ISSN1000-9825 Journal of Software 软件学报 2000,11(6):771~778

分布式存储的并行串匹配算法的设计与分析*

陈国良 林洁 顾乃杰

(中国科学技术大学计算机科学技术系 合肥 230027)

E-mail: glchen@mail.ustc.edu.cn

摘要 并行串匹配算法的研究大都集中在PRAM(parallel random access machine)模型上,其他更为实际的模型上的并行串匹配算法的研究相对要薄弱得多.该文采用将最优串行算法并行化的技术,利用模式串的周期性质,巧妙地将改进的KMP(Knuth-Morris-Pratt)算法并行化,提出了一个简便、高效且具有良好可扩放性的分布式串匹配算法,其计算复杂度为$O(n/p+m)$,通信复杂度为$O(u\log p)$,其中n为文本串长,m为模式串长,u为模式串最小周期长,p为处理器数.

关键词 串匹配,KMP(Knuth-Morris-Pratt),分布式算法,可扩放性.

中图法分类号 TP301

串匹配问题是计算机科学中研究得最广泛的问题之一,在文字编辑与处理、图像处理、信息检索、分子生物学等领域都有很广泛的应用.所谓串匹配是指,给定一个长为n的正文串$T[1,n]$和长为m的模式串$W[1,m]$,t_i($1\leqslant i\leqslant n$)和m_j($1\leqslant j\leqslant m$)$\in\Sigma$(Σ为字符集),找出模式串W在正文串T中所有成功匹配的起始位置.

串行串匹配算法有很多,其中最著名的有KMP(Knuth-Morris-Pratt)[1]算法,它的时间复杂度为$O(m+n)$.80年代以来,并行串匹配算法的研究主要集中在PRAM(parallel random access machine)模型上,直到90年代才出现了一些较好的基于分布式存储模型的并行算法[2].例如,1996年F. Moussouni和C. Lavault[3]提出了N-cube上的分布式串匹配算法,其计算时间复杂度为$O(n/p+m)$,通信复杂度为$O(\log p+m)$,其中p为处理器个数.本文采用将最优串行算法并行化的技术,利用模式串的周期性质,巧妙地将改进的KMP算法并行化,提出了一个简便、高效且具有良好可扩放性的分布式串匹配算法,其计算复杂度为$O(n/p+m)$,通信复杂度为$O(u\log p)$,其中u为模式串最小周期的长度,p为处理器个数.值得一提的是,算法[3]对通信复杂度的分析是基于特定的模型并采用流水线和扩展树通信技术.考虑到本算法适用于各种分布式环境,采用算法[3]的通信技术并不合适,所以本算法对通信的处理和分析更加符合实际.

为方便叙述,以下约定大写字母表示字符串,A^b表示字符串A作b次连接后所得到的字符串,$|A|$表示字符串A的长度.本文第1节介绍算法设计.第2节是算法分析.第3节给出曙光1000并行机上的实验数据及分析.第4节是结论.

1 算法设计

1.1 改进的KMP算法

KMP算法[4]的关键是根据给定的模式串$W[1,m]$定义一个Next函数.Next函数包含了模式串本身局部匹配的信息.Next函数的定义如下:

* 本文研究得到国家教育部博士点基金(No. 9703825)资助.作者陈国良,1938年生,教授,博士生导师,主要研究领域为并行算法,并行体系结构,并行与分布式计算.林洁,女,1973年生,硕士,主要研究领域为并行算法,并行与分布式计算.顾乃杰,1961年生,副教授,主要研究领域为并行算法,并行与分布式计算.

