CS420 – Lecture 18

Marc Snir

Fall 2018

Covariance Matrix

v is $m \times n$ matrix, c is $n \times n$ matrix.

$$c_{jj'} = \sum_{i=1}^{m} v_{ij} v_{ij'}$$

(We ignore normalizing division by m.) Denote by v^T the transpose of v: $v_{ij}^T = v_{ji}$. Then

$$c_{jj'} = \sum_{i=1}^{m} v_{ji}^T v_{ij'}$$

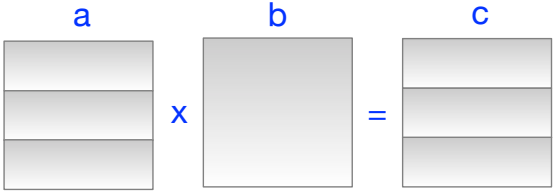
So $c = v^T \cdot v$. It is a matrix product of a special kind.
Note: $v^T \cdot v = v \cdot v^T$ is symmetric.

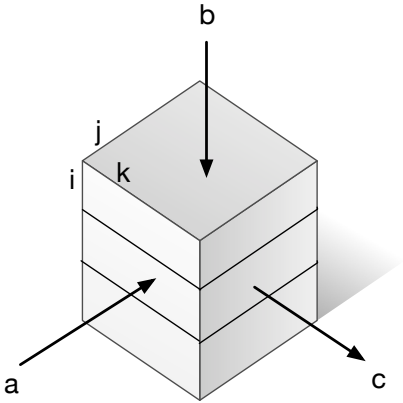
Tiled matrix product code

```
for (i=0;i<n;i++)
 for (j=0; j<n;j++)
  for (k=0;k<n;k++)
cube    c[i][j] += a[i][k]*b[k][j];
```

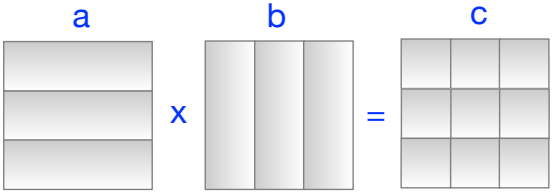
- 1D: Tile `i`
- 2D: Tile `i,j`
- 3D: Tile `i,j,k`

