所需处理机数目,2005年文[4]提出了PSO算法,对于基于任务复制的调度算法(如TDS和OSA)可在不影响已获得的调度长度和现有算法复杂度的前提下,使任务调度所需处理器数目达到最少。

在PSO算法基础上,本文按照上述周期性实时任务的调度目标,基于一种称之为调度簇树(SCT)的新的结构,提出了一种周期性分布实时任务调度算法(SAS)。

2 调度模型

一组有序任务可以表示为一个DAG图(如图4)。作为调度模型,DAG图可抽象为$G=(V,E,\tau,C)$。这里:$V=\{n_i \mid n_i$ 是有序任务;$i=1,2,\cdots,v, v=|V|$ 表示任务的数目$\}$;$E=\{e_{ij} \mid e_{ij}$ 表示任务 n_i 到 n_j 的通信$\}$;$\tau=\{\tau_i \mid \tau_i$ 表示任务 n_i 的计算时间,$i=1,2,\cdots,v\}$;$C=\{c_{ij} \mid c_{ij}$ 表示任务 n_i 到 n_j 的通信时间$\}$。其中,任务 n_i 的前驱 $\mathrm{pred}(n_i)=\{n_j \mid e_{ji}\in E\}$;任务 n_i 的后继 $\mathrm{succ}(n_i)=\{n_j \mid e_{ij}\in E\}$;如果 $|\mathrm{pred}(n_i)|=0$,则 n_i 为开始任务(记为 n_{start});如果 $|\mathrm{pred}(n_i)|\geqslant 2$,则 n_i 为join任务;如果 $|\mathrm{succ}(n_i)|=0$,则 n_i 为结束任务(记为 n_{end});如果 $|\mathrm{succ}(n_i)|\geqslant 2$,则 n_i 为fork任务。

对DAG图的调度结果是映射到不同处理器的一组有序任务簇,簇内的各任务在一个处理器上非抢占地顺序执行。记有序任务簇为 C_k,$k=1,2,\cdots,m$,m 是有序任务簇的数目,也是调度该DAG图执行所需处理器的数目。若以 $L(C_k)$ 表示一个有序任务簇全部任务的执行时间,则定义 $SL=\max(L(C_k))$ 为该DAG图的调度长度。

*)本文受十五国防重点预先研究项目(413160201)资助。王小非 博士生,研究方向为安全数据库、计算机网络等;方 明 博士研究生,主要研究方向是分布实时计算及计算机容错技术。

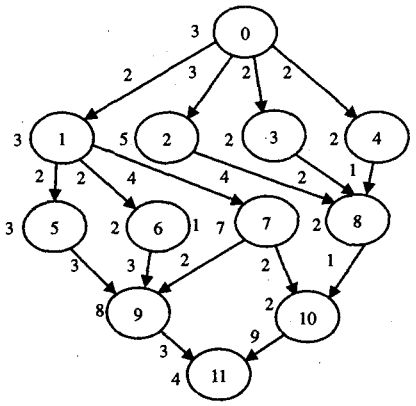


图 1　DAG 图

一组周期性分布实时任务可由一个 DAG 图及其周期来表示。任务的周期即为任务的 Deadline，这里简记为 D。当一组任务的周期 D 确定，其 DAG 图的调度长度 SL 必须满足 $SL \leqslant D$。图 1 为一组周期性分布实时任务所对应的 DAG 图，其周期 D 为一确定值，例如 $D=27$。

3　SCT 树及其特性

图 2 为 OSA 算法的调度图 1 中 DAG 图的结果。为区别于调度过程中产生的任务簇，可称调度完成后映射到同一处理器上的有序任务簇为调度簇(Scheduled Cluster)，则 DAG 图的调度结果映射为一组调度簇，记为 $C_i, i=1,2,\cdots,m$。其中，包含开始任务和结束任务、执行时间等于调度长度 SL 的调度簇为关键调度簇，记为 C_{key}。C_{key} 之外的所有其它调度簇为非关键调度簇，记为 C_{n-key}。图中灰色方格表示一个处理器空闲的时间单位。