本文通讯联系人:陈国良,合肥 230027,中国科学技术大学计算机科学技术系

本文1999-01-11收到原稿,1999-06-17收到修改稿

$$next(j)=\begin{cases}0, & j=1,\\ \max\{k\mid 1<k<j, & \text{使得 } W[1,k-1]=W[j-(k-1),j-1]\},\\ 1, & \text{对所有 } k,1<k<j,W[1,k-1]\neq W[j-(k-1),j-1].\end{cases}$$

KMP 算法的基本思想是:假设在模式匹配的进程中执行 $T[i]$和 $W[j]$的匹配检查.若 $T[i]=W[j]$,则继续检查 $T[i+1]$和 $W[j+1]$是否匹配.若 $T[i]\neq W[j]$,则分成两种情况:若 $j=1$,则模式串右移一位,检查 $T[i+1]$和 $W[1]$是否匹配;若 $1<j\leqslant m$,则模式串右移 $j-next(j)$位,检查 $T[i]$和 $W[next(j)]$是否匹配.重复此过程,直到 $j=m$ 或 $i=n$ 结束.显然,KMP 算法只能找到第 1 个匹配位置,如何找到所有的匹配位置呢?当找到一个匹配位置时,是否也可以做到尽量利用已经获得的匹配结果,使得正文串指针不必回溯,而且模式串可以右移尽可能大的一段距离呢?文献[1]给出了肯定的答案.将模式串的 Next 函数的定义扩展到 $next((m+1),next(m+1)$的定义与 $next(j)(1<j\leqslant m)$的定义相同,找到一个匹配位置时将文本串指针 i 右移一位,模式串指针 j 移到 $next(m+1)$,即检查 $T[i+1]$和 $W[next(m+1)]$是否匹配.

在文献[4]中,朱洪对 KMP 算法作了修改.他修改了 KMP 算法中的 Next 函数,即求 Next 函数时不但要求 $W[1,next(j)-1]=W[j-(next(j)-1),j-1]$,而且要求 $W[next(j)]\neq W[j]$,记修改后的 Next 函数为 Newnext.显然,在模式串字符重复高的情况下,朱洪的 KMP 算法比原 KMP 算法更加有效.

另外,我们发现在 KMP 算法结束时,模式串指针 $j-1$ 的值就是文本串尾模式串最大前缀串的长度.这个发现使得我们可以丝毫不增加时间复杂度而找到此最大前缀串的长度.这对于分布式并行算法的设计有很重要的意义,我们将在后面的并行算法设计部分讨论这一点.以下给出改进的 KMP 算法、Next 函数和 Newnext 函数的计算算法.

在算法 1 中,当内 while 循环遇到成功比较并找到文本串中模式串的一个匹配位置时,文本串指针 i 均加 1,所以至多作 n 次比较;内 while 循环每次不成功比较时文本串指针 i 保持不变,但是模式串指针 j 减小,所以 $i-j$ 的值增加且上一次出内循环时的 $i-j$ 值等于下一次进入时的值,因此,不成功的比较次数不大于 $i-j$ 的值的增加值,即小于 n,所以总的比较次数小于 $2n$,故算法 1 的时间复杂度为 $O(n)$.朱洪在文献[4]中证明了计算 Next 函数和 Newnext 函数的算法时间复杂度为 $O(m)$,本文的算法 2 只比朱洪的算法多计算 $next(m+1)$,至多多作 $m-1$ 次比较,所以,算法 2 的时间复杂度仍为 $O(m)$.

算法 1. KMP 串匹配算法

输入:正文串 $T[1,n]$和模式串 $W[1,m]$

输出:匹配结果 $match[1,n]$

```
procedure KMP
begin
    i=1
    j=1
    while i<=n do
        while j! =0 and W[j]! =T[i] do
            j=newnext[j]
        endwhile
        if j=m
            match[i-(m-1)]=1
            j=next[m+1]
            i++
        else
            j++
            i++
        endif
    endwhile
    max_prefix_len=j-1
end
```

算法 2. Next 函数和 Newnext 函数的计算算法

输入:模式串 $W[1,m]$

输出:$next[1,m+1]$和 $newnext[1,m]$

```
procedure NEXT
begin
    next[1]=newnext[1]=0
    j=2
    while j<=m+1 do
        i=next[j-1]
        while (i! =0 and W[i]! =W[j-1]) do
            i=next[i]
        endwhile
        next[j]=i+1
        if j! =m+1
          if W[j]! =W[i+1]
            newnext[j]=i+1
          else
            newnext[j]=newnext[i+1]
          endif
        endif
        j++
    endwhile
end
```

