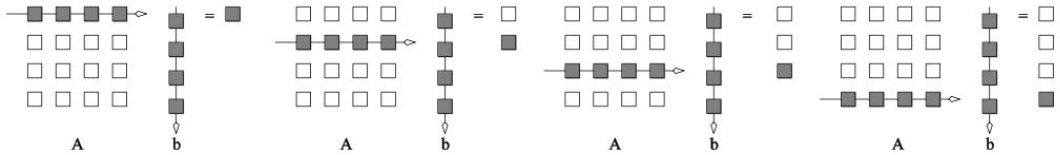
Discussion #3:

WS #5, Lab #1, and Memory Fence

Daniel Tan, Jason Lau, and Jason Cong

Worksheet #5

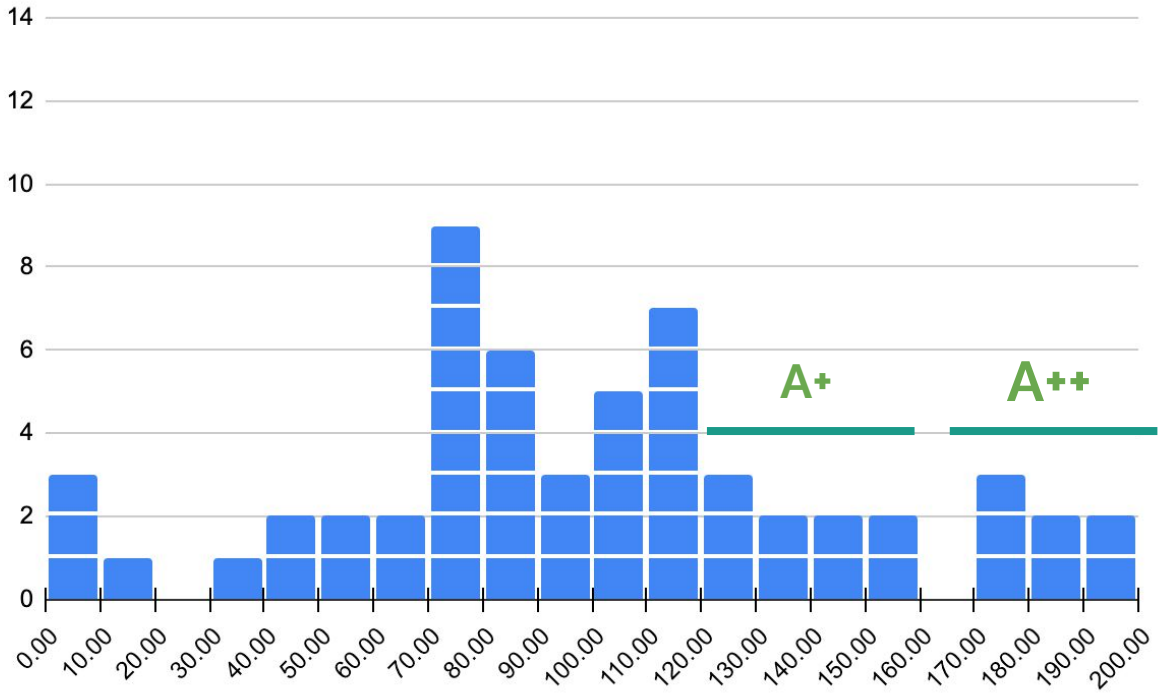
(a) Column major data access



(b) Row major data access.

Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Lab #1: Performance Distribution



A++ Students

1	Poggers	199.43
2	Teem	192.37
3	sumedha	182.95
4	Speed Cola	181.95
5	m	178.73
6	AT	176.37
7	Yingge He	176.14

Lab #1: Things to Consider

- Which loops to parallelize? Outer loop
- Static, dynamic or guided scheduling?
- How many threads? 8 threads

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < kI; i++)
    for (int j = 0; j < kJ; j++)
        for (int k = 0; k < kK; k++)
            c[i][j] += a[i][k] * b[k][j];
```

0.74
GFLOPS

3.89x
Sequential

3.89x
Improvements

5/25

Lab #1: Things to Consider

- Which loops to parallelize? Outer loop
- Static, dynamic or guided scheduling?
- How many threads? 8 threads
- **Row major access or column major?**

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < kI; i++)
    for (int k = 0; k < kK; k++)
        for (int j = 0; j < kJ; j++)
            c[i][j] += a[i][k] * b[k][j];
```

36.74
GFLOPS

193x
Sequential

49.6x
Improvements

15/25

Lab #1: Why Blocked Design?

