低年级讨论班

第10章 寻求快速的种群计数

数学科学学院 杨嘉骐

种群计数（pop count）

- 种群计数（population count或简称为popcount）是一个非常基本的算法，其目的是对一个变量求其二进制表示中“1”的个数。
- 在信息论与编码理论中，这个值也被叫做Hamming weight（汉明重量）.汉明距离就是就是求两个编码异或之后的Hamming weight。这个算法在某些编码过程中会被大量重复使用，所以有必要进行精细的改进。
- 本文主要针对指令系统里不含此命令的RISC（精简指令集计算机）处理器系统。

本文内容梗概

- 平凡的算法
- 对算法的改进
- 算法的应用

本文要解决的目标问题非常简明，最平凡的算法易于想到，作者对算法进行反复且多角度优化，告诉我们最简单的算法也大有改进空间，并能反映算法设计者的巧思。

最平凡的算法

```
pop = 0;
  for (i = 0; i < 32; i++){
    if (x & 1) pop = pop + 1;
    x = x >> 1;
  }
```

32位机器中C语言下一个无符号整数由32位二进制码存储，最简单的思路是每次判定最末一位是否是1，然后右移一位继续比较。经过32次循环计算出结果。每一循环被编译成大约7条指令。

最平凡的算法

```
pop = 0;
  for (i = 0; i < 32; i++){
    if (x & 1) pop = pop + 1;
    x = x >> 1;
  }
```

为方便和后来的算法比较我们只统计算术运算（包括比较）的次数，我们认为x的二进制表示每位为0和1的概率相等，则每一循环经过1.5次add，1次and，1次shift，2次比较，那么平均需要耗费32 ×5.5=176次运算

第一次改进

```
pop = 0;
while (x) {
    pop = pop + (x & 1);
    x = x >> 1;
}
```

一般来说处理的数字都不会太大，二进制表示下前面的0很多，所以当右移至x各位都是0的时候（这时x值也为0）就可以跳出循环了。另一个改进是去掉比较过程， x & 1为false的时候其值为0，正好对pop的值没有影响。所以直接加上就可以了。即使按上个算法假设也只有4×32=128次算术运算，如果值比较小的话运算数目减少更加明显。

从缩短循环次数入手

```
pop = 0;
while (x) {
    pop = pop + 1;
    x = x & (x - 1);
}
```

本算法并不那么直观，但是改进效果还是明显的，算法主要目标是减少循环次数。
x & (x - 1)的作用是把其最末尾的一个“1”位清零。每一次循环减少一个“1”，总计算量与“1”的个数成正比，如果按照每个数位上0和1出现概率相同计算需要耗费
32×0.5×4 = 64次算术运算，大大降低了运算量，但是若以数位长度N来估计复杂度的话仍然是O（N）

时间代价最小的查表法

```
static char table[256] = {0, 1, 1, 2, 1, 2, 2, 3, ..., 8};
pop = table[x & 0xFF] + table[(x >> 8) & 0xFF]
+table[(x >> 16) & 0xFF] + table[x >> 24];
```

每8位为一个单元，预先计算好2的8次方共256种可能的情形每一种的pop值，然后将这4个值相加。

时间代价最小的查表法

本算法采用了牺牲空间来达到时间代价最大化的目的，只需要执行10次移位、求与和查表运算，但是由于所用空间较大不能像其他几种算法一样在寄存器中运算，不过由于数组固定且不是很大所以在缓存中存储的话还是可以达到很高的效率。

Divide and conquer is an important technique that should be near the top of every programmer's bag of tricks.

——Henry S. Warren, Jr.

生活中的例子

- 早晨一个女生背着一堆书进了阅览室,结果警报响了.大妈让女生看看是哪本书把警报弄响的,结果那女生把书倒出来,准备一本一本的测.大妈见状急了,把书分成两份,第一份过了一下,响了,她又把这一份又分成两份接着测, 三回就找到了那本书.大妈又用鄙视的眼光看着那个女生,仿佛在说:O(n)和O(log2n)都分不清……

——转载自某同学校内日志

分治法

- 字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

——Wikipedia

- 分治法有希望把将原来的 $O(N^n)$ 复杂度降低到 $O(N^{n-1}\log_2 N)$ ，同时便于并行算法的实现。

- 很多著名算法就是采用了分治法的思想，比如二分查找、快速排序（C.A. R. Hoare）、归并排序（John Von Neumann）、快速傅里叶变换（ Cooley & Tukey ）都是著名的快速算法。

原理其实非常直观

- 由于二进制表示的特殊性，我们只需要把每一位的0和1看成十进制下的0和1，然后把每一位都加起来就能得到最终的结果。这和前面查表法的原理是一样的。
- 运用分治法解决多个数的加法就可以达到优化算法的目的。

