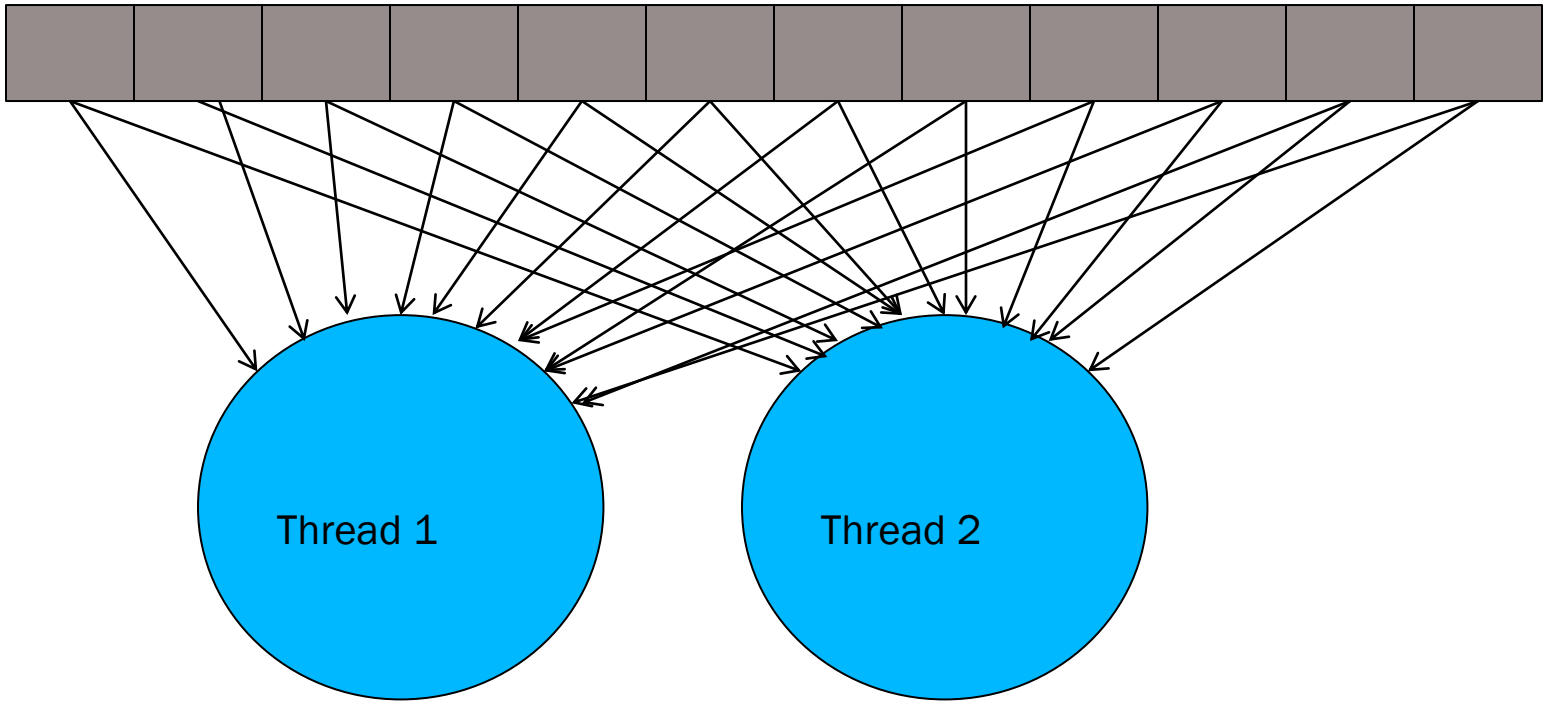
RECITATION

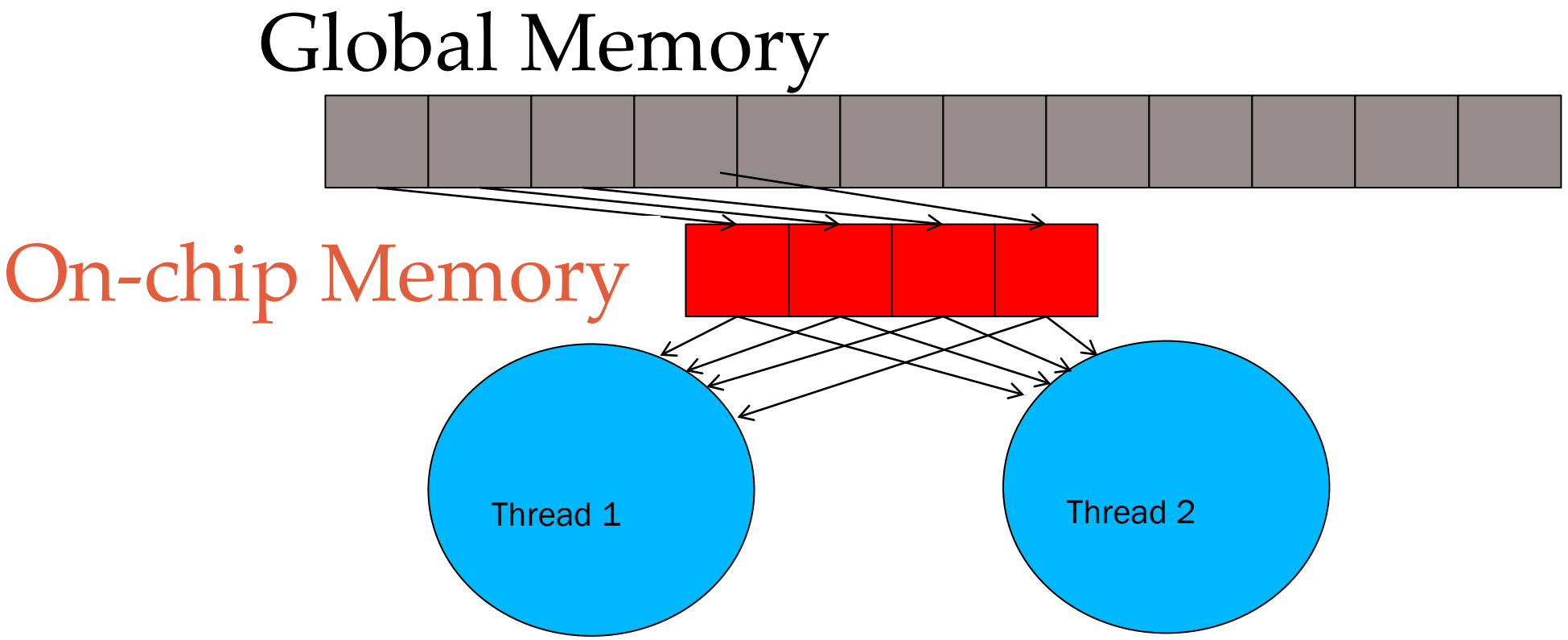
MATRIX MULTIPLICATION (PARALLEL ARCHITECTURE)