	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

Figure: Intel https://software.intel.com/content/www/us/en/develop/articles/efficient-use-of-tiling.html

Lab #1: Why Blocked Design?

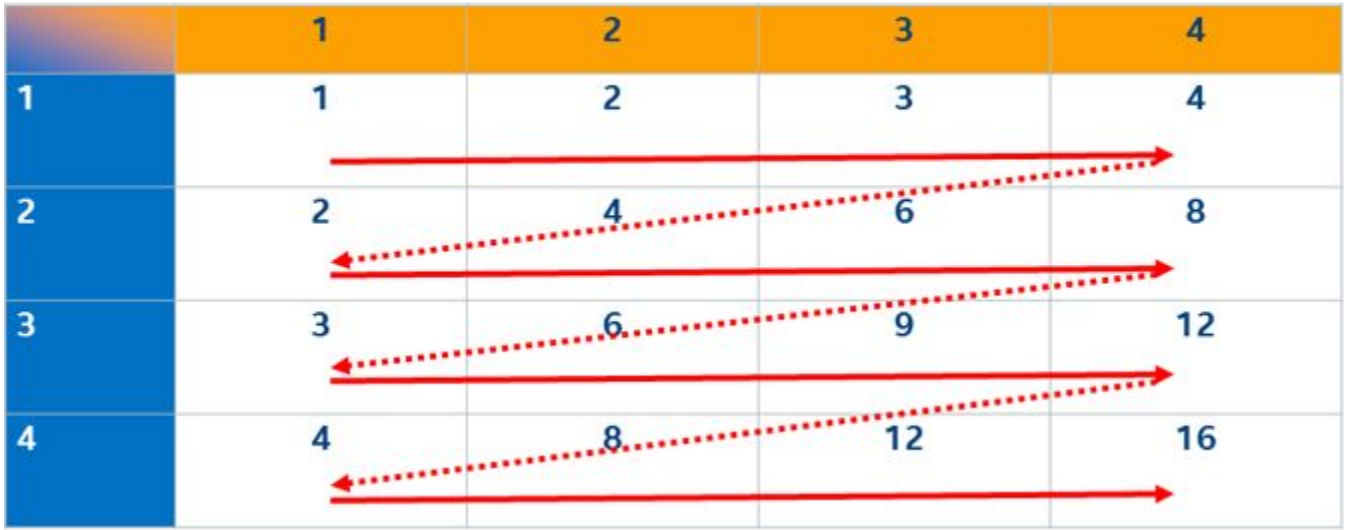


Figure: Intel https://software.intel.com/content/www/us/en/develop/articles/efficient-use-of-tiling.html

Lab #1: Why Blocked Design?