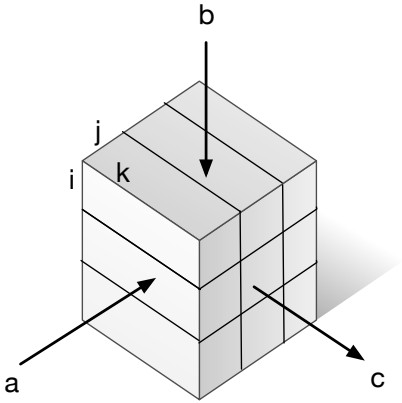
1D tiling



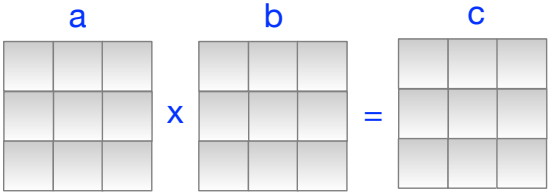


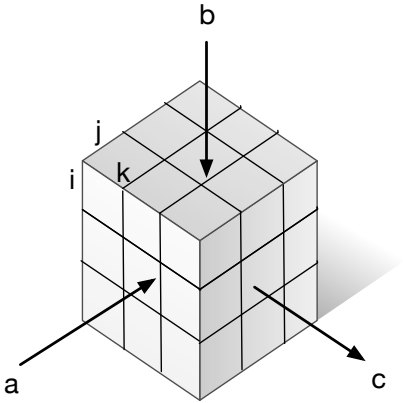
2D tiling



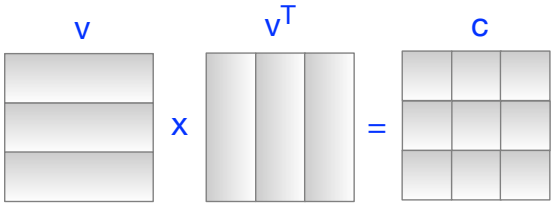


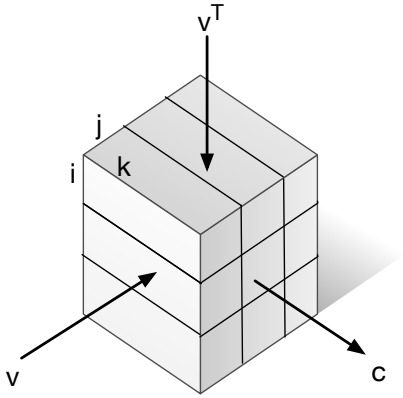
3D tiling



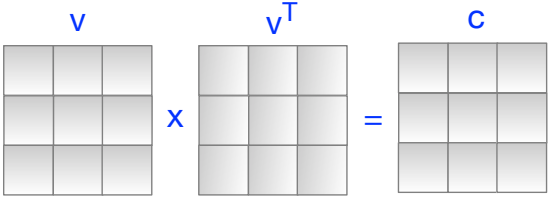


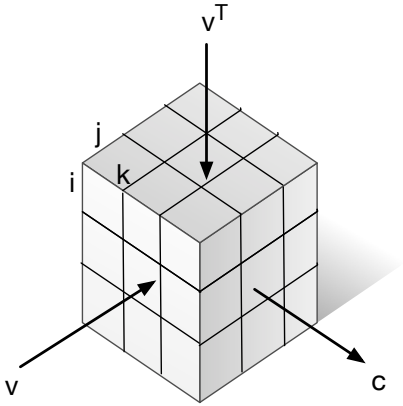
Covariance Matrix: 2D tiling





Covariance Matrix: 3D tiling





Covariance matrix code

```
for (i=0;i<n;i++)
 for (j=0; j<n;j++)
  for (k=0;k<m;k++)
   c[i][j] += v[i][k]*v[j][k];
```

Leverage symmetry

```
for (i=0;i<n;i++)
 for (j=0; j<=i;j++)
  for (k=0;k<m;k++)
   c[i][j] += v[i][k]*v[j][k];
```

Even better: Store only triangular matrix

2D tiling

```
for (I=0; I<n; I+=T)  {
 Inext = I+T < n? I+T:n;
 for (J=0; J<I; J+=T) {
  Jnext = J+T < Inext? J+T:Jnext;
  for(i=I;i<Inext;i++)
   for(j=J;j<Jnext;j++)
    for(k=0;k<m;k++)
     c[i][j]+=v[i][k]*v[j][k]
  }
}
```

2D tiling

```
for (I=0; I<n; I+=T)  {
 Inext = I+T < n? I+T:n;
  for (J=0; J<I; J+=T) {
   Jnext = J+T < Inext? J+T:Jnext;
    for (K=0;K<m; K+=t) {
     Knext= K+T<m? K+t:m;
   for(i=I;i<Inext;i++)
    for(j=J;j<Jnext;j++)
     for(k=K;k<m\Knext;k++)
      c[i][j]+=v[i][k]*v[j][k]
   }
  }
   }
```

2D tiling with GPU

```
for (I=0; I<n; I+=T) {
 Inext = I+T < n? I+T:n;  Ilen=Inext-I;
 #pragma omp target entry data map(to:v[I:Ilen][m]), I, Inext, m, n)
  for (J=0; J<I; J+=T) {
   Jnext = J+T < Inext? J+T:Jnext; Jlen=Jnext-J;
   flatten(c,I,J,Inext-I,Jnext-J,*ctemp);
   #pragma omp target map(to:v[J:Jlen][m], J, Jnext) \
      map(tofrom: ctemp[:Ilen*Jlen])
  for(int i=I;i<Inext;i++)
   for(int j=J;j<Jnext;j++)
    for(int k=0;k<m;k++)
     *(ctemp+Jlen*i+j)=v[i][k]*v[j][k];
  }
 #pragma omp target exit data map(release:v[I:Ilen])
 }
```