GLOBAL MEMORY ACCESS PATTERN OF THE BASIC MATRIX MULTIPLICATION KERNEL

Global Memory



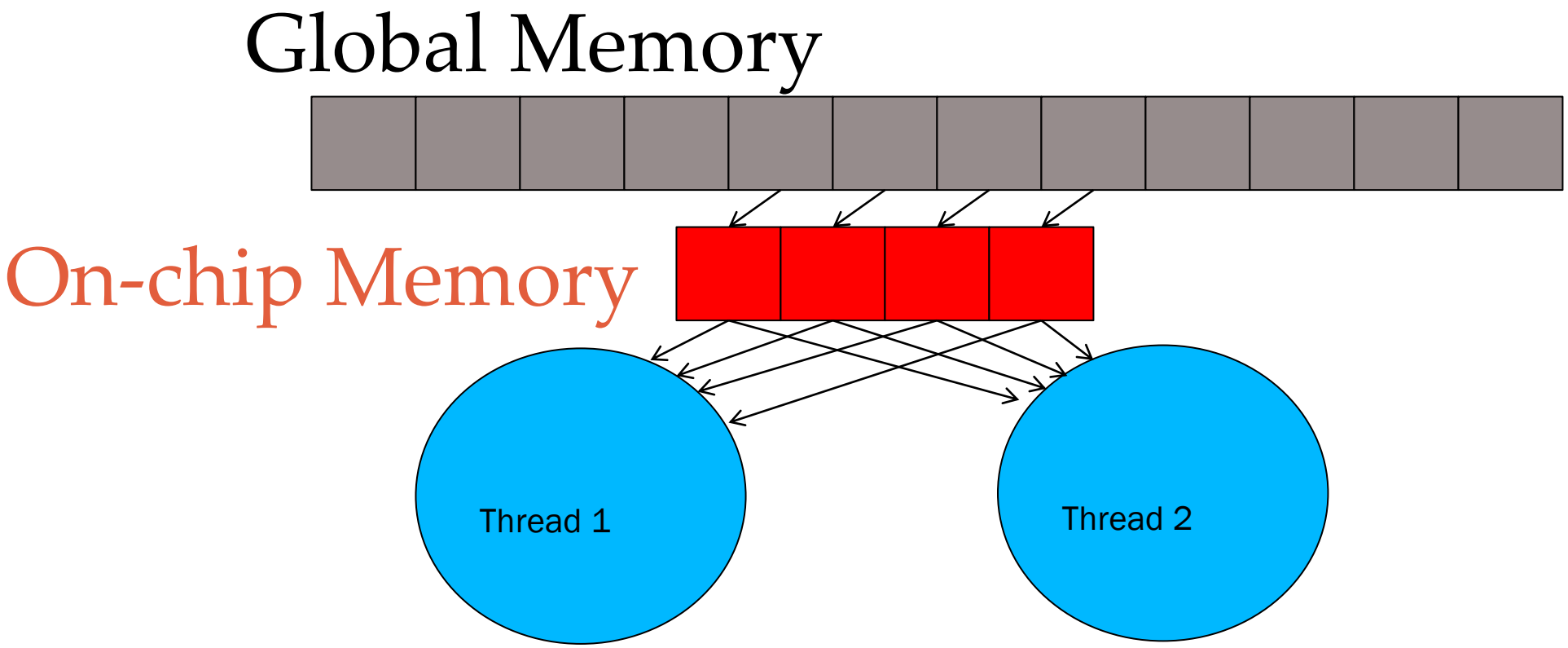
TILING/BLOCKING - BASIC IDEA



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

TILING/BLOCKING - BASIC IDEA



BASIC CONCEPT OF TILING

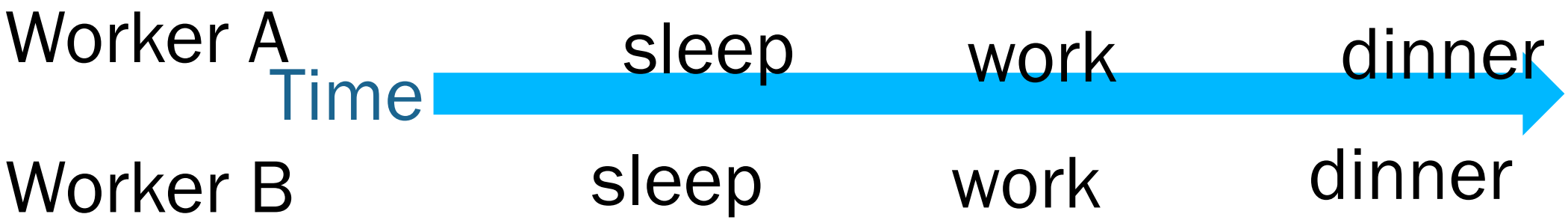
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Tiling for global memory accesses
    - drivers = threads accessing their memory data operands
    - cars = memory access requests