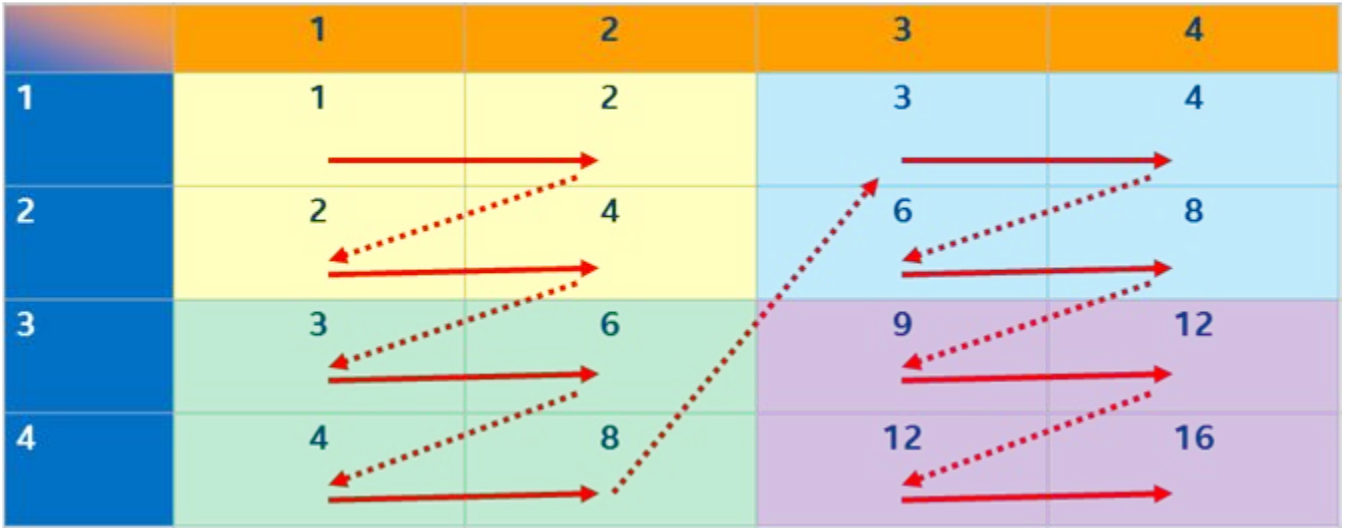
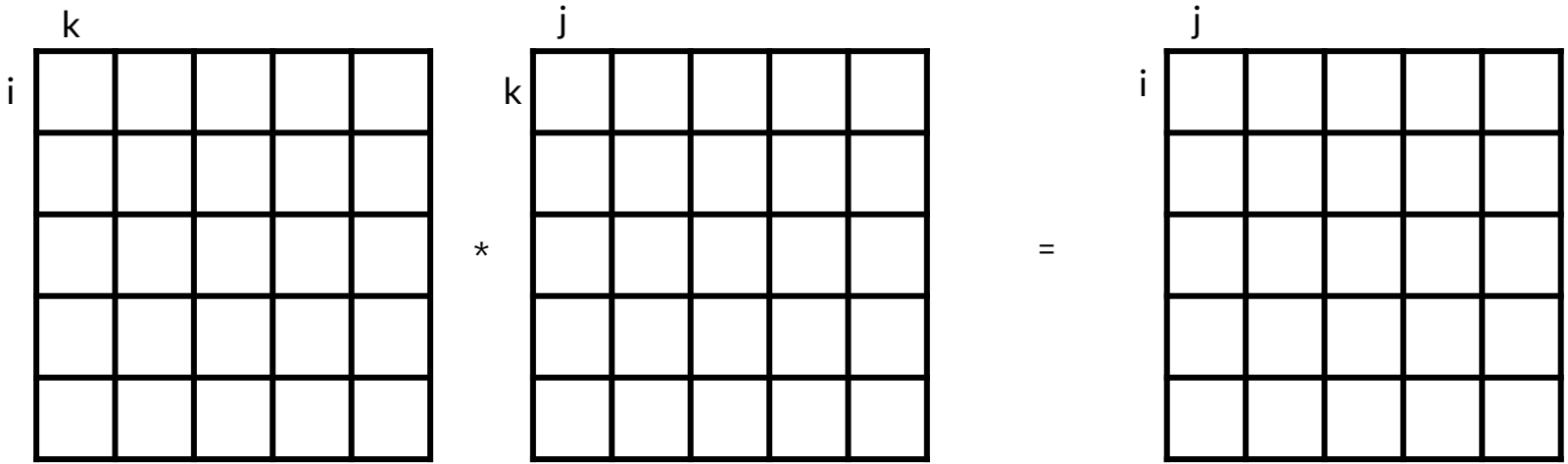


Figure: Intel https://software.intel.com/content/www/us/en/develop/articles/efficient-use-of-tiling.html

Lab #1: Blocked Design?

