**Parallel Counting**

***[Word Parallelism]***

**5.  MIT HAKMEM Count**

HAKMEM (Hacks Memo) is a legendary collection of neat mathematical and programming hacks contributed mostly by people at MIT and some elsewhere. This source is from the MIT AI LABS and this brilliant piece of code originally in assembly was probably conceived in the late 70's.

 int BitCount(unsigned int u)                           
 {  
         unsigned int uCount;  
      
         uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);  
         return ((uCount + (uCount >> 3)) & 030707070707) % 63;  
 }

Let’s take a look at the theory behind this idea.

Take a 32bit number n;  
n =  a31 \* 231 + a30 \* 230 +.....+ ak\* 2k  +....+ a1\* 2 + a0;

Here a0through a31 are the values of bits (0 or 1) in a 32 bit number. Since the problem at hand is to count the number of set bits in the number, simply summing up these coefficients would yield the solution

(a0+ a1+..+ a31).

How do we do this programmatically?

Take the original number n and store in the count variable.  
count=n;

Shift the orignal number 1 bit to the right and subtract from the original.  
count = n - (n >>1);

Now Shift the original number 2 bits to the right and subtract from count;  
count = n - (n>>1) - (n>>2);

Keep doing this until you reach the end.  
count = n - (n>>1) - (n>>2) - ... -( n>>31);

Let analyze and see what count holds now.  
n         = a31 \* 231 + a30 \* 230 +.....+ ak\* 2k  +....+ a1\* 2 + a0;

n >> 1 = a31 \* 230 + a30\* 229 +.....+ ak\* 2k-1  +....+ a1;  
n >> 2 = a31 \* 229 + a30 \* 228 +.....+ ak\* 2k-2 +....+ a2

;  
..  
n >> k = a31 \* 2(31-k) + a30 \* 2(30-k) +.....+ ak \* 2k;

..  
n>>31 = a31;

You can quickly see that: (Hint: 2k - 2k-1 = 2k-1;)  
count = n - (n>>1) - (n>>2) - ... -( n>>31) =a31+ a30+..+a0;  
which is what we are looking for;

int BitCount(unsigned int u)  
{  
       unsigned int uCount=u;  
       do  
       {  
             u=u>>1;  
             uCount -= u;    
       }  
       while(u);  
}

This certainly is an interesting way to solve this problem. But how do you make this brilliant? Run this in constant time with constant memory!

 int BitCount(unsigned int u)                           
 {  
         unsigned int uCount;  
      
         uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);  
         return ((uCount + (uCount >> 3)) & 030707070707) % 63;  
 }

For those of you who are still wondering what’s going? Basically use the same idea, but instead of looping over the entire number, sum up the number in blocks of 3 (octal) and count them in parallel.

After this statement uCount = n - ((n >> 1) & 033333333333) - ((n >> 2) & 011111111111);  
uCount has the sum of bits in each octal block spread out through itself.

So if you can a block of 3 bits

u = a2\*22 + a1\*2+ a0;  
u>>1 =a2\*2 + a1;  
u>>2 =a2;

u - (u>>1) - (u>>2) is a2+a1+a0 which is the sum of bits in each block of 3 bits.

The next step is to grab all these and sum them up:

((uCount + (uCount >> 3)) will re-arrange them in blocks of 6 bits and sum them up in such a way every other block of 3 has the sum of number of set bits in the original number plus the preceding block of 3. The only exception here is the first block of 3.  The idea is not to spill over the bits to adjacent blocks while summing them up. How is that made possible. Well, the maximum number of set bits in a block of 3 is 3, in a block of 6 is 6. and 3 bits can represent up to 7. This way you make sure you don’t spill the bits over. To mask out the junk while doing a uCount>>3. Do and AND with  030707070707. The only exception is the first block as I just mentioned.

What does ((uCount + (uCount >> 3)) & 030707070707) hold now?  
Its 2^0 \* (2^6 - 1) \* sum0 +  2^1 \* (2^6 - 1) \* sum1 + 2^2 \* (2^6 - 1) \* sum2  + 2^3 \* (2^6 - 1) \* sum3 + 2^4 \* (2^6 - 1) \* sum4 + 2^5 \* (2^3 - 1) \* sum5  
where sum0 is the sum of number of set bits in every block of 6 bits starting from the 'low' position.  
What we need is sum0 + sum1 + sum2 + sum3 + sum4 + sum5;  
2^6-1 is 63. Clearly a modulo with 63 will get you what you want.

Remember, that this works only with 32 bits numbers, For 64-bit numbers (well I will wait for comments or do this in the next post).

<http://blogs.msdn.com/b/jeuge/archive/2005/06/08/hakmem-bit-count.aspx>