多数相加的分治算法

- 求A+B+C+D

0A0C = ABCD >> 2 & 0101

\+ 0B0D = ABCD & 0101

EFGH

00EF = EFGH >> 2 & 0011

\+ 00GH = EFGH & 0011

I J KL

在本算法中A+B=EF，C+D=GH，EF+GH=IJKL，所以算法可以完成运算，加法运算次数为log2n，同时利用了原值的空间没有额外的空间开销。

这里的字母既可代表一位也可代表多位

数字变形采取移位和掩码处理造成的额外运算速度开销较小。

C语言实现

- x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
  x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
  x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);

0x55555555即为8段的0101， 0x33333333为8段的0011， 0x0F0F0F0F为4段的00001111，0x00FF00FF为2段的0000000011111111，0x0000FFFF为前16位为0后16位为1。

总共需要需要5次移位，10次求与，5次加法共20次算术运算。

进一步优化

```
int pop(unsigned x) {
    x = (x & 0x55555555) + ((x  >> 1) & 0x55555555);
    x = x - ((x >> 1) & 0x55555555);

    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x33333333) +  ((x >> 2) & 0x33333333);

    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x + (x >> 4)) & 0x0F0F0F0F;

    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = x + (x >> 8);

    x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
    x = x + (x >> 16);

    return x & 0x0000003F;
}
```

进一步优化

```
int pop(unsigned x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2)
& 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x0000003F;
}
```

共需要 5次移位, 4次求与, 4次加法, 1次减法，共14次算数运算。减少了6次算术运算。但是程序的整体结构被破坏的较为严重。

可以更进一步优化

```
int popcount_mult(unsigned x) {
    x -= (x >> 1) & 0x55555555;
    x = (x & 0x33333333) + ((x >> 2)
    & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    return (x * 0x01010101)>>24;
    }
```

如果计算机执行乘法较快的话可以考虑用乘法代替加法，多于的高位自动溢出，共总共需要 4 次移位, 4次求与, 2次加法，1次减法，1次乘法共12次算术运算。

而且本算法易于推广至64位整数，只需要改变最后的右移位数即可。

HAKMEN169

HAKMEN是1972年由MIT人工智能实验室发表的一本算法备忘录，包含那个时代大量的优秀算法，主要使用汇编语言写成，共191个算法。其中第169算法与种群计数有关，可以参阅如下网址：

http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html

HAKMEN169

```
int pop(unsigned x) {
  unsigned n;

  n = (x >> 1) & 033333333333;
  x = x – n;
  n = (n >> 1) & 033333333333;
  x = x - n;
  x = (x + (x >> 3)) &
030707070707;
  return x%63;
}
```

算法的主要思想仍然是分治法，和前一个算法的区别有两点。

第一点是第一步的时候是每位一组，相邻三组相加。比如说对于

x=(abc)=4*a+2*b+c，

x>>1&常数1= 2*a+b，

x>>2&常数2=a，

于是 x - （x>>1&常数1 ）

- （x>>2&常数2） = a+b+c

HAKMEN169

```
int pop(unsigned x) {
  unsigned n;

  n = (x >> 1) & 033333333333;
  x = x – n;
  n = (n >> 1) & 033333333333;
  x = x - n;
  x = (x + (x >> 3)) &
030707070707;
  return x%63;
}
```

第二点是第二步做完之后，这时是每6位成一组，以12位整数（2组）为例，x=(ab)64 = a*64 + b。很明显，a*64+b ≡ a+b(mod 63)。模63就可以得到a+b。

总共需要3次移位, 3次求与, 2次加法, 1次乘法, 1次取模共10次算术运算。

不过取模运算速度较慢易成为瓶颈，所以在实际测试中并没有表现出特别快的速度。

不太实用的方法

$$\mathrm{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i)$$

其中加法靠不断对字长取模进行，需要62次算术运算，不能对非2的n次方字长的字进行运算。

Wikipedia上的测试结果

- popcount_naive: 1.6666e+07 iters in 688 msecs for 41.28 nsecs/iter

//原始算法

- popcount_8: 1e+08 iters in 995 msecs for 9.95 nsecs/iter
- popcount_6: 2e+08 iters in 1725 msecs for 8.625 nsecs/iter
- popcount_hakmem: 2e+08 iters in 1411 msecs for 7.055 nsecs/iter

//HAKMEM算法，明显被取模耽误时间

- popcount_keane: 1e+09 iters in 6566 msecs for 6.566 nsecs/iter
- popcount_3: 1e+09 iters in 6270 msecs for 6.27 nsecs/iter
- popcount_2: 1e+09 iters in 5658 msecs for 5.658 nsecs/iter

