Matrix Operations and Locality

Based on the paper

Organizing Matrices and Matrix Operations for Paged Memory Systems

By A.C.McKellar and E.G.Coffman, Jr.

Communications of the ACM March 1969

Virtual Memory

Most comventional computers since the mid 1970s use virtual memories.

In its simplest form, programs (including their data) are divided into *pages* of fixed length (usually 1024-4098 bytes).

The main memory is divided into *page frames*, each capable of holding a program page.

When the program begin execution, all its pages are in secondary memory (usually disk).

For each memory reference issued by the program (instruction or data) the computer checks whether the page containing the location being referenced is in main memory. If it is, the computer fetches it. Otherwise, the page is brought from secondary memory to the main memory (we assume that all variables and arrays are initialized to some value and therefore reside in disk when the program starts).

If there are unused page frames in main memory, one is chosen to place the incoming page.

Otherwise, an occupied page frame, say PF, has to be selected. If the page occupying PF was modified by the program, it would have to be written to disk before the incoming page is copied into the page frame. We will assume that the computer follows a *page replacement strategy* known as Least Recently Used (LRU) to select PF. In this strategy the system chooses the page frame containing the page that was least recetly accessed by the program as the one where the incoming page is to be placed.

We say there is a *page fault* whenever a page has to be brought from secondary storage.

Example 1

Consider the following loop:

```
      do i=1,4000
         a(i) = a(i)+1
      end do
```

Asume there is only one page frame available to hold data and that pages in this system contain 4096 bytes. In this case, when `a(1)` is referenced, the first page containing the array a will be brought from secondary memory. Then, no new page will be brough until the program tries to access `a(1025)`, the first element of the second page of a (we assume that each element of a is four bytes long). When this happen, the first page of a will be copied to secondary storage and the second page will be brought to the page frame in main memory.

Example 2

Consider the loop

```
      real a(1024,100)
      ...
      do i=1,1024
         do j=1,100
            a(i,j)=a(i,j)+1
         end do
      end do
```

Assume there are two page frames available to store data and that pages are 4096 byles long.

Assume also that a is sorted by columns (as is the case in Fortran. In C, arrays are stored by rows). That means that each page will contain *exactly* one column.

This program will cause one page fault for each iteration of the loop `j`. To understand why, let us analyze the sequence of memory references generated by the loop in terms of page being accessed. This sequence (known as *memory trace*) has the form:

a_1 a_1 a_2 a_2 a_3 a_3 a_4 a_4 ... a_{100} a_{100} a_1 a_1 a_2 a_2 a_3 a_3 ... a_{100} a_{100} a_1 a_1 a_2 a_2 ...

where a_i represents the page containg the ***i***th column. Notice that each iteration accesses the same element (and therefore the same column) twice.

Notice that a_j accessed by pairs. If the computer follows the LRU replacement policy, the first access to each pair a_j will cause a page fault. Thus, the first access of all (a_1) will cause a page fault because there will be no page of a in main memory. The third reference (a_2) will also cause a page fault because only t column a_1 will be in main memory. After this, the first access in each pair a_j will cause a page fault because the two pages residing in memory will be different from a_j (a_{j-1} and a_{j-2} if $j>2$ -- a_{100} and a_1 if $j=2$ -- a_{100} and a_{99} if $j=1$).

In other words, the previous loop will cause 102,400 page faults.

On the other hand, if the loops were interchanged, the previous loop will take the form:

```
real a(1024,100)
...
do j=1,100
   do i=1,1024
      a(i,j)=a(i,j)+1
   end do
end do
```

The trace for this loop is:

a_1 a_1 ... a_1 a_1 a_2 a_2 ... a_2 a_2 a_3 a_3 ...

It is easy to see that this second form of the loop causes only 100 page faults.

Since each page fault takes a long time, the second version would be significantly faster than the first one.

Notice that the number of page faults for the previous two loops will be valid whenver the number of page frames is less than 100. If the number of page frames is 100 or more, the number of page frames will be 100 for both loops.

Transposing a matrix.

Now consider the loop:

```
      do i=1,1024
         do j=i+1,1024
            t=a(j,i)
            a(j,i)=a(i,j)
            a(i,j) = t
         end do
      end do
```

Assume again that there are two page frames available for data. For simplicity of analysis assume also that variable `t` is kept in a register.