8192 numbers in cache

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < kI; i++)
    for (int k = 0; k < kK; k++)
        for (int j = 0; j < kJ; j++)
            c[i][j] += a[i][k] * b[k][j];
```



Lab #1: Blocked Design?

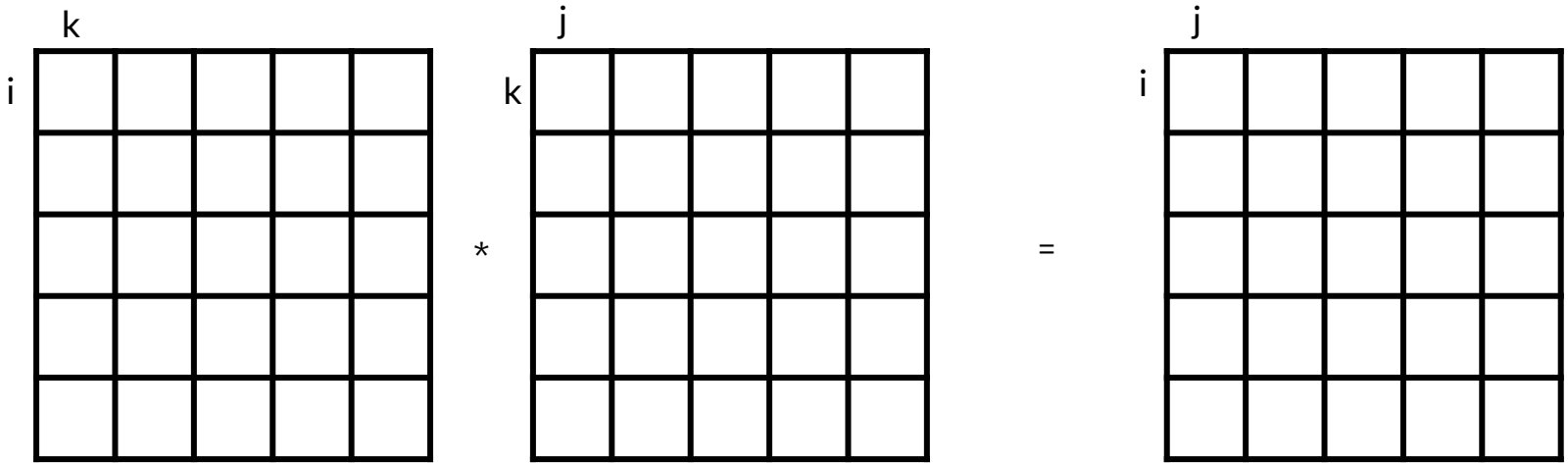
8192 numbers in cache

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < kI; i++)
    for (int k = 0; k < kK; k++)
        for (int j = 0; j < kJ; j++)
            c[i][j] += a[i][k] * b[k][j];
```

$B_i = 64$

$B_j = 64$

$B_k = 32$



Lab #1: Things to Consider

- **Blocked design**

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 32)
for (int jj = 0; jj < kJ; jj += 64)
for (int i = 0; i < kI; i++)
    for (int k = 0; k < kK; k++)
        for (int j = 0; j < kJ; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Lab #1: Things to Consider