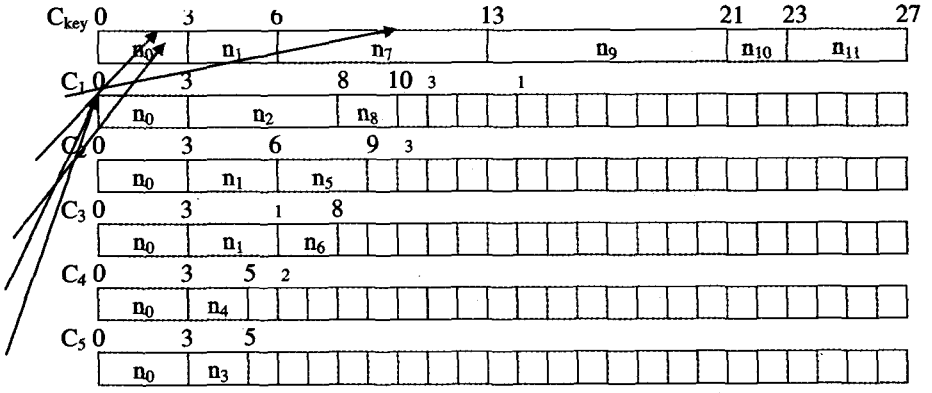


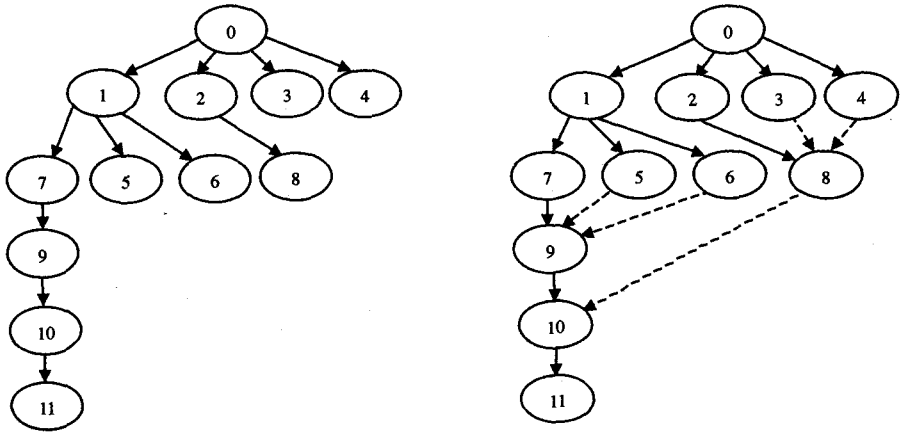
图 2　OSA 算法的调度结果($SL=D=27$)

调度完成后，对于 DAG 图中 $\forall e_{kl} \square E$，若任务 n_k、n_l 分别被调度到调度簇 C_i、C_j，$i \neq j$，则 c_{kl} 不可忽略，可称此类不同调度簇之间的通信为剩余通信(Remained Edge)，记为 e_r，且记 $|e_r|$ 为调度完成后剩余通信的数目。按照调度模型，剩余通信之外的其它通信为相同处理器上的任务间通信，通信时间忽略不计。

由于 DAG 图中的 fork 任务所在的任务簇总是被复制并与其后继任务合并形成新的任务簇，使得调度完成时得到的各调度簇中必然存在一部分位于调度簇靠前位置的任务因任务复制而相同，如 n_0，n_1。

定义 1(SCT 树)　对于 DAG 图经调度后得到的一组调度簇 $C_i (i=1,2,\cdots,m)$，以 C_{key} 为基础，将各 C_{n-key} 上的复制任务合并而生成的树称为调度簇树(Scheduled Clusters Tree)，简称为 SCT 树。

例如：对于图 2 中的各调度簇，以 C_{key} 为基础，通过合并复制任务 n_0、n_1，可得图 3(a)中的 SCT 树，图 3(a)中 的调度簇 $C_{key}=\{n_0,n_1,n_7,n_9,n_{10},n_{11}\}$，$C_1=\{n_0,n_1,n_5\}$，$C_2=\{n_0,n_1,n_6\}$，$C_3=\{n_0,n_2,n_8\}$，$C_4=\{n_0,n_3\}$，$C_5=\{n_0,n_4\}$，可在 SCT 树上可由根节点到每个叶子节点的通路表示；此外，剩余通信 e_r 可在 SCT 树上用虚线箭头标出，分别为 $e_{3,8}$、$e_{4,8}$、$e_{5,9}$、$e_{6,9}$、$e_{8,10}$，如图 3(b)。