1.2 分布式串匹配算法

分布式串匹配算法的设计思路是,将长为 n 的文本串 T 均匀划分成互不重叠的 p 段,分布于处理器 $0\sim(p-1)$中,且使相邻的文本段分布在相邻的处理器中,显然每个处理器中局部文本段的长度为$\lceil n/p\rceil$(最后一个处理器可在其段尾补上其他特殊字符,使其长度为$\lceil n/p\rceil$).再将长为 m 的模式串 W 和模式串的 Newnext 函数播送到各处理器中.各处理器使用 KMP 算法并行地对局部文本段进行匹配,找到所有段内匹配位置.但是,每个局部段($p-1$ 段除外)段尾 $m-1$ 字符中的匹配位置必须跨段才能找到.一个简单易行的办法就是每个处理器(处理器 $p-1$ 除外)将本局部段的段尾 $m-1$ 个字符传送给下一处理器,下一处理器接收到前一处理器传来的字符串后,再结合本段的段首 $m-1$ 个字符构成一个长为 $2(m-1)$的段间字符串,对此字符串作匹配,就能找到所有段间匹配位置.但是,这样做通信量太大.文献[3]的算法提出,每个处理器在其段尾 $m-1$ 个字符中找到模式串 W 的最长前缀串,如图 1 所示.因为每个处理器上都有模式串信息,所以只需传送此前缀串的长度 max$prefixlen$ 就行了,这样就大大降低了通信量.另外,能否进一步降低播送模式串和 Newnext 函数的通信复杂度呢?我们设想利用串的周期性质,先对模式串 W 作预处理,获得其最小周期长度 $|U|$、最小周期个数 s 及后缀长度 $|V|$ $(W=U^sV)$,只需播送 U,s 和 $|V|$ 以及部分 Newnext 函数就可以了,从而大大减少了播送模式串和 Newnext 函数的通信量.

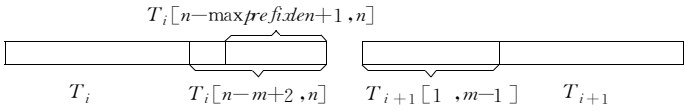


Fig. 1 Match between segments

图1 段间匹配示意图

1.2.1 串的周期性分析

串周期分析算法对于我们的并行算法设计非常关键,我们发现,串的最小周期和 Next 函数之间的关系存在着如定理 1 所示的简单规律,使得我们能设计出时间复杂度为常数的串周期分析算法.

定义 1. 引用文献[3]的周期定义,给定串 W,如果存在字符串 U 以及正整数 $k\geqslant 2$,使得串 W 是串 U^k 的前缀,则称字符串 U 为串 W 的周期.在字符串 W 的所有周期中长度最短的周期称为串 W 的最小周期.

定理 1. 已知串 W 长为 m,记 $u=m+1-next(m+1)$,则 u 为串 W 的最小周期长度.

证明:(1) 先证 u 是串 W 的周期长度.

由 $next(m+1)$的定义,有 $W[1,next(m+1)-1]=W[m-next(m+1)+2,m]$,即 $W[1,next(m+1)-1]=W[u+1,m]$,如图 1 所示,所以有 $W[1,u]=W[ku+1,(k+1)u](k=1,2,\ldots,[m/u]-1)$且 $W[[m/u]u+1,m]=W[([m/u]-1)u+1,next(m+1)-1]=W[1,m-[m/u]u]$,所以按照定义 1,$W[1,u]$是串 W 的周期.

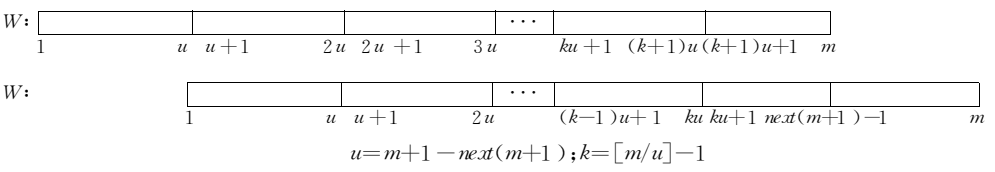


Fig. 2 Demonstration of Lemma 1's proof

图2 定理1证明示意图

(2) 再证 u 是串 W 的最小周期长度.