- Blocked design

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 32)
for (int jj = 0; jj < kJ; jj += 64)
for (int i = ii; i < ii + 64; i++)
    for (int k = kk; k < kk + 32; k++)
        for (int j = jj; j < jj + 64; j++)
            c[i][j] += a[i][k] * b[k][j];
```

63.94
GFLOPS

337x
Sequential

1.74x
Improvements

20/25 **95/100**

Lab #1: Blocked Design?

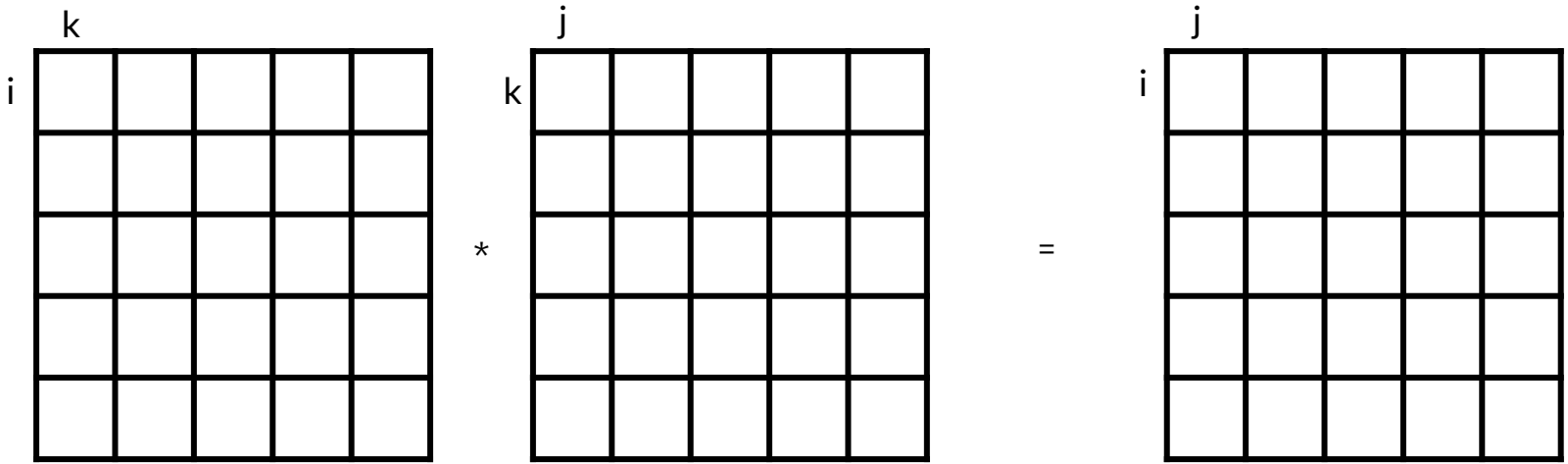
8192 numbers in cache

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < kI; i++)
    for (int k = 0; k < kK; k++)
        for (int j = 0; j < kJ; j++)
            c[i][j] += a[i][k] * b[k][j];
```

$B_i = 64$

$B_j = 1024$

$B_k = 8$



Lab #1: Things to Consider

- **Blocked design**

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int k = kk; k < kk + 8; k++)
        for (int j = jj; j < jj + 1024; j++)
            c[i][j] += a[i][k] * b[k][j];
```

85.09
GFLOPS

448x
Sequential

1.33x
Improvements

20/25

Lab #1: Further Optimization

- **Blocked design**
- **Put data in register**

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int k = kk; k < kk + 8; k++)
        for (int j = jj; j < jj + 1024; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Lab #1: Further Optimization

- Blocked design
- Put data in register

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int k = kk; k < kk + 8; k++) {
        float reg = a[i][k];
        for (int j = jj; j < jj + 1024; j++)
            c[i][j] += reg * b[k][j];
    }
```

104.6
GFLOPS

547X
Sequential

1.23X
Improvements

25/25

Lab #1: Further Optimization

- **Blocked design**
- **Put data in register**

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int k = kk; k < kk + 8; k++)
        for (int j = jj; j < jj + 1024; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Lab #1: Further Optimization

- **Blocked design**
- **Put data in register**

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int j = jj; j < jj + 1024; j++)
        for (int k = kk; k < kk + 8; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Lab #1: Further Optimization

- Blocked design
- Put data in register

```
#pragma omp parallel for schedule(static)
for (int ii = 0; ii < kI; ii += 64)
for (int kk = 0; kk < kK; kk += 8)
for (int jj = 0; jj < kJ; jj += 1024)
for (int i = ii; i < ii + 64; i++)
    for (int j = jj; j < jj + 1024; j++) {
        float reg = c[i][j];
        for (int k = kk; k < kk + 8; k++)
            reg += a[i][k] * b[k][j];
        c[i][j] = reg;
    }
```

144.1
GFLOPS

768x
Sequential

1.65x
Improvements

28/25

Lab #1: Further Optimization

- Check all possibilities and fine tune

```
#pragma omp parallel for schedule(static) num_threads(4)
for (int ii = 0; ii < kI; ii += 64)
for (int jj = 0; jj < kJ; jj += 1024)
for (int kk = 0; kk < kK; kk += 8)
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 1024; j++) {
        float reg = c[i + ii][j + jj];
        for (int k = 0; k < 8; k++)
             reg += a[i+ii][k+kk] * b[k+kk][j+jj];
        C[i + ii][j + jj] = reg;
    }
```