3D tiling code is similar – just more complicated... and possibly improved by loop exchange

Cannon's Algorithm

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

- Assume 3×3 grid of processes; each processe contains a submatrix of a and a submatrix of b
- Shift cyclically right a submatrices and cyclically down b submatrices

$$\begin{pmatrix} a_{11}b_{11} & a_{12}b_{22} & a_{13}b_{33} \\ a_{22}b_{21} & a_{23}b_{32} & a_{21}b_{13} \\ a_{33}b_{31} & a_{31}b_{12} & a_{32}b_{23} \end{pmatrix} \quad \begin{pmatrix} a_{13}b_{31} & a_{11}b_{12} & a_{12}b_{23} \\ a_{21}b_{11} & a_{22}b_{22} & a_{23}b_{33} \\ a_{32}b_{21} & a_{33}b_{32} & a_{31}b_{13} \end{pmatrix} \quad \begin{pmatrix} a_{12}b_{21} & a_{13}b_{32} & a_{11}b_{13} \\ a_{23}b_{31} & a_{21}b_{12} & a_{22}b_{23} \\ a_{31}b_{11} & a_{32}b_{22} & a_{33}b_{33} \end{pmatrix}$$

Compute $v^T v$ using same algorithm

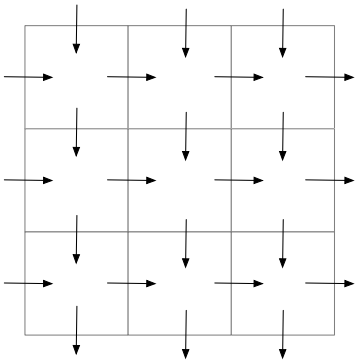
$$\begin{pmatrix} v_{11}v_{11} & v_{21}v_{22} & v_{31}v_{33} \\ v_{22}v_{21} & v_{32}v_{32} & v_{12}v_{13} \\ v_{33}v_{31} & v_{13}v_{12} & v_{23}v_{23} \end{pmatrix} \quad \begin{pmatrix} v_{31}v_{31} & v_{11}v_{12} & v_{21}v_{23} \\ v_{12}v_{11} & v_{22}v_{22} & v_{32}v_{33} \\ v_{23}v_{21} & v_{33}v_{32} & v_{13}v_{13} \end{pmatrix} \quad \begin{pmatrix} v_{21}v_{21} & v_{31}v_{32} & v_{11}v_{13} \\ v_{32}v_{31} & v_{12}v_{12} & v_{22}v_{23} \\ v_{13}v_{11} & v_{23}v_{22} & v_{33}v_{33} \end{pmatrix}$$

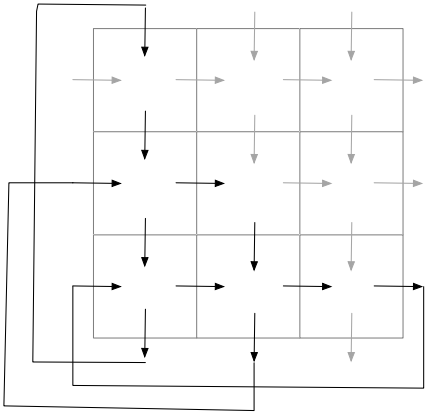
- Diagonal opposites make the same computations (symmetry)
- Shifts on row i are same as shifts on column i

Take advantage of symmetry

$$\begin{pmatrix} v_{11}v_{11} & & \\ v_{22}v_{21} & v_{32}v_{32} & \\ v_{33}v_{31} & v_{13}v_{12} & v_{23}v_{23} \end{pmatrix} \quad \begin{pmatrix} v_{31}v_{31} & & \\ v_{12}v_{11} & v_{22}v_{22} & \\ v_{23}v_{21} & v_{33}v_{32} & v_{13}v_{13} \end{pmatrix} \quad \begin{pmatrix} v_{21}v_{21} & & \\ v_{32}v_{31} & v_{12}v_{12} & \\ v_{13}v_{11} & v_{23}v_{22} & v_{33}v_{33} \end{pmatrix}$$