(a) 由图 2 中的各调度簇合并得到的 SCT 树　　(b) 带剩余通信的 SCT 树

图 3

在SCT树上，对于$\forall n_j \square$调度簇C_i，我们引入四个值来刻画其执行时间：

$est_{c_i}(n_j)$和$ect_{c_i}(n_j)$分别表示C_i中任务n_j的最早开始时间和最早结束时间。在SCT树上，对于SCT树中的入口节点，$est_{c_{key}}(n_{start})=0$，$ect_{c_{key}}(n_{start})=\tau_{start}$；对于其他节点$n_j$，$est_{c_i}(n_j)$通过基于任务复制的调度算法调度后得到，$ect_{c_i}(n_j)=est_{c_i}(n_j)+\tau_j$；对于出口节点，$est_{c_{key}}(n_{end})=SL-\tau_{end}$，$ect_{c_{key}}(n_{end})=SL$。

$lst_{c_i}(n_j)$和$lct_{c_i}(n_j)$分别表示C_i中任务n_j的最迟开始时间和最迟结束时间。在SCT树上，对于出口节点，$lct_{c_{key}}(n_{end})=SL$，$lst_{c_{key}}(n_{end})=SL-\tau_{end}$；对于其他节点$n_j$，其$lst_{c_i}(n_j)$和$lct_{c_i}(n_j)$值可从SCT树上反向求得。

定义2(聚簇空间) 对于任何调度簇C_i，若其叶子节点(记为$leaf_i$)的最迟结束时间$lct_{c_i}(leaf_i)$确定，则定义$CR_i=lct_{c_i}(leaf_i)-\tau_i$为调度簇$C_i$的聚簇空间(Clustering Room)。

定义3(聚簇序列) 若调度簇C_i的聚簇空间$CR_i>0$，则对于SCT树上任何非关键调度簇C_k，$k\neq i$，C_k上总存在一个由与C_i不重合的节点所构成的序列，称为C_k上对应于聚簇空间CR_i的聚簇序列(Clustering List)，记为$CL_{k,i}$。

例如，图3(a)中的SCT树上，对于$C_1=\{n_0, n_1, n_5\}$的聚簇空间CR_1，其他调度簇上对应的聚簇序列为$CL_{key,1}=\{n_7, n_9, n_{10}, n_{11}\}$，$CL_{2,1}=\{n_6\}$，$CL_{3,1}=\{n_2, n_8\}$，$CL_{4,1}=\{n_3\}$，$CL_{5,1}=\{n_4\}$。

因为聚簇序列$CL_{k,i}$由C_k上与C_i不重合的节点所构成，所以若将聚簇空间CR_i与其对应的聚簇序列$CL_{k,i}$合并，实际上是将调度簇C_i与C_k进行合并。若以$L(CL_{k,i})$表示聚簇序列$CL_{k,i}$上所有任务计算时间之和，则发生合并的基本条件是$CR_i \geqslant L(CL_{k,i})$。

定义4(聚簇) 若根据聚簇空间与聚簇序列的对应关系将两个调度簇合并，且合并不使SCT树的调度长度SL增加，则该合并操作称为聚簇(Clustering)。

调度簇合并是否成为一次有效的聚簇，要看合并后SCT树的调度长度是否增加，因此需要判定聚簇的有效性。若判定聚簇有效，则SCT树增加了一个生成的调度簇，减少了两个被合并的调度簇，从而使所需处理器的数目减少。此外，若被聚簇的两个调度簇之间存在剩余通信e_r，则聚簇使该e_r转化为调度簇内的节点间通信，导致$|e_r|$减少。

定理1 在SCT树上，非关键调度簇的叶子节点至少存在一个后继。

证明：假设在SCT树中，存在一个没有后继的非关键调度簇叶子节点，则该叶子节点为n_{end}。此时，关键调度簇的叶子节点亦为n_{end}，则SCT树上共有两个n_{end}。

根据调度模型，DAG图仅有一个n_{end}。则在调度DAG图的过程中，n_{end}由于没有后继，根据基于任务复制的调度算法，n_{end}不会被复制，则调度生成的调度簇中，仅有一个n_{end}，所以SCT上也仅有一个n_{end}，导致矛盾，原命题得证。

