# Shared Memory Programming with OpenMP

## Case Study: Matrix Products

# Recap

Parallel programs performs well when atomic and synchronization constructs are reduced.

- total removal not possible in all cases.

Locality of reference importance for matrix operation

- Spatial and Temporal Locality

Access latency

- In the increasing order: L1 cache, L2 cache, L3 cache, Main Memory, Hardisk (secondary storage)

Sequential Consistency

- Output of parallel program should be same as one sequential execution
- Inter iteration Dependence (RAW/WAR/WAW) should not be there.

Matrices can be allocated statically or dynamically (malloc() function)

# Recap -

Analyzing Performance of  Parallel Program

- Unix command - time shows (real,user, sys)
- Measure performance by inserting function calls
  - gettimeofday, clock etc.
- gprof
  - contains detailed statics
  - compile with proper arguments to gcc
  - contains details on  how much time each function executed
  - contains call graph information
  - output of gprof can be  customized with proper arguments.
- Tools for debugging
  - gdb, valgrind, objdump

# Lecture Plan

A Simple Matrix Multiplication Code.
A simple Example.
Look at Hardware.
- Memory hierarchy, Access Latency
- Processor Unit- Registers, ALU

Modify the program so that running time can be reduced.
- Unrolling loop
- Tiling loop
- Loop interchange

# Python code for matrix multiplication

```
1   import sys,random
2   from time import *
3   n = 4096
4   A = [ [ random.random()
5           for row in range(n)]
6           for col in range(n)]
7   B = [ [ random.random()
8           for row in range(n)]
9           for col in range(n)]
10
11  C = [ [ 0 for row in range(n)]
12        for col in range(n)]
13
14  start = time()
15  for i in range(n):
16      for j in range(n):
17          for k in range(n):
18              C[i][j] += A[i][k]*B[k][j]
19  end = time()
20  print ('%0.6f' % (end - start))
21
```

# C code for matrix multiplication

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/times.h>
#define n 4096
double A[n][n], B[n][n],C[n][n];
#include <sys/time.h>
double rtclock()
{
    struct timezone Tzp;
    struct timeval Tp;
    int stat;
    stat = gettimeofday (&Tp, &Tzp);
    if (stat != 0) printf("Error return from gettimeofday: %d",stat);
    return(Tp.tv_sec + Tp.tv_usec*1.0e-6);
}