用反证法.假设串 W 的最小周期长度为 t,而且 $t<u$,则由定义 1 有,$W[1,t]=W[kt+1,(k+1)t](k=1,2,\ldots,[m/t]-1)$且 $W[[m/t]t+1,m]=W[1,m-[m/t]t]$,所以有 $next(m+1)=m+1-t$,显然 $m+1-t>m+1-u$,这与 $next(m+1)$的定义矛盾,所以 u 是串 W 的最小周期长度.

由上述(1)和(2),定理 1 得证. □

算法 3. 串周期分析算法

输入:$next[m+1]$

输出:最小周期长度 $period\_len$,最小周期个数 $period\_num$,模式串的后缀 $pattern\_suffixlen$

```
procrdure PERIOD_ANALYSIS
begin
    period_len=m+1-next(m+1);
    period_num=(int)m/period_len;
    pattern_suffixlen=m mod period_len;
end
```

1.2.2 **算法描述**

算法 4. 分布式串匹配算法

输入:分布存储的文本串 $T_i[1,[n/p]](i=0,2,p-1)$ **和存储于** PE_0 **的模式串** $W[1,m]$

输出:匹配结果

```
begin
(1) PE0 call procedure NEXT                 /* 处理器 PE0 求模式串 W 的 Next 函数和 Newnext 函数 */
    PE0 call procedure PERIOD_ANALYSIS      /* 处理器 PE0 对模式串 W 进行周期分析 */
(2) PE0 broadcast period_len,period_num,period_suffixlen
                                            /* 播送模式串最小周期长度,最小周期个数,后缀长度 */
    PE0 broadcast W[1,period_len]           /* 播送模式串的最小周期 */
    if period_num=1                         /* 播送模式串的部分 Newnext 函数,如果周期数为 1,则播送整个 Newnext 函数;否则,播送两倍周期长度的 Newnext 函数 */
      broadcast newnext[1,m]
    else
      broadcast newnext[1,2*period_len]
    endif
(3) for i=1 to p-1 par_do                   /* 由传来的模式串周期和部分 Newnext 函数重构整个模式串和 Newnext 函数 */
      call procedure REBUILD
    endfor
    for i=0 to p-1 par_do                   /* 各处理器调用过程 KMP 作局部段匹配,并获得局部段尾最大前缀串的长度 */
      KMP(Ti,W,n,0,match)
    endfor
(4) for i=0 to p-2 par_do                   /* 处理器 PEi 把 maxprefixlen 发送给处理器 PEi+1 */
      PEi send maxprefixlen from PEi+1
    endfor
    for i=1 to p-1 par_do                   /* 处理器 PEi 接收 PEi-1 发送来的 maxprefixlen,调用 KMP 作段间匹配 */
      PEi receive maxprefixlen from PEi-1
      PEi call KMP (Ti,W,m-1,maxprefixlen,match'+m-1)
    endfor
end
```

用 KMP **算法作段间匹配时,因为已经匹配了** max*prefixlen* **长度的字符串,所以模式串指针只需从** max*prefixlen*+1 **处开始.对** KMP **算法再作如下修改.**

```
procedure KMP(T,W,textlen,matched_num,match)
begin
  i=1
  j=matched_num+1
  while i<=textlen
      while j! =0 and W[j]! =〈〉T[i] do
          j=newnext[j]
      endwhile
      if j=m
```

```
                match[i-(m-1)]=1
                j=next[m+1]
                i++
            else
                j++
                i++
            endif
        endwhile
        max prefixlen=j-1
    end
```