由于叶子节点的后继不可能在同一调度簇上，故必然存在一条e_r指向其后继。由定理1，叶子节点可能有一个或多个后继。若叶子节点有一个后继，如图1(a)中C_1的叶子节点n_6，则$lct_{c_1}(n_6)=lst_{c_{key}}(n_{10})-c_{6,10}$；若叶子节点有多个后继，如图1中$C_3$的叶子节点$n_2$，则$lct_{c_3}(n_2)=\min(lst_{c_{key}}(n_8)-c_{2,8}, lst_{c_6}(n_9)-c_{6,9})$。

显然，若$C_{n\text{-}key}$叶子节点所有后继的lst值都确定，叶子节点的lct值可确定，从而叶子节点所在调度簇的聚簇空间CR值可确定。

定理2 对于聚簇生成的调度簇，若其上任何节点在其它调度簇上的后继不因聚簇而使est值增大，则判定聚簇有效。

证明：对于调度簇Ci的聚簇空间CRi，若依据策略选择另一调度簇Ck上对应的聚簇序列CLk，i与其聚簇，则记生成的调度簇为$M=\text{Merge}(Ci, Ck)$。依据聚簇的定义，当聚簇后SCT树的调度长度SL不受影响时，聚簇有效。

对于$\forall n_p \square C_i$或C_k，合并后$n_p \square M$。则对于任何$n_p \square M$，可搜索每一条e_r，若该e_r是由n_p出发，且不是M内的节点间通信，则n_p在M之外的其他调度簇上存在后继。若任何M上节点在其他调度簇上的后继不因聚簇而使est值增大，则SCT树上任何不属于M的节点不因聚簇而使est值增大，因而n_{end}不因聚簇而使est值增大，即SCT树不因聚簇而使的SL增大，故判定聚簇有效。得证。

4 SAS算法

SAS算法的主要思想基于以下两点：

1. 将DAG图的调度结果合并为SCT树，搜索所有调度簇的聚簇空间，继而搜索及其对应的聚簇序列，采用适当策略选择聚簇序列与聚簇空间进行聚簇，从而减少处理器的数目。

2. 计算关键调度簇C_{key}的聚簇空间时，采用任务周期D代替关键调度簇的叶子节点(即n_{end})的lct值(即SL)，则$CR_{key}=D-\tau_i$，由此SL与D之间的空闲时间作为聚簇空间被充分利用，因而可使SL不超过且最接近D。

因为一次有效的聚簇将减少一个$C_{n\text{-}key}$，我们采用最大聚簇空间-最小聚簇序列策略。该策略依据如下的启发思想：当聚簇空间可容纳多个聚簇序列时，首先选择最大聚簇空间与最小聚簇序列进行聚簇，可使聚簇发生的次数最多。具体实现时，先选择CR值最大的调度簇，其对应的各聚簇序列的计算时间$L(CL)$值存在三种情况：1)若任何$L(CL)$值大于CR值，则不能发生聚簇；2)若存在一个或多个$L(CL)$值小于等于CR值，但不存在多个$L(CL)$值之和小于等于CR值，则只能发生一次聚簇。此时选择$L(CL)$值最大的聚簇序列参与聚簇，使聚簇空间尽可能被利用；3)若存在多个$L(CL)$值之和小于等于CR值，则可能发生多次聚簇。此时选择$L(CL)$值最小的聚簇序列参与聚簇，使聚簇发生的次数尽可能多。

SAS算法可描述为：

第一步，根据基于任务复制的调度算法对DAG图的调度结果，获取SCT树的信息(SCT树不需要实际生成)。此时SCT树上各调度簇与剩余通信为已知，所有节点的est、ect值为已知。

第二步，当$SL \leqslant D$时，使$lst_{c_{key}}(n_{end})=SL=D$；否则判定算法不能接受该DAG图的调度。

第三步，搜索具有确定lct值的叶子节点，计算其所在调度簇的聚簇空间CR值(搜索开始时，只有$lst_{c_{key}}(n_{end})$确定，则CR_{key}可计算)。若存在聚簇空间为0，则标记为已处理，置已处理调度簇上非叶子节点的lst值为与其est值相等。

第四步，按照策略选择最大CR值所在调度簇，搜索其对应的各聚簇序列。

第五步，按照策略对最大聚簇空间及其对应的各聚簇序列的关系进行判断。若为策略中所述的第1种情况，则对该聚簇空间标记为已处理；若为第2或第3种情况，则按照策略选择一个聚簇序列。