204.3
GFLOPS

1075x
Sequential

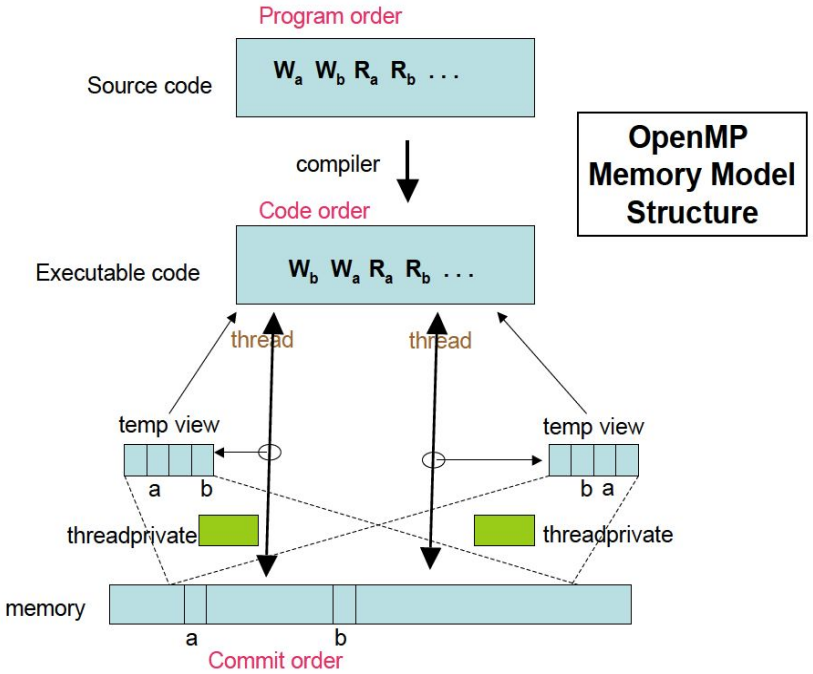
1.42x
Improvements

30/25

Memory Fence in OpenMP

- Relaxed-consistency, shared-memory model
  - All threads have access to the ***memory*** store and retrieve variables
  - Each thread is allowed to have its own ***temporary view*** of the memory
    - If multiple threads write to the same shared variable without synchronization, the resulting value of the variable in memory is unspecified
    - If at least one thread reads from a shared variable and at least one thread writes to it without synchronization, the value seen by any reading thread is unspecified
  - ***Threadprivate memory*** must not be accessed by other threads

OpenMP Memory Model Structure



Hoeflinger, de Supinski, “The OpenMP Memory Model,” IWOMP 2005.

Memory Fences in OpenMP

- #pragma omp flush (<variable-list>)
  - All the shared variables will be flushed if not given the list
  - Automatically inserted at most points where it is needed
    - barrier
    - parallel - upon entry and exit
    - critical - upon entry and exit
    - ordered - upon entry and exit
    - for - upon exit
    - sections - upon exit
    - single - upon exit
    - Read/write an object with a volatile-qualified type
    - Section 2.7.5 in OpenMP Specification 2.5

Example: flush **Pragma**

```
int flag[MAX];
int i;

for (i = 0; i < MAX; i++)
    flag[i] = 0;

#pragma omp parallel shared(flag)
{
      int ID = omp_get_thread_num();
      do_a_wholebunch(ID);
      flag[ID] = 1;
#pragma omp flush(flag[ID]);
      while (!flag[neighbor(ID)]) {
#pragma omp flush(flag[neighbor(ID)]);
      }
      do_more_stuff();
}
```

- How thread A reads the shared variable *x* written by thread B?
  - Thread B writes the value to the variable *x*
  - Thread B flushes the variable *x*
  - Thread A flushes the variable *x*
  - Thread A read the value from the variable *x*

Importance of Variable List in Flush

Incorrect example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush`(b)`	*flush*`(a)`
flush`(a)`	*flush*`(b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`

Correct example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush`(a,b)`	*flush*`(a,b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`

The flush operation has two effects:

I. make the temporary view and share memory consistent, and

II. impose some ordering constraints on the memory accesses.

In the first implementation, flush(b) in thread 1 and flush(a) in thread 2 might be moved to the end by the compiler, resulting both of them entering the critical sections at the same time.

Q&A