2 算法分析

算法的时间复杂度包含计算时间复杂度和通信时间复杂度两个方面.在分析计算时间复杂度时,假定文本串初始已经分布在各个处理器上,这是合理的,因为文本串一般很大,不会有大的变化,只需分布一次就可以,同时也假定结果分布在各处理器上.本算法的计算时间由算法步骤(1)中算法 2 的时间复杂度 $O(m)$和算法 3 的时间复杂度 $O(1)$,算法步骤(3)和算法步骤(4)的改进的 KMP 算法的时间复杂度 $O(n/p)$和 $O(m)$构成.所以,本算法总的计算时间复杂度为 $O(n/p+m)$.通信复杂度由算法步骤(2)播送模式串信息(最小周期串 U 及最小周期长度、周期个数和后缀长度 3 个整数)和 $Newnext$ 函数(长度为 $2u$ 的整数数组,u 为串 U 的长度)以及算法步骤(4)传送最大前缀串长度组成,采用二分树通信技术,所以总的通信复杂度为 $O(u\log p)$.

3 实验结果

我们在国家高性能计算中心(合肥)的大规模并行机曙光 1000 上测试了上面的算法.我们测试了模式串长为 128K 和周期串长为 4K,16K,64K 的 3 组数据,每组数据的处理器数分为 6 种情况:1,2,4,8,16 和 31,文本串长分为 6 种情况,即:28M,56M,112M,224M,448M 和 868M.下面是对这些数据的分析.

3.1 计算时间的分析

利用曲线拟合分别拟合了周期串长为 4K,16K,64K 的数据,得到的 3 个方程分别为:

(1) 周期长为 4K:

$$t(n,p)=0.631076n/p+2.41241/p+12.4404\times10^{-6}n-0.2106.$$

(2) 周期长为 16K:

$$t(n,p)=0.63114n/p+2.41422/p+8.32277\times10^{-6}n-0.214093.$$

(3) 周期长为 64K:

$$t(n,p)=0.631078n/p+2.42302/p+6.14216\times10^{-6}n-0.228162.$$

曲线拟合的误差(定义为:(拟合值-测量值)/测量值)曲线如图 3 所示:

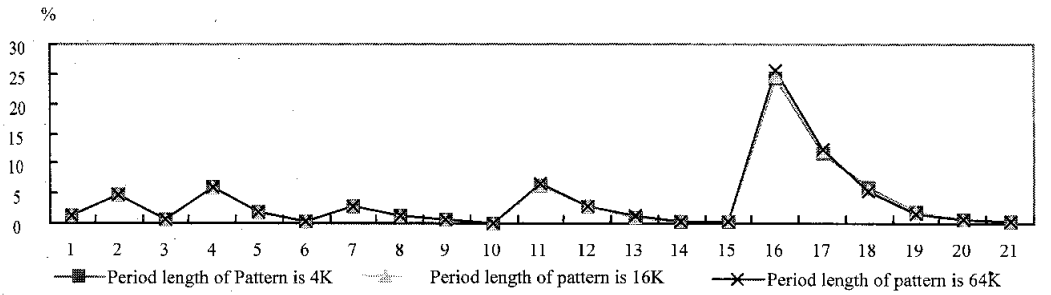


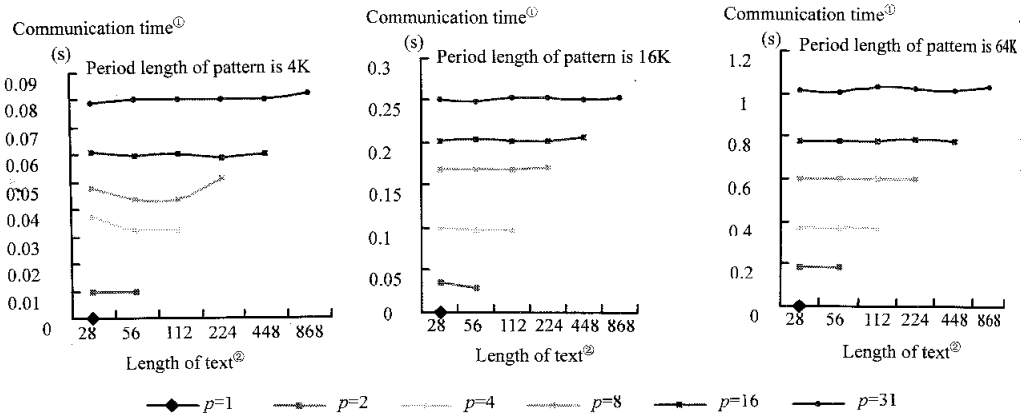
Fig. 3 Error curve of computational time complexity