第六步，将所选聚簇空间与聚簇序列所在的调度簇进行合并，得到新生成的调度簇，并进行聚簇有效的判定。若判定聚簇无效，则取消聚簇生成的调度簇，标记所选聚簇序列为已处理；否则判定聚簇有效，用生成的调度簇替换聚簇空间所在的调度簇，更新其聚簇空间和剩余通信，对聚簇序列所在的调度簇标记为已删除。

第七步，返回第二步，循环搜索-聚簇。当所有未删除调度簇的聚簇空间都标记为已处理时，结束循环。

第八步，输出搜索-聚簇后所有未删除调度簇并映射至处理器。

图 4(a)为采用 SAS 算法进行优化 OSA 调度的伪代码，首先执行 OSA_Schedule()函数，使所得调度簇的调度长度最短，然后执行 Search_Cluster()函数，使处理器数目最少。

图 4(b)为 Search_Cluster()函数的伪代码，它是 SAS 算法的主体，负责对 SCT 树进行搜索-聚簇。其中又调用了以下三个函数：

Select_List()函数，负责按照最大聚簇空间-最小聚簇序列策略选择参与聚簇的聚簇序列。其伪代码如图 4(c)。

Merge()函数，负责完成聚簇操作。该函数与 OSA 算法中用于合并任务簇的 Merge()函数类似，且已证明该函数生成的任务簇在完成时间上具有最优性(见文[3])，故其伪代码略去。

```
SAS_Optimize(G)
{ //input: DAG 图 G(V,E,τ,C), 且其任务周期为 D
  //output: 调度长度 SL 最短、处理器数目 m 最少的一组调度簇
    OSA_Schedule(G); //采用 OSA 算法，使所得调度簇的调度长度 SL 最短
    if SL≤D then
       estC_key(n_end)=SL=D;
       Search_Cluster(clusters); //对 SCT 树进行搜索-聚簇，使处理器数目 m 最少
    else DAG 图不被接收;
} //end of SAS_Optimize
```

(a) 采用 SAS 算法进行优化的伪代码

```
Search_Cluster(clusters)
{//input: a set of clusters(Shortest_SL_Schedule(G)的调度结果)
  //output: a set of clusters(SAS 算法的优化结果)
    for(i=0; i<m; i++) //初始化各调度簇，m 为调度簇的数目，cluster[0]为 C_key
       cluster[i].resultflag=true; //resultflag=false 指调度簇标记为已删除
       cluster[i].roomflag=true; //roomflag=false 指 C_n-key 聚簇空间标记为已处理
    do{
       for(i=0; i<m; i ++) //求调度簇的 CR 值
          if((cluster[i].resultflag) && (cluster[i].roomflag))
             if(i ==0)计算 CR_key; //cluster[i]为 C_key
             else
                for(j=0; j<D; i++) //D 为 cluster[i]的叶子节点 leaf[i]的出度(D<d)
                   if(leaf[i]指向 C_key 或已处理的 C_n-key)//判断 leaf[i]的 lct 值是否可确定
                      计算 leaf[i]的 lct 值; //从多个 lct 值中选最小值
                计算 cluster[i]的 CR 值;
          if(CR 值为 0)
             cluster[i].roomflag=false;
             置 cluster[i]上非叶子节点的 lst 值与其 est 值相等;
       r←i;//保存 CR 值最大的调度簇的序号
       for(i=1; i<m; i++) //搜索 cluster[r]在各 C_n-key 中对应的聚簇序列
          if((cluster[i].resultflag)&&(i!=r)&&(cluster[l]相对于 cluster[r]未处理))
            list-num=0;
             将在 cluster[i]中，而不在 cluster[r]中的节点序列保存在 temp-list[]中;
             if(temp-list[]的长度<cluster[r].room)
                将 temp-list[]保存到 list[list-num][]中;
                list-num++;
       l=Select_List(list[][]); //l 保存按策略所选择的聚簇序列所在调度簇的序号
       if(l!=0)
          temp-cluster=Merge(cluster[r],cluster[l]); //聚簇操作生成的调度簇
          if(Confirm_Merge(temp-cluster)) //若判定聚簇有效
             cluster[r]←temp-cluster; //用生成的调度簇替代 cluster[r]
             cluster[r].room= cluster[r].room–序列长度; //计算聚簇后 cluster[r]的 CR 值
             cluster[l].resultflag=false; //将 luster[l]标记为删除
       else 标记 cluster[l]相对于 cluster[r]已处理; //若判定聚簇无效
        else cluster[r].roomflag=false，对 cluster[i]上 lst 值还未确定的节点置其 lst 值与 est 值相等;
    }while(have a result cluster's roomflag is true)
}//end of Search_Cluster
```