SOME COMPUTATIONS ARE MORE CHALLENGING TO TILE

- Some carpools may be easier than others
  - Car pool participants need to have similar work schedule
  - Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling

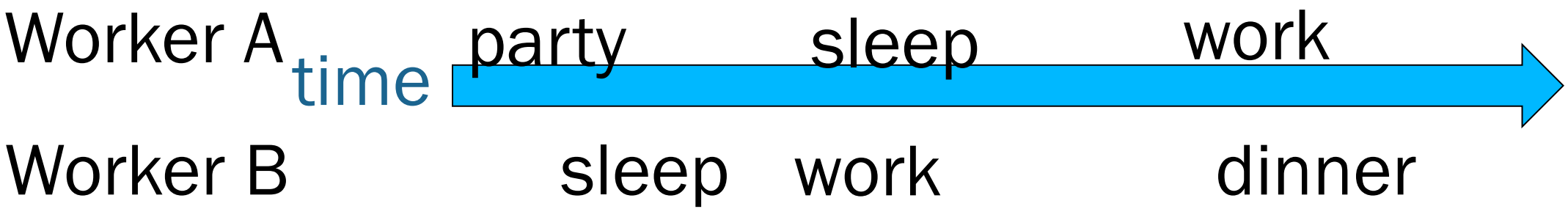
CARPOOLS NEED SYNCHRONIZATION.

- Good: when people have similar schedule



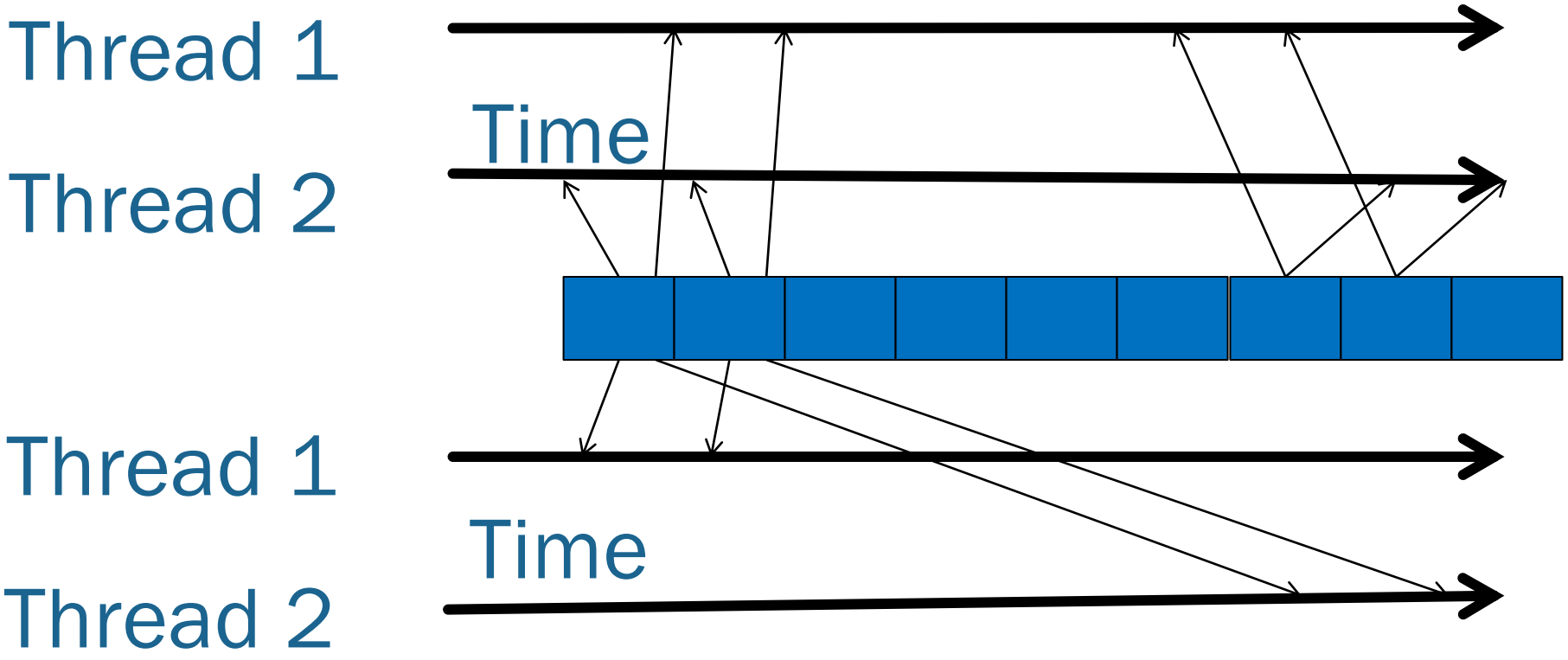
CARPOOLS NEED SYNCHRONIZATION.