Use symmetry to "fold" shifts





- shift right until reach diagonal then down
- shift from bottom of column i to start of row i

Analysis

- Assume $P = p(p+1)/2$ processes.
- Each process stores two $\frac{m}{p} \times \frac{n}{p}$ submatrices of v and a $\frac{n}{p} \times \frac{n}{p}$ submatrix of c.
- Need (in a naive implementation) space to buffer two extra submatrices of v, in order to send and receive simultaneously.
- Computation has p phases of compute/communicate/communicate
- Each computation takes $\alpha \frac{m}{p} \cdot (\frac{n}{p})^2 = \frac{\alpha m n^2}{p^3}$ times
- Each communication takes $\beta \frac{m}{p} \cdot \frac{n}{p} = \frac{\beta m n}{p^2}$ time
- Total time is

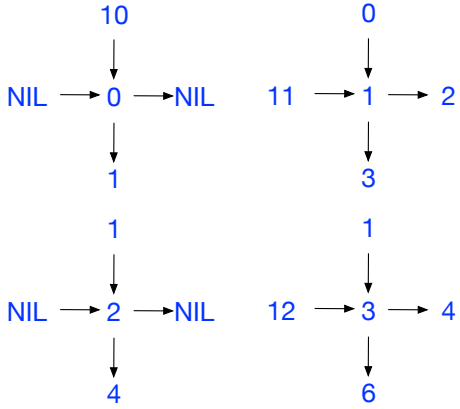
$$\begin{aligned} T &= p(\frac{\alpha m n^2}{p^3} + \frac{\beta m n}{p^2}) \\ &= \frac{\alpha m n^2}{p^2} + \frac{\beta m n}{p} \\ &\approx \frac{\alpha m n^2}{2P} + \frac{\beta m n}{\sqrt{2P}} \end{aligned}$$

- Sequential time (assuming everything fits in memory) is $T_1 \approx \alpha m n^2/2$.
- Parallel time is $T_p \approx \frac{\alpha m n^2}{2P} + \frac{\beta m n}{\sqrt{2P}}$
- For large n first term dominates: $T_p = \approx \frac{1}{P}\alpha m n^2/2$
- algorithm is efficient for large n.

Process numbering

	0	1	2	3	4
0	0				
1	1	2			
2	3	4	5		
3	6	7	8	9	
4	10	11	12	13	14

Neighbor connections



Process numbering

	0	1	2	3	4
0	0				
1	1	2			
2	3	4	5		
3	6	7	8	9	
4	10	11	12	13	14

- First process in row i has rank $r = i(i+1)/2$. We get $i = \sqrt{2r + 0.25} - 0.5$
- Therefore process with rank r is in row $\lfloor \sqrt{2r + 0.25} - 0.5 \rfloor$

MPI (pseudo)code

Assume p divides m and n

```
MPI_Comm_size(MPI_COM_WORLD, &size);
MPI_Comm_rank(MPI_COM_WORLD, &rank);
p = sqrt(2.0*size);
if(p*(p+1)!=2*size) exit(1);
/* Compute Cartesian coordinates */
i =  sqrt(2.0*rank-0.25)-0.5;
j = rank - i*(i+1)/2;
/* Compute neighbors */
neighbors(rank,i,j,&left,&right,&up,&down);
```

Compute pipeline

```
for(iter=0;iter<p;iter++) {
MPI_Irecv(v1[1-turn],mn/(p*p), MPI_FLOAT,up,0,&req[0])
MPI_Irecv(v2[1-turn],mn/(p*p), MPI_FLOAT,left,0,&req[1]);
MPI_Isend(v1[turn],mn/(p*p), MPI_FLOAT,right,0,&req[2])
MPI_Isend(v2[turn],mn/(p*p), MPI_FLOAT,down,0,&req[3])
if(j=i) { /* corner process */
 for(j1=0; j1<n/p;j1++)
  for(j2=0;j2<=j1;j2++)
   for(i=0;i<m/p;i++)
    c[j1][j2] += v1[i][j1]*v1[i][j2];
else {  /* regular process */
 for(j1=0; j1<n/p;j1++)
  for(j2=0;j2<=j1;j2++) {
   c[j1][j2] = 0;
   for(i=0;i<m/p;i++)
  c[j1][j2] += v1[i][j1]*v2[i][j2];
   }
```

```
MPI_Waitall(4,req,MPI_STATUSES_IGNORE);
turn=1-turn;
}
```