(b) Search_Cluster()函数的伪代码

```
Select_List(list[][])
{
    switch (list-num) //按照最大聚簇空间-最小聚簇序列策略选择参与聚簇的聚簇序列
       case 0: return 0;
       case 1: return list[0][]的序号;
       default:
          从 list[][]中找到 first-min (最小长度)、second-min(次小长度)和 max (最小长度);
          if(cluster[i].roomlength>=first-min+second-min)
             return first-min 在 list[][]中的序号;
          else return max 在 list[][]中的序号;
}//end of Select_List
```

(c) Select-List()函数的伪代码

```
Confirm_Merge(temp-cluster[])
{ //input: temp-cluster[]，即聚簇生成的调度簇
  //output: true(聚簇有效) / false(聚簇无效)
    for(j=0; j<le_r|; i++)
        for(p=1; p<length of temp-cluster[]; p++)
            flag1=false;
            if(e[j]是由节点 temp-cluster[p]出发的剩余通信) flag1=true; break;
        for(q=1;q<length of temp-cluster[]; q++)
            flag2=false;
            if(e[j]指向节点 temp-cluster [q]) flag2=true; break;
        if((flag1)and(!flag2))
            if(ect[temp-cluster[p]]+c[j]>est[↑e[j]])return false;
    return true;
}//end of Confirm_Merge
```

(d) Confirm_Merge()函数的伪代码

图 4

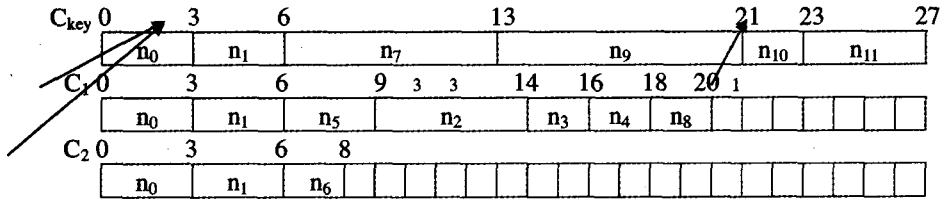
Confirm_Merge()函数，负责判定 Merge()产生的新调度簇是否导致 SCT 树的调度长度增加，即判定聚簇的有效性。其伪代码如图 4(d)。

定理 3 采用 SAS 算法对 OSA 调度进行优化并不增加原算法的时间复杂度。

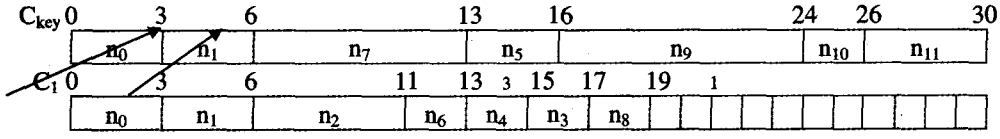
证明：SAS 算法的时间复杂性分析如下。Search_Cluster()所调用的函数中，Select_List()函数的时间复杂度为 $O(|V|)$，Merge()函数为 $O(|V|)$，Confirm_Merge()函数为 $O(|e_r|)$。记 d 为 DAG 图中节点的入度或出度的最大值，由于任一节点的剩余通信最多为 $d-1$ 个，则 $|e_r| \leqslant (d-1)|V|$，则 Confirm_Merge()函数的时间复杂度为 $O(d|V|)$。所以 Search_Cluster()的时间复杂度为 $O(md|V|)$，其中 m 为调度簇的数目，$|V|$ 为任务的数目。又因 $m \leqslant |V|$，则 Search_Cluster()的时间复杂度为 $O(d|V|^2)$。而 OSA_Schedule()函数的时间复杂度与 OSA 算法相同，为 $O(d^2|V|^2)$(见文[3])。因此，SAS_Optimize()的复杂度依然为 $O(d^2|V|^2)$。得证。