int main(int argc,char *argv[]){
    srand(atoi(argv[0]));
    for (int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            A[i][j]= (double)rand()/(double) RAND_MAX;
            B[i][j]= (double)rand()/(double) RAND_MAX;
            C[i][j]=0;
        }
    }
    double t1=rtclock();
    for (int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            for(int k=0;k<n;k++){
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
    double t2=rtclock();
    printf("%0.6f", (t2-t1)*1000);
}
```

Time Taken by Python Code?
214 minutes on my machine
Time Taken by C code?
15 minutes on my machine, using clang compiler.

Python code is interpreted

# A Simple Example (3x2 and 2x3 matrices)

Matrix A (3x2)

| | |
|---|---|
| 2 | 4 |
| 6 | 8 |
| 10 | 12 |

| | | |
|---|---|---|
| 1 | 3 | 5 |
| 7 | 9 | 11 |

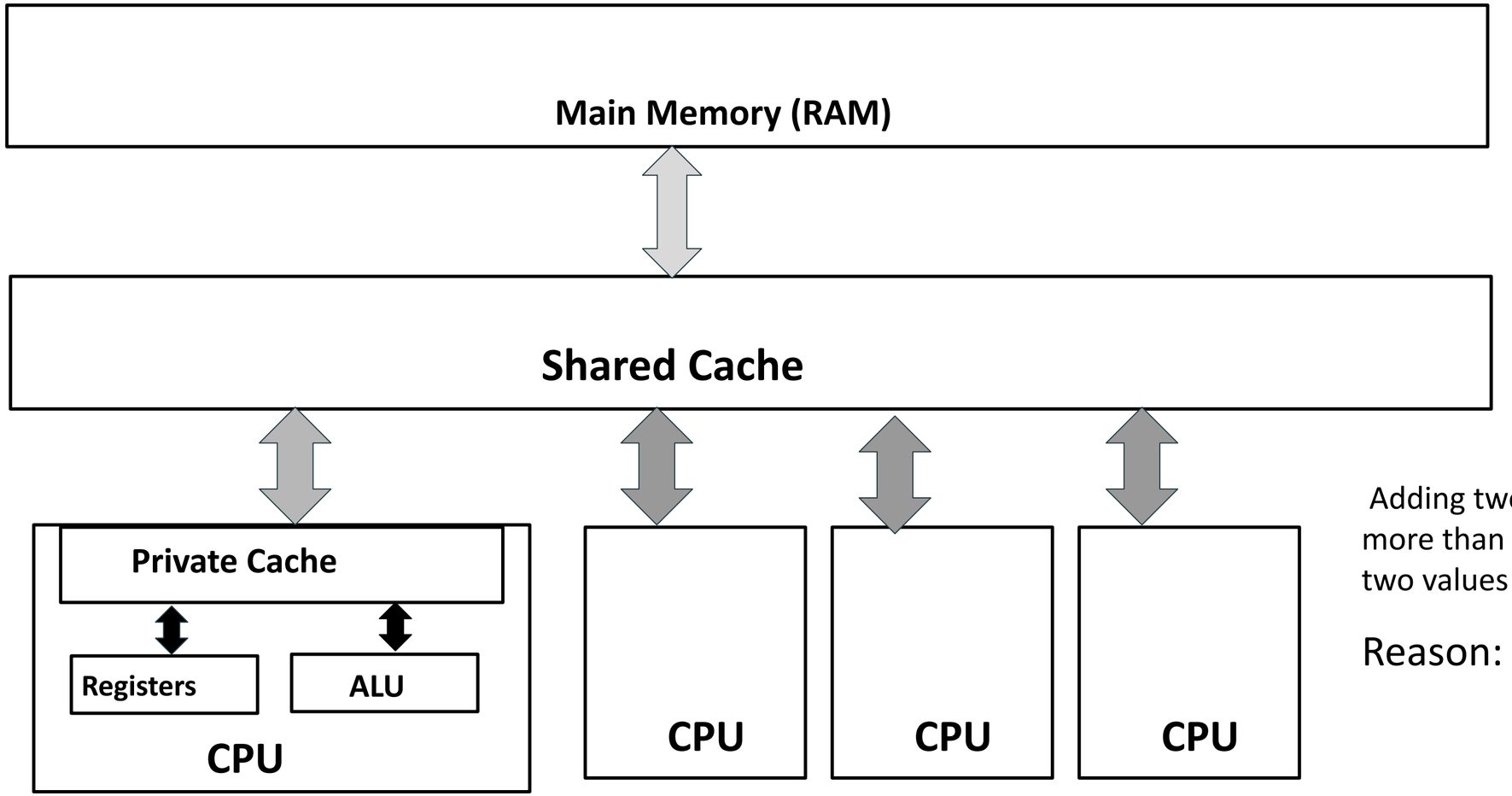
Matrix B (2x3)

| | | |
|---|---|---|
| 30 | 42 | 54 |
| 62 | 90 | 118 |
| 94 | 138 | 182 |

Matrix C (3x3)

**How to make the program efficient?**

# Multi-core CPU Architecture




Adding two register values could be more than 100x faster than adding two values in main memory.

Reason: Access Latency

# Matrix Product

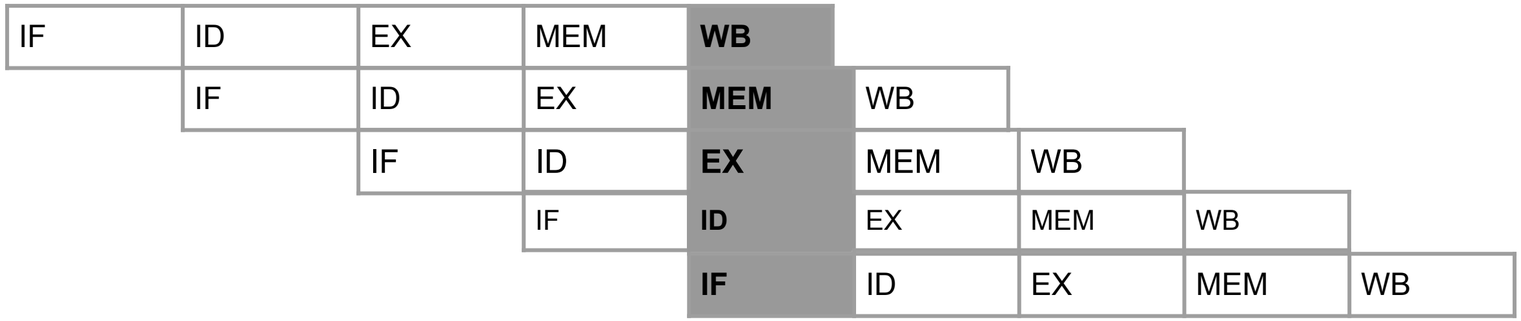
```
int A[M][T], B[T][N], C [M]][N];
……//other declaration statements
main(){
….//code fragment to read matrix A
…..
```