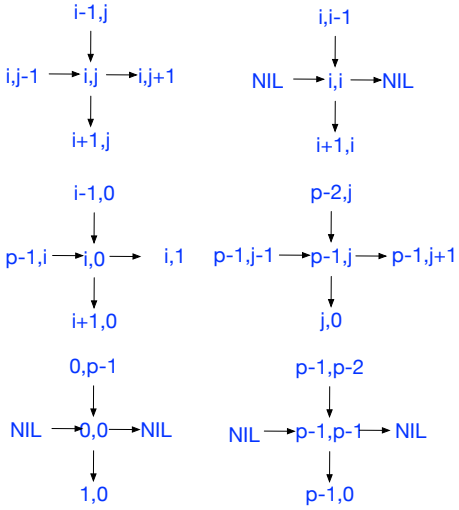
Improvements:

- Can tile computation loop
- Can further pipeline communications, sending submatrices in chunks, to reduce storage consumption

Process numbering

	0	1	2	3	4
0	0				
1	1	2			
2	3	4	5		
3	6	7	8	9	
4	10	11	12	13	14

Neighbor connections



Compute neighbors

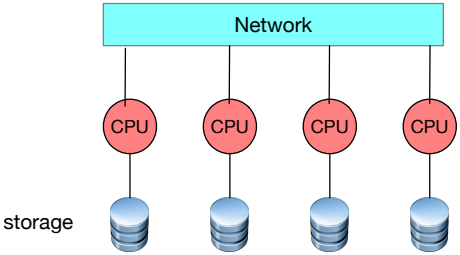
```
void neighbors(int rank, int i, int j,int *left,
     int *right,int *up,int *down) {
 left= j==0? (i==0?MPI_PROC_NULL:p*(p+1)/2+i)):rank-1;
 right= j<i? rank+1:(i==0?p*(p+1)/2:MPI_PROC_NULL);
 up = j<i? rank-i : (i=0?p*(p+1)/2:rank-1)
 down = i<p-1? rank+i: j*(j+1)/2
}
```

Parallel I/O

I/O systems for clusters

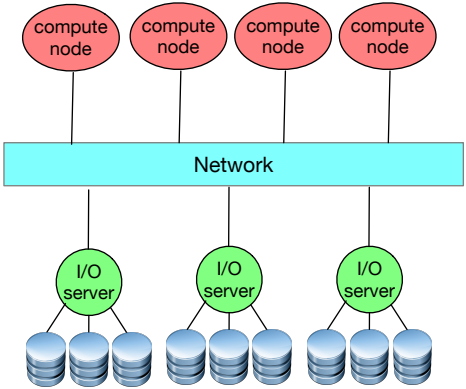
Local I/O – each node has its own file system

- pro:
  - Simple, cheap
- cons:
  - Most of the time, data will be remote – local CPU needs to provide remote access
  - Package (power, cooling) is less efficient



Global I/O – one file system.

- A large file may be distributed (stripped) across many disks and many I/O servers
- I/O servers and compute nodes are specialized for their respective tasks
- Package is different



Application I/O

- One process (process 0) performs I/O for the entire application; data is then distributed using MPI
  - I/O becomes a bottleneck, for large process counts
- Each process reads/writes separate files
  - Need to manage large number of files
  - Not convenient when files are written then read by different number of processes
- All processes read or write different parts of one large file
  - Performance will be very dependent on how the file is distributed and how it is accessed
- I/O is performed by a specialized library
  - Replaces "sequence of bytes" by higher-level objects, such as "array of floats".
  - Library can pick a good distribution
  - Collective accesses provide opportunity for aggregating disk accesses