//改进算法

- popcount_mult: 1e+09 iters in 4169 msecs for 4.169 nsecs/iter

//带乘法改进算法

- 转载自http://wiki.cs.pdx.edu/forge/popcount.html
- 使用算法源代码参见http://en.wikipedia.org/wiki/Hamming_weight

数组中的种群计数

- 最平凡的方法：把数组中的每个值分别进行种群计数，然后相加。
- 基于原算法的改进：由于数组中连续存储的二进制码相当于对一个32n位的数进行种群计数，于是发挥原算法的空间极限对每3个字为一组，在每三个4位/8位/16位的掩码结果相加。

基于CSA的一个算法

- 本算法借鉴了逻辑电路的设计思想，利用CSA（carry-save adder）保留进位加法器的设计思路，完成快速处理一个数组内种群计数总和的问题。
- 核心是定义这样两个输出值:高位输出h，低位输出l

h=(a|b)&(b|c)&(c|a)=(a|b)&(a&b)|c=a|b&(a^b)|c

l=(a^b)^c

基于CSA的一个算法

```
#define CSA(h,l, a,b,c) \
   {unsigned u = a ^ b; unsigned v = c; \
      h = (a & b) | (u & v); l = u ^ v;}
int popArray(unsigned A[], int n) {
   int tot, i;
   unsigned ones, twos;
   tot = 0;                    // Initialize.
   ones = 0;
   for (i = 0; i <= n - 2; i = i + 2) {
      CSA(twos, ones, ones, A[i], A[i+1])
      tot = tot + pop(twos);
   }
   tot = 2*tot + pop(ones);
   if (n & 1)                  // If there's a last
one,
      tot = tot + pop(A[i]);    // add it in.
   return tot;
}
```

对于3个32位字，对其同一数位上的3个二进制码构成的三元组（a,b,c）

若有3个“1”则h=1，L=1;

2个“1”则h=1，L=0;

1个“1”则h=0, L=1;

0个“1”则h=0, L=0;

故三个数在同一位的种群计数值为2h+L

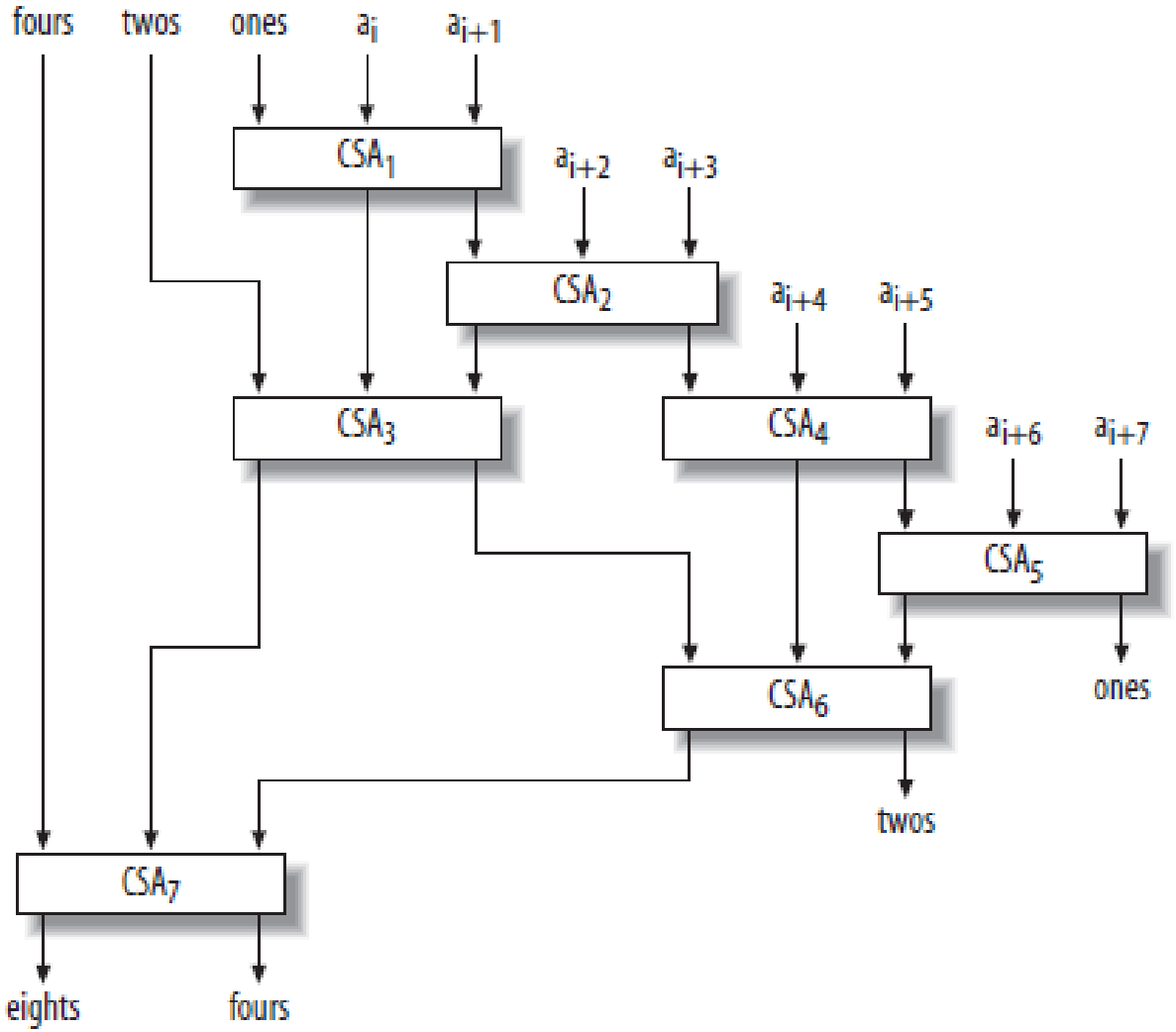
```
a = 0110 1001 1110 0101   9
b = 1000 1000 0100 0111   6
c = 1100 1010 0011 0101   8
---------------------------
l = 0010 1011 1001 0111   9
h = 1100 1000 0110 0101   7*2 = 14
```