```
   …..//print matrix  C
}//end main
```

# Instruction Level Parallelism (ILP)

An instruction execution consist of five stages.

- Instruction Fetch (**IF**)
- Instruction Decode (**ID**)
- Execute (**EX**)
- Memory Access (**MEM**)
- Write Back (**WB**)



# ILP: example with one dimensional arrays

**f o r** ( **int** i = 1; i <= size; i++) C[ i ] = k **+** A[i]***B[i]**
A,B,C are one dimensional array

```
//R4= k, R10=size
//Loop Code
L:   LD R5 , O ( Rl + + )   // R1= &A
     LD R6 , O(R2++)
     MUL R7 , R5 , R6
     nop
     ADD R8 , R7 , R4
     nop
     ST  0(R3++) , R8  //
     BL R10 , L //BL decrement R10, if R10 > 0 goto L.
```

// **Seven Clock Cycles for one iteration**

Assume a Machine that Can issue in a single clock
*one load
*one store
*one arithmetic operation, and
*one branch operation BL

*Results of arithmetic operation available after two clock cycle.
**Can we reduce clock cycles per iteration??**

# Loop Unrolling

The loop code with one iteration was taking
- seven clock cycles for one iteration of the loop

loop unrolling
- for (int i=1;i<=size;i=i+4) { itr(i);itr(i+1); itr(i+2);itr(i+3); }
- Assuming value of **size** is divisible by four.
- itr (i) → C[ i ] = k **+** A[i]*B[i]
- itr (i+1) → C[ i+1 ] = k **+** A[i+1]*B[i+1]
- itr(i+2) → C[ i+2 ] = k **+** A[i+2]*B[i+2]
- itr(i+3) → C[ i+3 ] = k **+** A[i+3]*B[i+3]

# Unrolled loop execution (Software Pipelining)

| itr= 1 | itr= i+1 | itr= i+2 | itr= i+3 | | clock cycle |
|---|---|---|---|---|---|
| LD | | | | | 1 |
| LD | | | | | 2 |
| | LD | | | | 3 |
| MUL | LD | | | | 4 |
| | MUL | LD | | | 5 |
| ADD | | LD | | | 6 |
| | ADD | | LD | | 7 |
| ST | | MUL | LD | | 8 |
| | ST | | MUL | | 9 |
| | | ADD | | | 10 |
| | | | ADD | | 11 |
| | | ST | | | 12 |
| | | | ST | BL L | 13 |

four iterations
13 clock cycles
clock cycles per iteration = 3.25 (13/4)
=7 for normal loop

# Matrix Multiplication code

```
for( int i=0;i< N;i++) { //Loop 1
      for(int j=0;j<N;j++) { //Loop 2

    for(int k=0;k<N;k++){ //Loop 3

        C[i][j]=C[i][j]+A[i][k]*B[k][j]