图 3 计算时间复杂度曲线拟合的误差曲线

从上面的误差曲线可以看出,绝大部分的误差在5%之内,只有两个地方的误差大于10%,其原因是在这两个点本身的计算时间就很小,这样,在进行测量时,由于UNIX系统上的时间单位比较大和测量时的其他误差导致的误差可能会比较大,总的来说,拟合出来的结果和测量数据吻合得很好,而这3个拟和公式与我们理论推导出来的计算时间$O(n/p+m)$是一致的(由于固定了m,所以拟合公式中没有m出现).另外,上面3个拟合公式所对应的系数相差非常小,这说明计算时间和模式的周期长几乎没有关系.

3.2 通信时间的分析

根据实验结果,我们给出了不同周期、不同处理器数和不同文本串长情况下的通信时间曲线.从图4可以看出,通信时间只与模式串最小周期串长和处理器数有关.

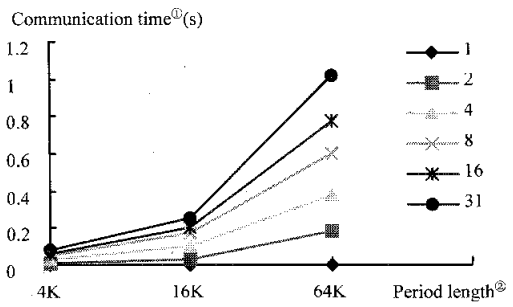


①通信时间,②文本串长.

Fig. 4 Curve of communication time

图4 通信时间曲线

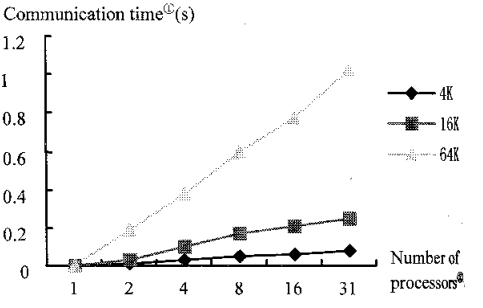
我们对每一组周期串长和处理器数的组合求出平均通信时间,画出它们与周期和处理器数的关系图,如图5和图6所示.



①通信时间,②周期长.

Fig. 5 Relationship between communication time and period

图5 通信时间与周期的关系



①通信时间,②处理器数.

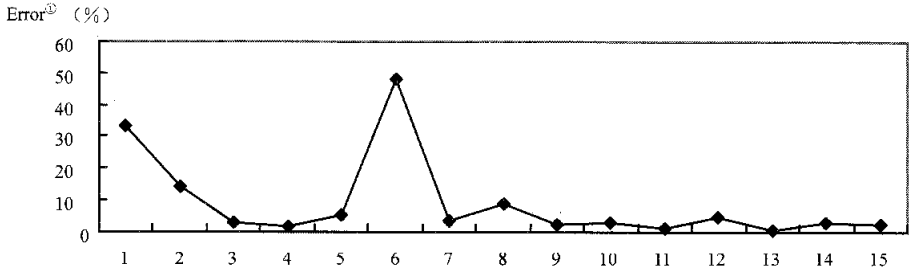
Fig. 6 Relationship between communication time and number of processors

图6 通信时间与处理器关系

然后再利用曲线拟合,可以得到通信时间与周期和处理器数的关系式:

$$comm(u,p)=0.00452519u\ln p+0.00479584\ln p-0.000242529u-0.00156526,$$

误差曲线如图7所示.



①误差,②|U|和 p 的组合.

Fig. 7 Error curve of communication time

图 7 通信复杂度拟合误差曲线

从图 7 中可以看出,绝大部分的误差小于 10%,只有两个点误差比较大,其原因同样在于这两个地方本身的通信时间很小,所以测量误差可能较大. 上面的拟合公式和理论推导出来的公式 O(u*log*p)相吻合. 从上面的分析和实验结果可以看出,本算法的通信时间只和模式串的周期相关,而与文本串、模式串长无关.