- Bad: when people have very different schedule



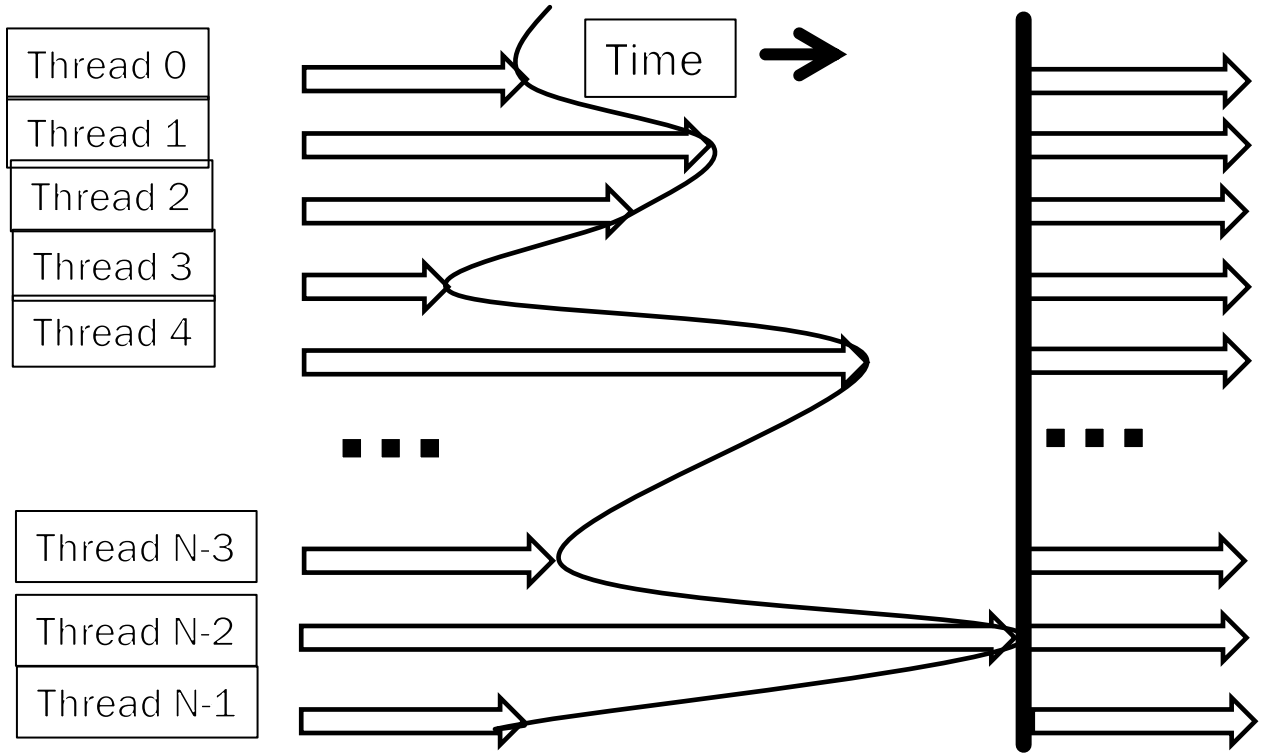
SAME WITH TILING

- Good: when threads have similar access timing



- Bad: when threads have very different timing

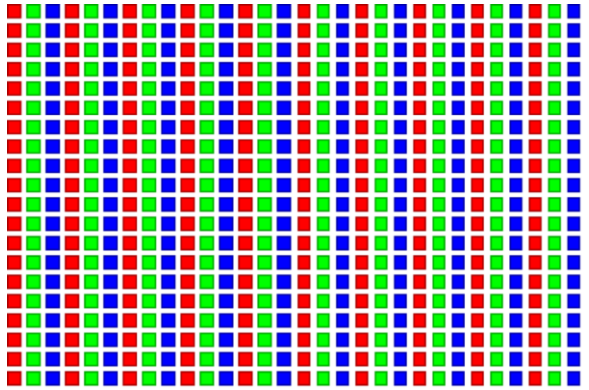
BARRIER SYNCHRONIZATION FOR TILING



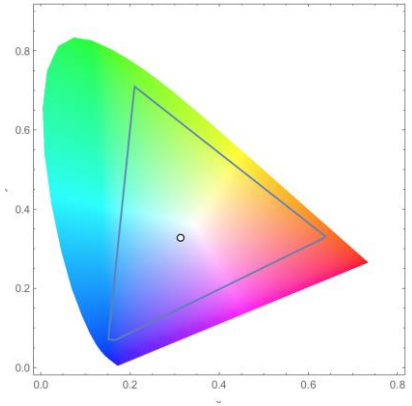
OUTLINE OF TILING TECHNIQUE

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

RGB COLOR IMAGE REPRESENTATION



- Each pixel in an image is an RGB value
- The format of an image's row is
(r g b) (r g b) … (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdbobeRGB color space
  - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R
  - The triangle contains all the representable colors in this color space



RGB TO GRAYSCALE CONVERSION

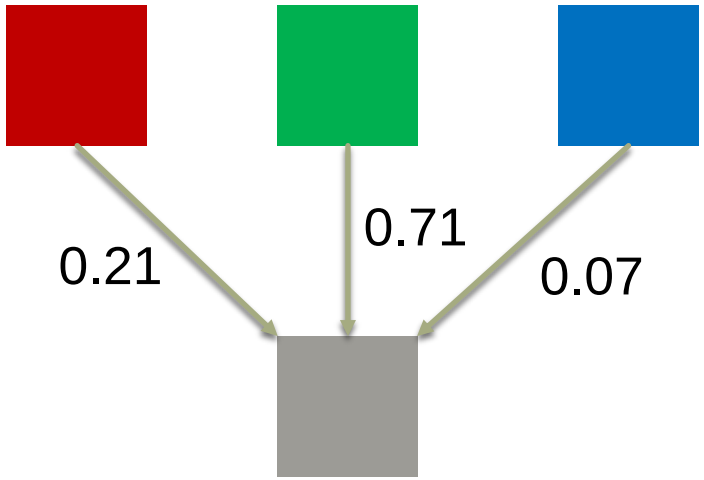
A grayscale digital image is an image in which the value of each pixel carries only intensity information.