      }

   }

}
```

Assumption

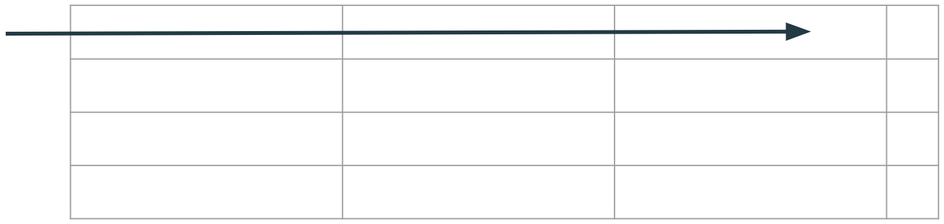
Square matrices

Matrices size NxN

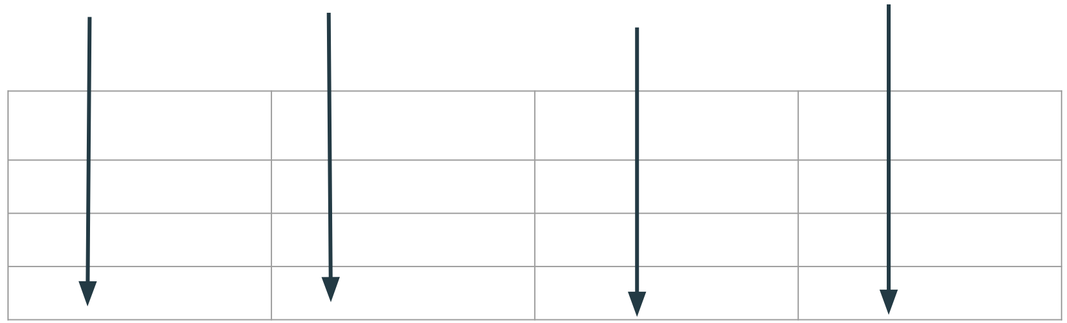
# Matrix Multiplication

The algorithm takes as two NxN matrices: A, and B
Produces a new NxN matrix: C
Number of operations: $N^3$
Assumption: matrix stored in row major order.

Data fetched from main memory to cache as a cache line of **L bytes**.
for ( k= 0 to (N-1)) C[i,j] **+=** A[i,k]*B[k,j]



A [i,k]



B [k,J]

# Matrix Multiplication code

```
for( int i=0;i< N;i++) { //Loop 1
for(int j=0;j<N;j++) { //Loop 2
        C[i][j]=0;
for(int k=0;k<N;k++){ //Loop 3
        C[i][j]=C[i][j]+A[i][k]*B[k][j]
    }
  }
}
```

Low/No Spatial locality
As Access is Column wise.

# Loop Interchange

```
for( int i=0;i< N;i++) { //Loop 1
for(int j=0;j<N;j++) { //Loop 2
                    C[i][j]=0;
for(int k=0;k<N;k++){ //Loop 3
        C[i][j]=C[i][j]+A[i][k]*B[k][j]
    }
  }
}
```