3.3 可扩放性分析

Kumar 在文献[5]中提出了等效率函数(*iso-efficiency function*)这种度量并行算法可扩放性的标准,指出对于给定的算法,如果计算量保持固定,则效率随着处理器数的增加而减小;为了维持效率为某一定值,即等效率(*iso-efficiency*),则在处理器数增多的同时,计算量亦必须按比例增大才行. 等效率函数定义为维持效率不变时计算量随处理机数增长的模式. 如果此模式为线性的,则算法是线性可扩放的;如果是亚线性的,则算法是可扩放的;如果是指数的,则算法是不可扩放的,因为此时当处理器数增大时,为了维持效率不变,工作量必须急剧增大,而这是并行系统所无法承受的.

由于我们所关注的主要是计算量、机器数目和效率之间的关系,所以在下面的分析中我们不考虑模式串,即把模式串长和周期看成是常数. 从上面的曲线拟合中(忽略一些小的项)可以得到计算时间和通信时间分别为:

$$t(n,p)=0.63n/p+2.41/p-0.21,$$

$$comm(u,p)=0.0045u\mathit{ln}p+0.0048\mathit{ln}p-0.0016.$$

根据效率 E 的定义可以得到(忽略了模式串长),

$$E=\frac{n}{p(t(n,p)+comm(u,p))}=\frac{n}{p(0.63n/p+2.41/p-0.21+0.045u\mathit{ln}p+0.0048\mathit{ln}p-0.0016)},$$

于是

$$n=\frac{E}{1-0.63E}(0.45up\mathit{ln}p-0.21p+2.41).$$

从上述公式中可以看出工作量与 p*ln*p 是成比例的,所以,该算法是可扩放的,但不是线性可扩放的.

4 结 论

在分布式存储系统中,由于文本串的分布存储和模式串的局部存储,使得并行串匹配算法必须在尽可能降低通信开销的情况下处理段间匹配. 本文给出的算法很好地解决了这个问题,不仅通信复杂度很小,计算复杂度也达到了最优,而且还具有较好的可扩展性.

参考文献

1 Knuth D E, Morris J H, Pratt V R. Fast pattern matching in strings. SIAM Journal of Computing, 1997,6(2):323~350

2 Chen Guo-liang. Design and Analysis of Parallel Algorithm. Beijing: Higher Education Press, 1994
(陈国良. 并行算法的设计和分析. 北京:高等教育出版社,1994)

3 Moussouni F, Lavault C. Distributed string matching algorithm on the N-cube. Lecture Notes in Computer Science 1123, Euro-par'96 Parallel Processing, 1996

4 Zhu Hong, Chen Zeng-wu, Duan Zhen-hua *et al*. Design and Analysis of Algorithm. Shanghai: Shanghai Science and

Technology Press, 1989. 132～135
(朱洪,陈增武,段振华等. 算法设计和分析. 上海:上海科技文献出版社,1989. 132～135)
5 Kumar V, Rao V N. Parallel depth-first search, Part two: analysis. International Journal of Parallel Programming, 1987, 16(6):501～519

Design and Analysis of String Matching Algorithm on Distributed Memory Machine

CHEN Guo-liang LIN Jie GU Nai-jie

(*Department of Computer Science and Technology University of Science and Technology of China Hefei* 230027)

Abstract Parallel string matching algorithms are mainly based on PRAM (parallel random access machine) computation model, while the research on parallel string matching algorithm for other more realistic models is very limited. In this paper, the authors present an efficient and scalable distributed string-matching algorithm is presented by parallelizing the improved KMP (Knuth-Morris-Pratt) algorithm and making use of the pattern period. Its computation complexity is $O(n/p+m)$ and communication time is $O(u\log p)$, where n is the length of text, m the length of pattern, p the number of processors and u the period length of pattern.

Key words String match, KMP (Knuth-Morris-Pratt), distributed algorithm, scalability.