COLOR CALCULATING FORMULA

- For each pixel (r g b) at (I, J) do:

grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

- This is just a dot product <[r,g,b],[0.21,0.71,0.07]> with the constants being specific to input RGB space



RGB TO GRAYSCALE CONVERSION CODE

```
                                        // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                                        unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {









 }
}
```

RGB TO GRAYSCALE CONVERSION CODE

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                                    unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
   // get 1D coordinate for the grayscale image
   int grayOffset = y*width + x;
   // one can think of the RGB image having
   // CHANNEL times columns than the gray scale image
   int rgbOffset = grayOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel



 }
}
```

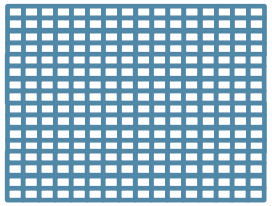
RGB TO GRAYSCALE CONVERSION CODE

```
                                    // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                                        unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
   // get 1D coordinate for the grayscale image
   int grayOffset = y*width + x;
   // one can think of the RGB image having
   // CHANNEL times columns than the gray scale image
   int rgbOffset = grayOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

IMAGE BLURRING

BLURRING BOX



Pixels processed by a thread block

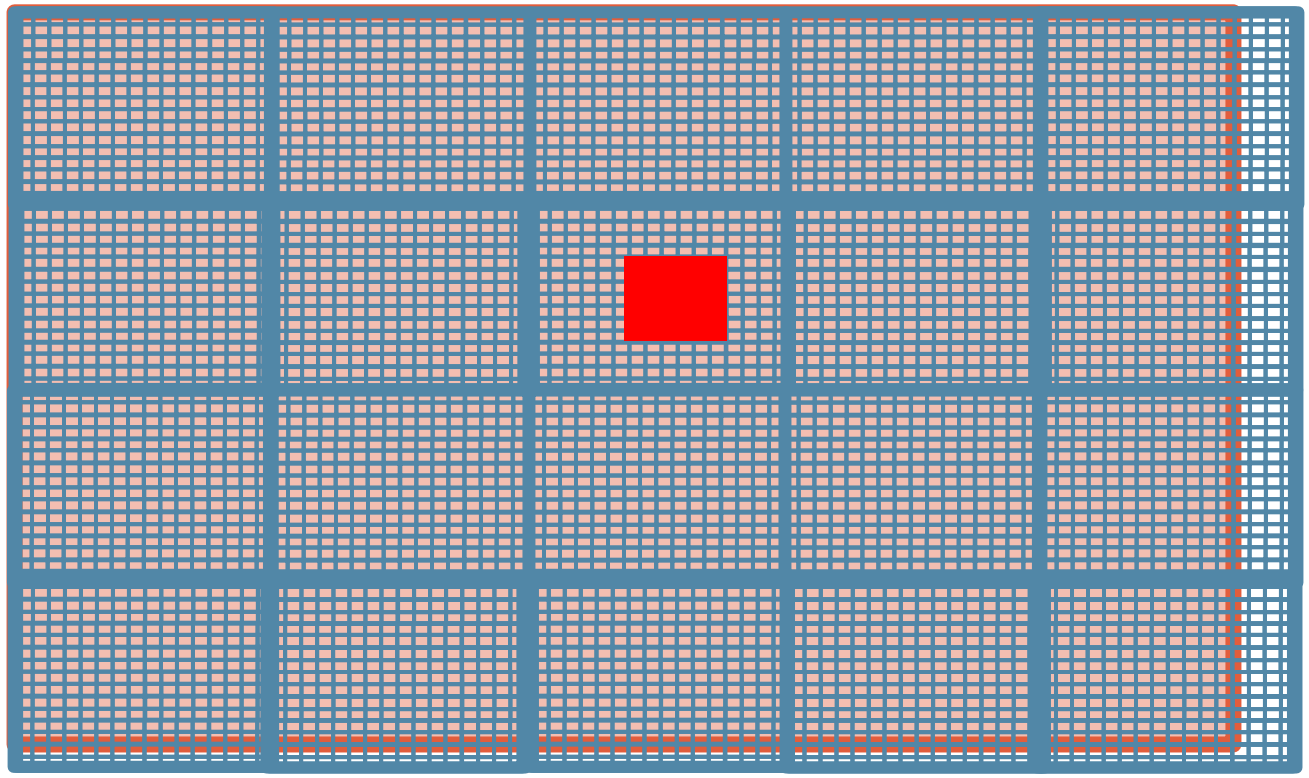


IMAGE BLUR AS A 2D KERNEL

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
      ... // Rest of our kernel
  }
}
```

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++; // Keep track of number of pixels in the accumulated total
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

L8 – PARALLEL ARCHITECTURE

BMCS3003 DISTRIBUTED SYSTEMS AND PARALLEL COMPUTING

CONTENTS

Handler’s Classification

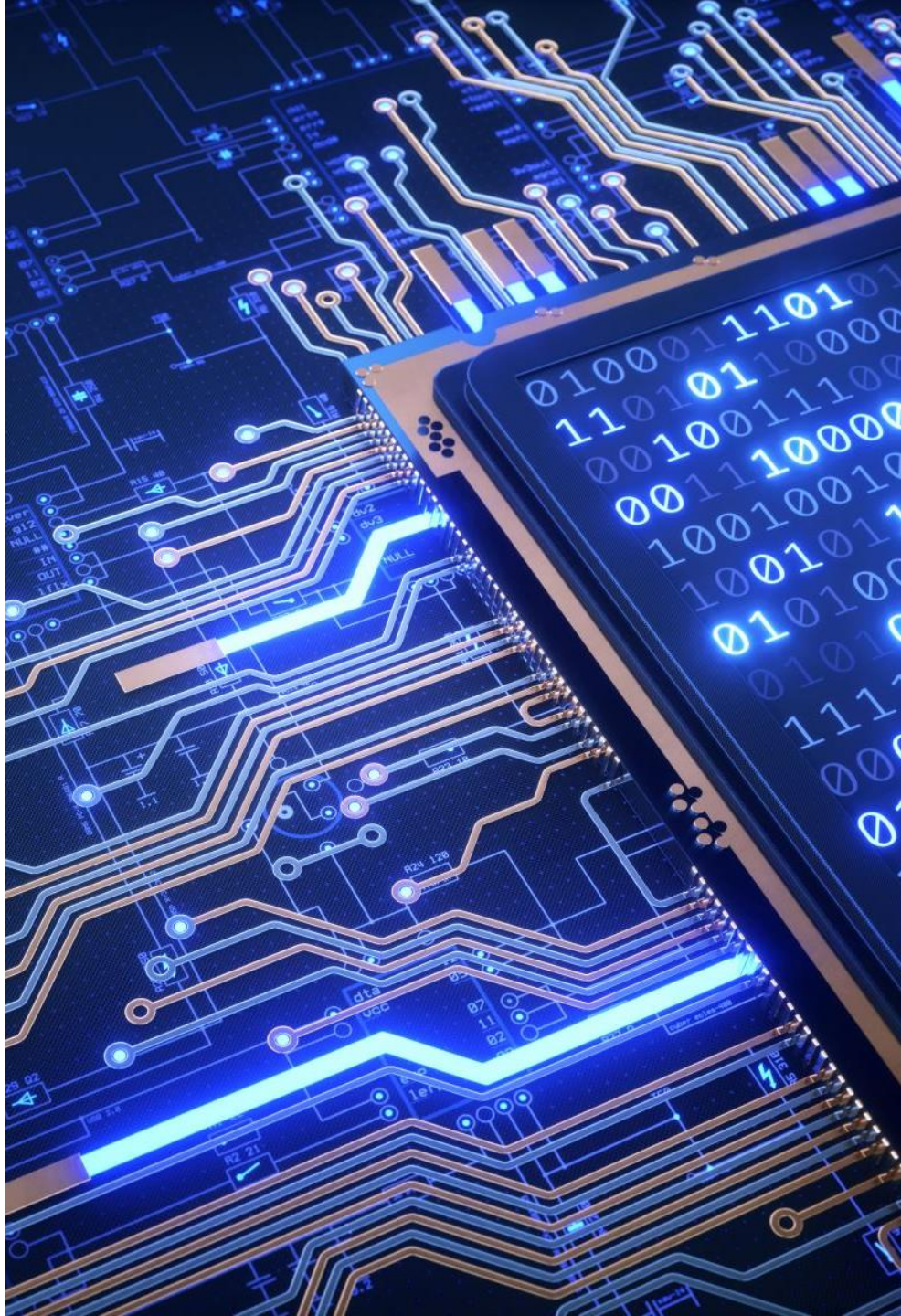
Dependency and its types

Bernstein Conditions for Detecting Parallelism

HANDLER'S CLASSIFICATION

- Wolfgang Handler has proposed a classification scheme for identifying the parallelism degree and pipelining degree built into the hardware structure of a computer system. He considers at three subsystem levels:
  - Processor Control Unit (PCU)
  - Arithmetic Logic Unit (ALU)
  - Bit Level Circuit (BLC)

- Each PCU corresponds to one processor or one CPU. The ALU is equivalent to Processor Element (PE). The BLC corresponds to combinational logic circuitry needed to perform 1 bit operations in the ALU.



LOOP DEPENDENCY

Consumer

Dependencies

Container

Statement 2: b=a+2; “5” Statement 1: a=5;

- Find dependencies within iterations of a loop
- Goal of determining different relationships between statements.
- To allow multiple processors to work on different portions of the loop in parallel
- First analyze the dependencies within individual loops.
- It help determine which statements in the loop need to be completed before other statements can start.
- Two general categories of dependencies: Data and Control dependency

NEED FOR DEPENDENCE ANALYSIS

- Reordering transformations.
  - As the name suggests, these transformations change the order of computations in a program. Reordering transformations are a very powerful class of transformations and can be used at source-level as well as at the intermediate-language level.
  - Consider a simple implementation of matrix-multiply.

```
for i = 1:N {
  for j = 1:N {
    C(i,j) = 0.0;
    for k = 1:N {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Works very well in a scalar machine that executes one instruction at a time

REORDERING TRANSFORMATION

However, suppose that we had a machine that had a 64-word vector floating point unit. Clearly the loop in this form does not take advantage of the vector unit. We could rewrite the code.

```
for i = 1:N {
  for j = 1:64:N {
    C(i,j:j+63) = 0.0;
    for k = 1:N {
       C(i,j:j+63) += A(i,k) * B(k,j:j+63)
    }
  }
}
```

BERNSTEIN'S CONDITIONS

- Since computationally intensive programs are likely to spend most of their time in loops it makes sense to pay attention to improving the performance of loops by parallelizing them. In a 1966 paper, Bernstein established that two iterations I_1 and I_2 can be safely execute in parallel if

1. iteration I_1 does not write into a location that is read by I_2,
2. iteration I_2 does not write into a location that is read by I_1, and
3. iteration I_1 does not write into a location that is also written into by iteration I_2.