```
for(int j=0;j<N;j++) { //Loop 1
for( int i=0;i< N;i++) { //Loop 2
                    C[i][j]=0;
for(int k=0;k<N;k++){ //Loop 3
        C[i][j]=C[i][j]+A[i][k]*B[k][j]
    }
  }
}
```

# Loop Interchange

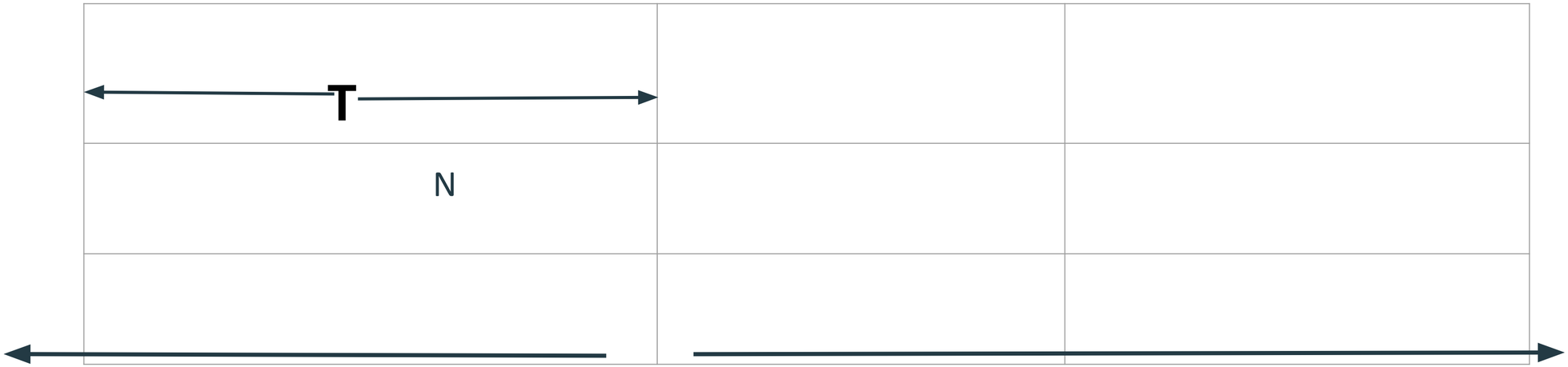
```
for( int i=0;i< N;i++) { //Loop 1
for(int j=0;j<N;j++) { //Loop 2
for(int k=0;k<N;k++){ //Loop 3
        C[i][j]=C[i][j]+A[i][k]*B[k]
[j]
    }
  }
}
```