Assume again that each page frame holds a whole column of the array `a`. The address trace has the following form:

a_1 a_2 a_1 a_2 - a_1 a_3 a_1 a_3 - ...a_1 a_{1023} a_1 a_{1023} -
a_1 a_{1024} a_1 a_{1024} + a_2 a_3 a_2 a_3 - a_2 a_4 a_2 a_4 - ...
a_2 a_{1024} a_2 a_{1024} + a_3 a_4 a_3 a_4 - ...
a_{1021} a_{1024} a_{1021} a_{1024} + a_{1022} a_{1023} a_{1022} a_{1023} -
a_{1022} a_{1024} a_{1022} a_{1024} + a_{1023} a_{1024} a_{1023} a_{1024}

Here, the - indicates the end on a `j` iteration, and the + the end of an `i` iteration.

Given an `i` iteration, the first `j` iteration causes two page faults (except for `i=1023` which on ly causes one). This is true for `i=1` because there is nothing in main mamory when the loop starts. When iteration `i=2` starts, memory will contain pages a_1 and a_{1024} and the first iteration of loop `j` for `i=2` will access a_2 and a_3.

Subsequent iteration of the `j` loop within a give `i` iteration will cause one page fault because each new iteration of the `j` loop will access one new column of the array a.

Therefore, the total number of page faults will be:

1023 + 1024*1023/2.

An alternative way of executing the loop would be to reverse the order of the inner loop:

```
      do i=1,1024
         do j=1024,i+1,-1
            t=a(j,i)
            a(j,i)=a(i,j)
            a(i,j) = t
         end do
      end do
```

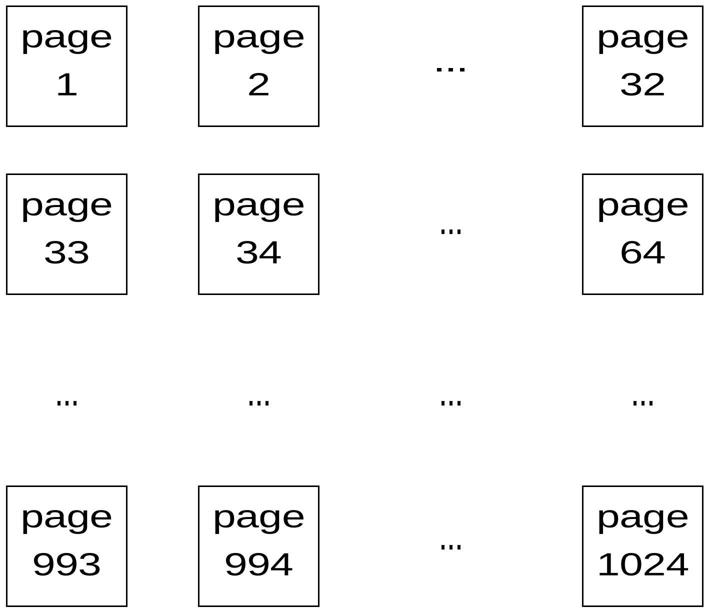
In this loop all iterations of the `j` loop will always cause one page fault. Therefore for this second loop the total number of page faults would be 1+1023*1024/2.

To see why, consider the address trace.

a_1 a_{1024} a_1 a_{1024} - a_1 a_{1023} a_1 a_{1023} - ...a_1 a_3 a_1 a_3 - a_1 a_2 a_1 a_2 + a_2 a_{1024} a_2 a_{1024} - a_2 a_{1023} a_2 a_{1023} - ... a_2 a_3 a_2 a_3 + a_3 a_{1024} a_3 a_{1024} - ... a_{1021} a_{1022} a_{1021} a_{1022} + a_{1022} a_{1024} a_{1022} a_{1024} - a_{1022} a_{1023} a_{1022} a_{1023} + a_{1023} a_{1024} a_{1023} a_{1024}

It is easy to see that for a given i, all j iterations after the first one cause only one page fault. Furthermore, except for a given i>1, the first j iteration will reuse one of the columns accessed last in the i-1 iteration.