5 实验与分析

对于 SAS 算法，当 $SL>D$ 时，判定 DAG 图不可在周期 D 内调度。当 $D=SL$ 时，采用 SAS 算法的调度结果如图 5(a)，此时 $SL=27, m=3$。当 $D>SL$ 时，采用 SAS 算法的调度结果如图 5(b)，此时 $SL=30, m=2$。图 6 为 $SL=27, D=30$ 时 OSA 算法的对比调度结果。由图 2 和图 5(a)，及图 5(a)和图 6 的比较显示，SAS 算法可实现既定的调度目标。



(a) 采用 SAS 算法的调度结果(SL≈D=27)



(b) 采用 SAS 算法的调度结果(SL=27，D=30，令 SL=D=30)

图 5

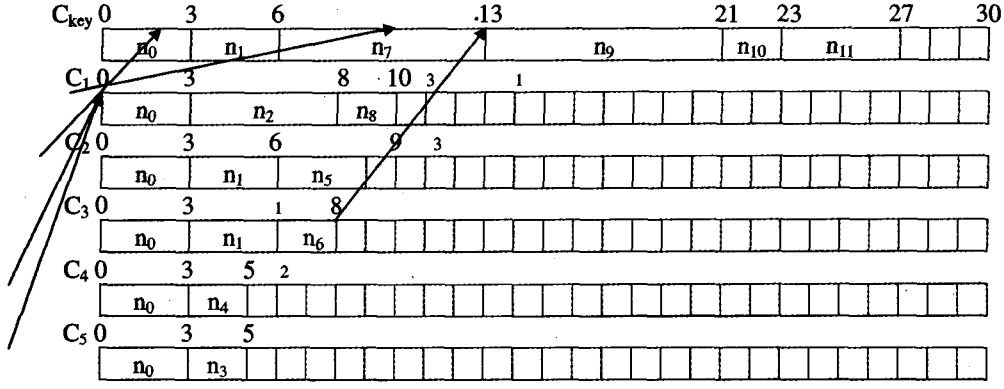


图 6 OSA 算法的调度结果(SL=27，D=30)

表 1　采用 SAS 算法对 OSA 调度的优化过程($D=30$, $SL=27$)

优化前 OSA 的调度结果:$C_{key}=\{n_0, n_1, n_7, n_9, n_{10}, n_{11}\}$, $C_1=\{n_0, n_1, n_5\}$, $C_2=\{n_0, n_1, n_6\}$, $C_3=\{n_0, n_2, n_8\}$, $C_4=\{n_0, n_3\}$, $C_5=\{n_0, n_4\}$,如图 6。
由于 $D>SL$,对于出口节点 $n11$,令 $ect_{C_{key}}(n_{11})=SL=D=30$。

步骤	确定 lct 值的叶子节点	确定的聚簇空间	聚簇空间最大的调度簇	最大聚簇空间在各调度簇上的 $CL_{3,key}$ 对应聚簇序列的长度	所选聚簇序列所在的调度簇	聚簇生成的调度簇	判定	操作
1	C_{key}	$CR_{key}=3$	C_{key}	$L(CL_{1,key})=3$, $L(CL_{2,key})=2$, $L(CL_{3,key})=7$, $L(CL_{4,key})=2$, $L(CL_{5,key})=2$	C_1	$C_{key}=\{n_0, n_1, n_7, n_5, n_9, n_{10}, n_{11}\}$	聚簇有效	标记 C_1 为删除, $CR_{key}=0$,标记为已处理。
2	C_2 C_3	$CR_2=5$ $CR_3=13$	C_3	$L(CL_{2,key})=27$, $L(CL_{2,3})=5$, $L(CL_{4,3})=2$, $L(CL_{5,3})=2$,	C_4	$C_3=\{n_0, n_2, n_3, n_8\}$	聚簇有效	标记 C_4 为删除, $CR_3=11$
3	C_2 C_3	$CR_2=5$ $CR_3=11$	C_3	$L(CL_{2,key})=27$, $L(CL_{2,3})=5$, $L(CL_{5,3})=2$,	C_5	$C_3=\{n_0, n_2, n_4, n_3, n_8\}$	聚簇有效	标记 C_5 为删除, $CR_3=9$
	C_2 C_3	$CR_2=5$ $CR_3=9$	C_3	$L(CL_{2,key})=27$, $L(CL_{2,3})=5$	C_2	$C_3=\{n_0, n_1, n_2, n_6, n_4, n_3, n_8\}$	聚簇有效	标记 C_2 为删除, $CR_3=4$
4	C_3	$CR_3=4$	C_3	$L(CL_{2,key})=27$	无	无		标记 CR_3 为已处理。