基于CSA的一个算法

```
int popArray(unsigned A[], int n) {
   int tot, i;
   unsigned ones, twos, twosA, twosB,
      fours, foursA, foursB, eights;
   tot = 0;                    // Initialize.
   fours = twos = ones = 0;
   for (i = 0; i <= n - 8; i = i + 8) {
      CSA(twosA, ones, ones, A[i], A[i+1])
      CSA(twosB, ones, ones, A[i+2], A[i+3])
      CSA(foursA, twos, twos, twosA, twosB)
      CSA(twosA, ones, ones, A[i+4], A[i+5])
      CSA(twosB, ones, ones, A[i+6], A[i+7])
      CSA(foursB, twos, twos, twosA, twosB)
      CSA(eights, fours, fours, foursA, foursB)
      tot = tot + pop(eights);
   }
   tot = 8*tot + 4*pop(fours) + 2*pop(twos) +
pop(ones);
   for (i = i; i < n; i++)      // Simply add in the last
      tot = tot + pop(A[i]);    // 0 to 7 elements.
   return tot;
}
```

上面的方法其实是CSA的串行方法，也可以一次读入多位进行并行设计。

基于CSA的一个算法


fours
twos
ones
a_i
a_{i+1}
CSA_1
a_{i+2}
a_{i+3}
CSA_2
a_{i+4}
a_{i+5}
CSA_3
CSA_4
a_{i+6}
a_{i+7}
CSA_5
ones
CSA_6
twos
CSA_7
eights
fours

应用

- popcount算法在某些方面有较广泛的应用，所以在某些处理器如Intel的64位处理器安腾2已经提供了这条指令，在gcc的较高版本和VC2008中也提供了这条函数。
- 简单的应用譬如位向量表示集合的求数目总和、开头提到的求汉明距离。
- 求后缀0：

  ntz(x) = pop (~x & (x -1))=32 – pop(x | -x)

- 生成符合二项分布的随机整数

应用

- 中等稀疏矩阵的储存

  对矩阵A，使用一个额外的位串bits，A[i]中凡是有定义的位置i，bits中相应的比特位i是1（位向量表示存储）。

  由于位串可能很长，将分成若干个32位字存储，用种群计数计算每个字的“1”含量就可以知道该字中含有几个元素，可以起到加速查找的作用。

启示

- 大的算法框架和小的算法优化都值得注意，大的算法框架更为重要因为其往往会带来算法效率数量级的进步，具体到每一步的算法优化也有机会在局部大幅度提高效率，但是往往会以牺牲程序可读性为代价。

参考文献

[1] Hamming Weight: http://en.wikipedia.org/wiki/Hamming_weight

[2] popcount: http://dafandong.spaces.live.com/blog/cns!2F6B5FFD15BDD1D6!637.entry

[3] HAKMEM ITEM 169: http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item169

[4] Popcount: http://wiki.cs.pdx.edu/forge/popcount.html

[5] HAKMEM 169 and other popcount implementations: http://www.dalkescientific.com/writings/diary/archive/2008/07/03/hakmem_and_other_popcounts.html

Thanks