

Programming models

A classification

- Sequential programming
- Parallel programming
 - > Vector
 - > Message-passing
 - > Shared-memory

Sequential programming

- Understanding performance of sequential codes is important because parallel programs are typically composed of sequential modules.
- Most important consideration is locality.
- Locality: “the concept that a program favors a subset of its segments during extended intervals (phases) is called locality.”
 - > There is no common figure of merit for locality.
 - > One related measurement is space-time product which when reduced maximizes throughput. It is not only relevant for multiprogrammed systems, but is also important for parallel programming particularly for *embarrassingly parallel* programs.
 - > One useful metric could be a cache size/# of cache misses graph. Closer to the origin would be better.

Tiling

A time-honored technique to “improve locality” is tiling.

We will illustrate its benefits to reduce cache misses using two simple examples.



Transposing a matrix.

Consider the loop:

```
for ( i=1, i <= n, i++) {  
    for ( j=i, j <= n, j++) {  
        t=a[j,i]  
        a[j,i]=a[i,j]  
        a[i,j] = t  
    }  
}
```

If the cache had fewer than n cache lines, there will be a cache miss

- > on every iteration of the inner loop for each new row of a accessed (n) and
- > one cache miss for each line across rows (n/L). Why ?

Therefore, number of cache misses would be $n^2(1+1/L)$

If, on the other hand, we tile the matrix transpose as follows:

```
for ( i=1, i <= n, i+=s) { /* assume n multile of s */
  transpose(a[i:i+s-1,i:i+s-1])
  for ( j=i+1, j <= n, j+=s) {
    transpose(a[i:i+s-1,j:j+s-1], a[j:j+s-1,i:i+s-1])
  }
}
```

where the function `transpose` transposes its parameter when there is only one parameter, otherwise transposes the two submatrices and exchanges them.

If the two submatrices fit in the cache, the number of cache misses will be only n^2/L .

Matrix multiplication.

Consider the loop

```
for ( i=1, i <= n, i++) {
    for ( j=1, j <= n, j++) {
        for ( k=1, k <= n, k++) {
            c[i,j]=a[i,k]*b[k,j]+c[i,j]
        }
    }
}
```

If the cache contains fewer than n/L lines, there will be cache miss for every execution of the k loop will bring n misses due to b and n/L due to a . Each execution of the j loop will bring n/L misses due to c .

Total: $n^3+n^3/L + n^2/L$



The middle product version of matrix matrix multiplication behaves better:

```
for ( i=1, i <= n, i++) {
    for ( k=1, k <= n, k++) {
        for ( j=1, j <= n, j++) {
            c[i,j]=a[i,k]*b[k,j]+c[i,j]
        }
    }
}
```

Under the same assumptions, we have that each iteration of the j loop brings n/L cache misses due to c and the same number due to b . Each iteration of the k loop brings additionally n/L misses due to a .

Total: $2n^3/L + n^2/L$



If we now tile matrix matrix multiplication, we get

```
for ( i=1, i <= n, i+=s) { /* assume n multile of s */
  for ( j=1, j <= n, j+=s) {
    for ( k=1, k <= n, k+=s) {
      matmul(c[i:i+s-1,j:j+s-1],
            a[i:i+s-1,k:k+s-1],b[k:k+s-1,j:j+s-1])
    }
  }
}
```

Assuming that the three tiles fit in cache, we will have $2 s^2 / L$ misses for `matmul` due to `a` and `b` and additionally s^2 / L misses due to `c` for each iteration of the `j` loop.

Total: $(n/s)^3 * 2 s^2 / L + (n/s)^2 * s^2 / L = 2 n^3 / (s L) + n^2 / L$