```
for(int i=0;i<N;j++) { //Loop 1
  for(int k=0;k<N;k++){ //Loop 2
  for( int j=0;j< N;i++) { //Loop 3
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    }
  }
}
```

# Tiling

It is possible to improve data locality
We need to change the execution order of the instruction.
Tiling

- Do not  compute result of a row at a time.
- Divide the matrix into submatrices called **blocks** or **Tiles**.
- Typically blocks are **squares** with dimension **T**.



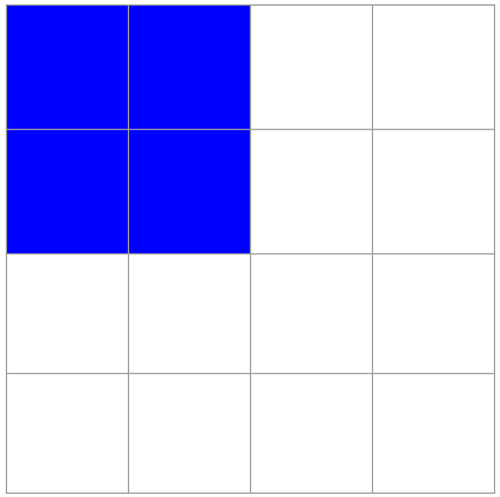
# Matrix Multiplication with Tiling

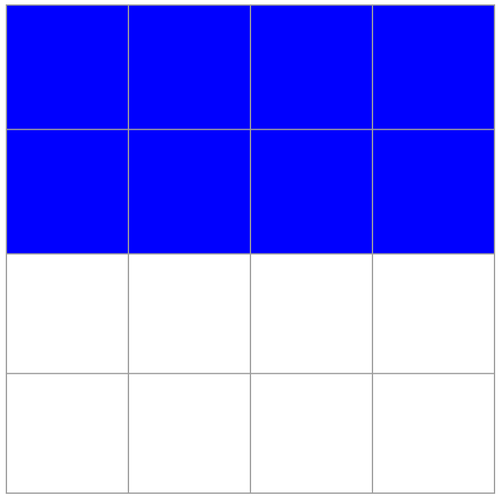
```
for( int ii=0;ii< N ;ii=ii+T)
    for( int jj=0;jj< N ;jj=jj+T)
        for(int kk=0;kk<N;kk=kk+T)
            for(int i=ii;i<ii+T; i++)
                    for(int j=jj;j<jj+T; j++)
                        for(int k=kk;k<kk+T; k++)
                            C[i][j]=A[i][k]*B[k][j]
```

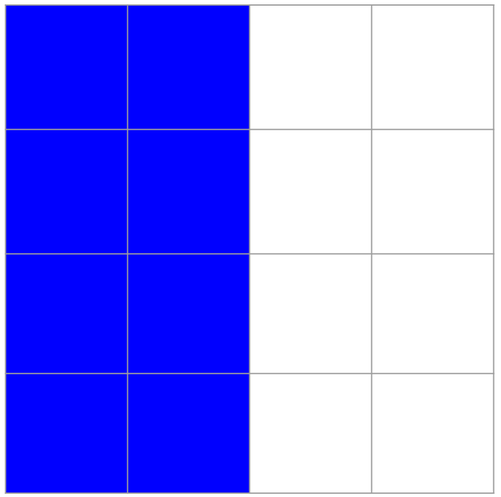
# Tiling: submatrices of  2x2 and matrices of 4x4

Matrices of size 4x4 and submatrices of size of 2x2
Matrices with tiles colored red, blue, brown, and yellow

C = A x B







# Tiling: submatrices of  2x2 and matrices of 4x4

Matrices of size 4x4 and submatrices of size of 2x2
Matrices with tiles colored red, blue, brown, and yellow
A X B = C

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |

| | | | |
|---|---|---|---|
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |

| | | | |
|---|---|---|---|
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |

# Tiling: submatrices of  2x2 and matrices of 4x4

Matrices of size 4x4 and submatrices of size of 2x2
Matrices with tiles colored red, blue, brown, and yellow
A X B = C (ii==2, jj==kk=4)

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |

| | | | |
|---|---|---|---|
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |

| | | | |
|---|---|---|---|
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

# Tiling: submatrices of  2x2 and matrices of 4x4

Matrices of size 4x4 and submatrices of size of 2x2
Matrices with blocked colored red, blue, brown, and  yellow
A X B = C

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 4 |

| | | | |
|---|---|---|---|
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |
| 1 | 3 | 6 | 9 |

| | | | |
|---|---|---|---|
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |
| 10 | 30 | 60 | 90 |

# Block Size

If the block T is taken carefully

- Number of cache misses can be reduced.
- **Choose T so that sub matrices of A,B,C can be stored in cache**.
- Each block of C is only required once in Cache.

To bring entire elements of submatrices of A and B with tile size T

- $T^2/L$ cache misses each, where L is cache line size for each matrix
- $T^3$ multiply-and-add operation performed

Entire matrix multiplication require $N^3$ multiply-and-add operations.

- number of times we need to bring a pair blocks is $N^3 / T^3$
- Total cache misses per matrix is $(T^2/L)x ( N^3 / T^3)= N^3 / (Lx T)$
- **Total misses is $(2x N^3)/( LxT)$ for matrices A and B.**

# Parallelizing the matrix multiplication code

Assign each row (outermost loop) to P processors.

Each processor gets N/P rows for computation.

Memory Access per processor

- needs to access N/P rows of matrix A (i.e $N^2/p$ elements)
- All elements of Matrix B ( $N^2$ elements)
- operation C[i,j] **+=** A[i,k] * B[k,j]
- The access to matrix B does not have variable i.

Computation of Values by Each Processor

- Computes $N^2/P$ elements of matrix C.
- Performs $N^3/P$ multiply-and-add operations.

# Summary and few optional homeworks.

Parallelize outermost loop.
Loop execution order of (i,k,j) with proper Tile size will give high performance.
Also unroll the loop to get more instruction level parallelism.
Small programming assignments

- **compare different scheduling primitives - static, guided, and dynamic.**
  Compare speedup of matrix multiplication with tiling, loop unrolling, and loop interchange. Parallelize the outermost loop.
- **Performance of parallel sections where each section performing one function.**
- **Nested parallelism with OpenMP collapse the manual way.**