- This forms the basis for all loop parallelization algorithms.

```
function Dep(a,b)
c:=a.b
d:=2.c
end function

(flow dependency)
```

```
function Nodep(a,b)
c:=a.b
d:=2.b
e:=a+b
end function

(no dependency)
```

DATA DEPENDENCY

TYPE	NOTATION	DESCRIPTION
True (Flow) Dependence	S1 ->T S2	A true dependence between S1 and S2 means that S1 writes to a location later read from by S2
Anti Dependence	S1 ->A S2	An anti-dependence between S1 and S2 means that S1 reads from a location later written to by S2.(before)
Output Dependence	S1 ->I S2	An input dependence between S1 and S2 means that S1 and S2 read from the same location.

True dependence

S0: int a, b;

S1: a = 2;

S2: b = a + 40;

S1 ->T S2, meaning that S1 has a true dependence on S2 because S1writes to the variable a, which S2 reads from.

Anti-dependence

S0: int a, b = 40;

S1: a = b - 38;

S2: b = -1;

S1 ->A S2, meaning that S1 has an anti-dependence on S2 because S1 reads from the variable b before S2 writes to it.

Output-dependence

S0: int a, b = 40;

S1: a = b - 38;

S2: a = 2;

S1 ->O S2, meaning that S1 has an output dependence on S2 because both write to the variable a.

ANTI DEPENDENCES

- WAR - Write-after-read

```
Sum = a+1;

First_term = sum*scale1;

Sum = b+1;

Second_term = sum*scale2;
```

- This is false dependence
- Rewrite with variable renaming

```
Sum1=a+1;

First_term=sum1*scale1;

Sum2 = b+1;

Second_term = sum2*scale2;
```

OUTPUT DEPENDENCES

- WAW - Write-after-write

```
Sum = a+1;

First_term = sum*scale1;

Sum = b+1;

Second_term = sum*scale2;
```

- This is false dependence
- Rewrite with variable renaming

```
Sum1=a+1;

First_term=sum1*scale1;

Sum2 = b+1;

Second_term = sum2*scale2;
```

CONTROL DEPENDENCY

Control flow gives rise to control dependences. Consider the following code.

```
        for ( ... ) {
S1        if (d == 0) continue;
S2        x = x / d;
          ...
        }
```

Here, the statement S_1 must be executed before S_2, since the execution of S_2 is conditional upon the execution of the branch in S_1. Executing S_2 before S_1 could generate a divide-by-zero error that would be impossible in the original code.

DEPENDENCY IN LOOP

Loops can have two types of dependence:

- Loop-carried dependency
- Loop-independent dependency

LOOP CARRIED DEPENDENCY

- In loop-carried dependence, statements in an iteration of a loop depend on statements in another iteration of the loop.

```
for(i=0;i<4;i++)
{
S1: b[i]=8;
S2: a[i]=b[i-1] + 10;
}
```

LOOP INDEPENDENT DEPENDENCY

- In loop-independent dependence, loops have inter-iteration dependence, but do not have dependence between iterations.
- Each iteration may be treated as a block and performed in parallel without other synchronization efforts.

```
for (i=0;i<4;i++)
{
S1: b[i] = 8;
S2: a[i] =b[i] + 10;
}
```

SIMPLE RULE OF THUMB

- A loop that matches the following criteria has no dependences and can be parallelized:
  - All assignments (to shared data) are to arrays
  - Each element is assigned by at most one iteration
  - No iteration reads elements assigned by any other iteration

ARE THESE LOOPS PARALLEL?

1.
```
for (i=1;i<N;i+=2)
    a[i] = a[i]+a[i-1];
```
2.
```
for (i=0;i<N/2;i++)
    a[i] = a[i]+a[i+N/2];
```
3.
```
for (i=0;i<=N/2;i++)
    a[i] = a[i]+a[i+N/2];
```
4.
```
for (i=0;i<N;i++)
    a[idx[i]] = a[idx[i]]+b[idx[i]];
```

LOOP PARALLELIZATION

- Extraction parallel tasks from loops
- Data is stored in random access data structures
- A program exploiting loop-level parallelism will use multiple threads or processes which operate on same time
- It provides speedup
- Amdhal’s law

REMOVING DEPENDENCES

```
for (i=0;i<N;i++){
   x=(b[i]+c[i])/2;
   a[i]=a[i+1]+x;
}
```

What kind of dependences are there?