Now assume that array a is stored so that each 32 by 32 submatrix is stored in a page as follows (notice that $32 = \sqrt{1024}$):



Then, the following program will access one page at a time and will cause only 1024 page faults:

```
do II=1,1024,32
   do JJ=II,1024,32
      do i=II, II+32
         do j=i+1,JJ+32
            t=a(i,j)
            a(i,j)+a(j,i)
            a(j,i)=t
         end do
      end do
   end do
end do
```

Matrix multiplication

In this discussion, we will assume that there are three page frames available for data.

Assume first, row storage. It is easy to see that the straightforward algorithm:

```
do i=1,1024,
   do j=1,1024
      do k=1,1024
         c(i,j)=a(i,k)*b(k,j)+c(i,j)
      end do
   end do
end do
```

will cause 1024^2*(1024+2) page faults.

To see why consider the address trace of the inner loop for a particular `i` and `j`:

a_1 b_j c_j c_j - a_2 b_j c_j c_j - ... - a_{1024} b_j c_j c_j

There will be 1024 faults due to a, but only 2 due to b and c.

The result follows because there are 1024^2 `(i,j)` iterations.

The middle product method behaves better. Recall that the middle product method is:

```
      do j=1,1024,
         do k=1,1024
            do i=1,1024
               c(i,j)=a(i,k)*b(k,j)+c(i,j)
            end do
         end do
      end do
```

Now, the inner loop memory trace for a given `j` and `k`, is as follows

$a_k\ b_j\ c_j\ c_j\ -\ a_k\ b_j\ c_j\ c_j\ -\ \ldots\ -\ a_k\ b_j\ c_j\ c_j$

and, therefore, only 3 page faults take place in this inner loop.

For different values of `k`, a_k will change, but b_j and c_j will remain constant. Therefore, there will be 1024 +2 page faults for the `k` loop and 1024*(1024+2) for the whole loop nest.

Finally, asume now that the matrix is stored by submatrices as depicted above.

Matrix multiplication can be done as follows:

```
do II=1,1024,32
   do JJ=1,1024,32
      do KK=1,1024,32
         do i=II,II+31
            do j=JJ,JJ+31
               do k=KK,KK+31
                  c(i,j)=a(i,k)*b(k,j)+c(i,j)
               end do
            end do
         end do
      end do
   end do
end do
```

Notice that the three inner loops (inside the square) represent the product of two submatrices. Therefore, this version is just the well-known submatix form of the matrix multiplication algorithm. Assuming again that there are three page frames in main memory, the loop will cause just three page faults for the innermost three loops. The KK loop will cause one page fault for c and 32 page faults for each of a and b. There fore, the total number of page faults will be:

$32*32*(1+2*32)=1024+1024^{1.5}*2$

Cache memories

An important component of modern computers are cache memories. They are another level of the memory hierarchy which is divided into cache blocks (which roughly corresponds to the page frames in main memory although much smaller) and are backed up to main memory.

For the purposes of this dicussion, we will assume that the block replacement strategy is LRU. This is not true in most real machines, but is a reasonable approximation in some cases.

So, to analyze the behavior of the algorithms dicussed above, the only change will be the size of the blocks.

Consider the loop of Example 2. Assume that the cache contains 64 blocks for data (in fact any number of blocks b, with $1<b<100$ would do). If we assume that each cache block is 32 bytes long (8 array elements long), the total number of *cache misses* for the first version of the loop will be 102,400 which is the same as the number of page faults in Example 2.

If the loop headers are interchanged, the number will be reduced to 102,400/8 which is the total number of cache blocks in array a.

Now consider matrix multiplication. Assume that the cahce can hold three columns. The simple approach will produce the following cache misses for the k loop: 1 for c, 1024/8 for b, 1024 for a. Total cache misses: $1024^2(1+1024+1024/8)$.

For the middle product method, thre will be the following cache misses for loop k: 1024/8 for c, 1024/8 for a, and 1024 for b. Total: $1024(1024/8+1024/8+1024) = 2*1024^2/8+1024^3$

Finally, for the submatrix product, for each iteration of the KK loop, there will be 1024/8 cache misses due to c and 32*1024/8 for each of a and b. The resulting number of cache misses will be: $32*32*(1024/8+2*32*1024/8)=1024^2/8+2*1024^{2.5}/8$.