由于所有未标记删除的调度簇的聚簇空间都已处理,搜索结束。
优化后的调度簇为 $C_{key}=\{n_0, n_1, n_7, n_5, n_9, n_{10}, n_{11}\}$, $C_3=\{n_0, n_1, n_2, n_6, n_4, n_3, n_8\}$,并映射到 P_0、P_1,如图 5(b)。

表 2　OSA、PPA 和 SAS 算法的实验结果比较(当 $D=SL=27$ 和 $D=30$, $SL=27$ 时)

比较项目：算法 / DAG 图属性		预留处理器数目	预留处理器时间(1 周期)	实际占用时间(1 周期)	处理器平均利用率
当 $D=SL=27$ 时	OSA 算法	6	$6*27=162$	64	39.5%
	SAS 算法	3	$3*27=81$	55	67.9%
当 $D=30$, $SL=27$ 时	OSA 算法	6	$6*30=180$	64	35.6%
	SAS 算法	2	$2*30=60$	49	81.7%

SAS 算法在 $SL<D$ 与 $SL=D$ 时搜索-聚簇过程基本相同,所不同的是 $SL<D$ 时需要先执行 $lst_{C_{key}}(n_{end})=SL=D$ 的赋值。现以 $SL<D$ 时对 OSA 调度的优化为例说明其搜索-聚簇过程,见表 1。

与图 5(a)相比,图 5(b)经过优化后已经不存在因 $SL<D$ 而产生的处理器资源的浪费,这是因为本例中通过搜索-聚簇恰好使关键调度簇最终完全占满一个周期。在一般情况下,关键调度簇的 SL 将接近但不一定等于周期 D。

以下采用预留处理器数目、处理器平均利用率等指标比较三种算法的调度结果,见表 2。其中,处理器平均利用率为 DAG 图经调度后在各处理器上的实际占用时间与预留处理器时间的比值;预留处理器时间则为预留处理器数目与周期 D 的乘积。通过比较表明:SAS 算法总保持在预留处理器数目和处理器利用率上对于 OSA 算法的优越性,同时可满足 $SL\leqslant D$ 条件,即保证实时分布任务在任务周期内的调度执行,OSA 算法则未考虑 DAG 图的 dealine。

结论　本文提出了一种新的结构 SCT 树,在此基础上给出了一种基于 SCT 树的周期性分布实时任务调度算法(SAS)。本文的主要贡献是:1、对于 OSA 调度的有效处理,证明本文所提出的 SCT 树是一种有效的研究工具,适用于研究和优化基于任务复制调度算法的调度结果;2、针对周期性分布实时任务的调度,提出了"以调度长度不超过且最接近任务周期为主要目标,以所需处理器最少为次要目标"的调度思想;3、通过理论分析和实验验证,证明 SAS 算法在预留处理器数目和处理器利用率上总是优于 OSA 算法,同时其优化并不增加算法的复杂度。

参 考 文 献

1　Park C I, Choe T Y. An optimal scheduling algorithm based on task duplication: [Technical Report CS-SS- 2000-04]. Dept. of Computer Science and Eng., POSTECH, Aug. 2000

2　Darbha S, Agrawal D P. Optimal scheduling algorithm for distributed-memory machines. IEEE Trans on parallel and Distributed systems, 1998, 9(1): 87～95

3　Park C I, Choe T Y. An optimal scheduling algorithm based on task duplication. IEEE Trans on computers, 2002, 51(4): 444～448

4　Fang Ming, Yuan Youguang. A Post-Scheduling Optimization Algorithm for Distributed Real-Time Tasks based on Scheduled Cluster Tree. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies, Dalian, China, 2005. 1083～1085

5　Liu Zhenying, Fang Binxing, Zhang Yi, Tang Jianqi. Scheduling Algorithms for a Fork DAG in a NOWs. In: Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region

6　刘振英, 方滨兴, 姜誉, 张毅, 赵宏. 一个调度 Fork-Join 任务图的新算法. 软件学报, 2002, 13(4): 693～697

7　周双娥, 袁由光, 熊兵周, 欧中红. 基于任务复制的处理器预分配算法. 计算机学报, 2004, 27(2): 216～223

8　Ahmad I, Kwork Y K. On exploit task duplication in parallel program scheduling. IEEE Trans on parallel and distributed systems, 1998, 9(9): 872～892