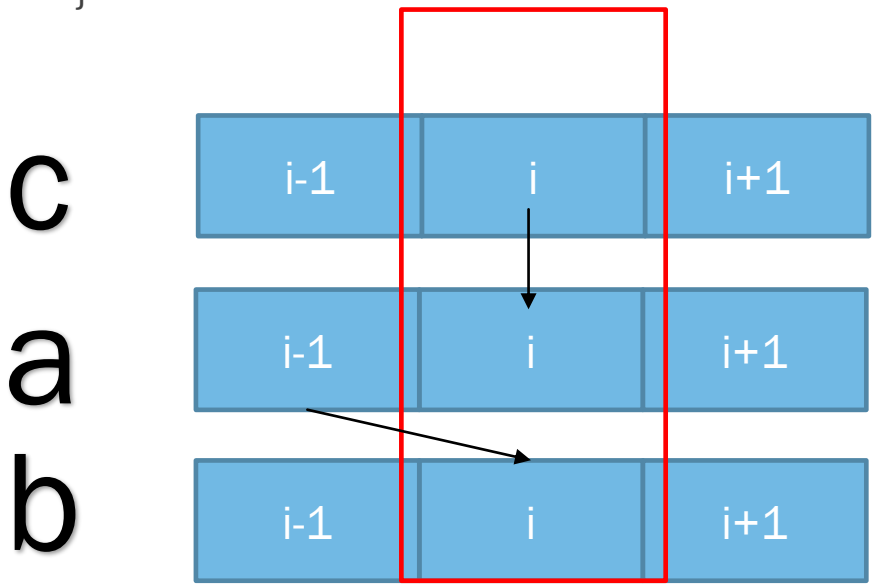
SAME LOOP MADE PARALLEL

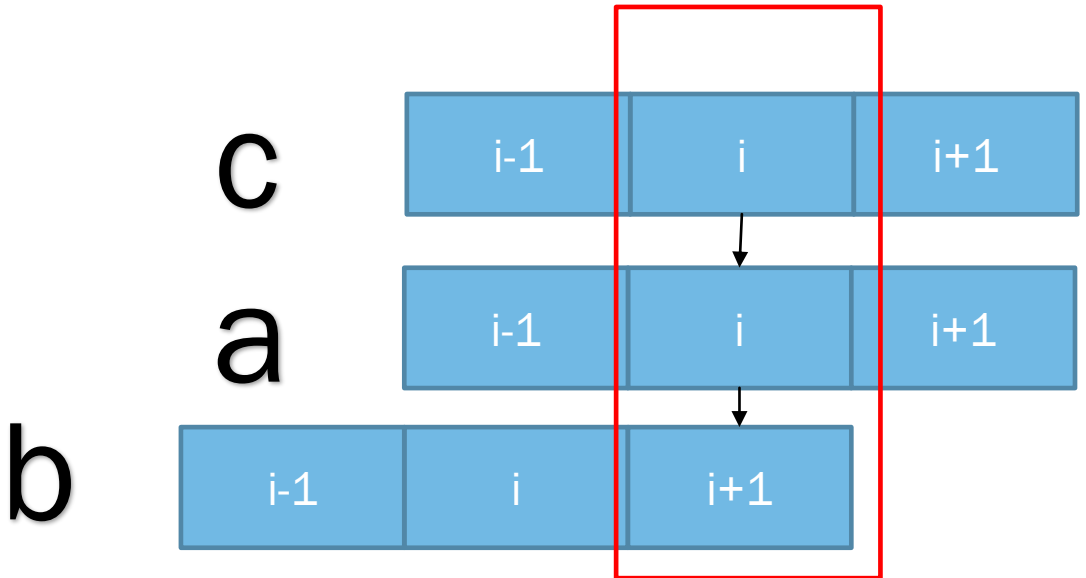
```
#pragma omp parallel for
for (i=0;i<N;i++){
a2[i]=a[i+1];
}
#pragma omp parallel for private(x)
for (i=0;i<N;i++{
x=(b[i]+c[i])/2;
a[i]=a2[i]+x;
```

- Space and extra data movement is traded for parallelism

WHAT ARE THE DEPENDENCES?

```
for(i=1;i<N;i++){
b[i]+=a[i-1];
a[i]+=c[i];
}
```





USE SOFTWARE PIPELINING TO MAKE IT PARALLEL

```
b[1]+=a[0];
#pragma omp parallel for
for (i=1;i<N-1;i++){
a[i]+=c[i];
b[i+1]+=a[i];
}
a[N]+=c[N];
```

BERNSTEIN CONDITIONS FOR DETECTION OF PARALLELISM,

BERNSTEIN CONDITIONS FOR DETECTION OF PARALLELISM

- For implementation of instructions or block of instructions in parallel, it should be guaranteed that the instructions are independent of each other. These instructions can be / control dependent / data dependent resource dependent on each other. We are considering only data dependency between the statements for taking decisions of parallel implementation. A.J. Bernstein has implicated the work of data dependency and resultant some conditions based on which we can settle on the parallelism of processes or instructions.
- Bernstein conditions are rely on the subsequent two sets of variables:
- i) The input set or Read set RI that consists of memory locations examine by the statement of instruction I_1.
- ii) The output set or Write set WI that consists of memory locations printed into by instruction I_1.
- The sets W_I and R_I are not disjoint as the similar locations are used for writing and reading by S_I.

BERNSTEIN PARALLELISM CONDITIONS WHICH ARE DETERMINE TO CONCLUDE WHETHER THE STATEMENTS ARE PARALLEL OR NOT:

1. Position in R_1 from which S_1 reads and the Position W_2 onto which S_2 writes must be commonly exclusive. It means S_1does'nt read from any memory location onto which S_2 writes. It can be indicated as:

$$R_1 \cap W_2 = \emptyset$$

2. R_1 Likewise, locations in R_2 from which S_1 writes and the locations W_1 onto which S_2 reads must be commonly exclusive.It means S_2 does not read from some memory location onto which S_1 writes. It can be indicate as:

$$R_2 \cap W_1 = \emptyset$$

3. The memory locations W_1 and W_2 onto which S_1 and S_2 write, should not be examine by S_1 and S_2. It means R_1 and R_2 should be free of W_1 and W_2. It can be indicated as :

$$W_1 \cap W_2 = \emptyset$$

LET CONSIDER THE FOLLOWING INSTRUCTIONS OF SEQUENTIAL PROGRAM:

$I_1 : x = (a + b) / (a * b)$

$I_2 : y = (b + c) * d$

$I_3 : z = x2 + (a * e)$

Now, the read set and write set of I1, I2 and I3 are as given:

R_1 = {a,b} W1 = {x}

R_2 = {b,c,d} W2 = {y}

R_3 = {x,a,e} W3 = {z}