Shared Memory Programming Model

Ahmed El-Mahdy and Waleed Lotfy

KFUPM training delivered by Brightskies Technologies

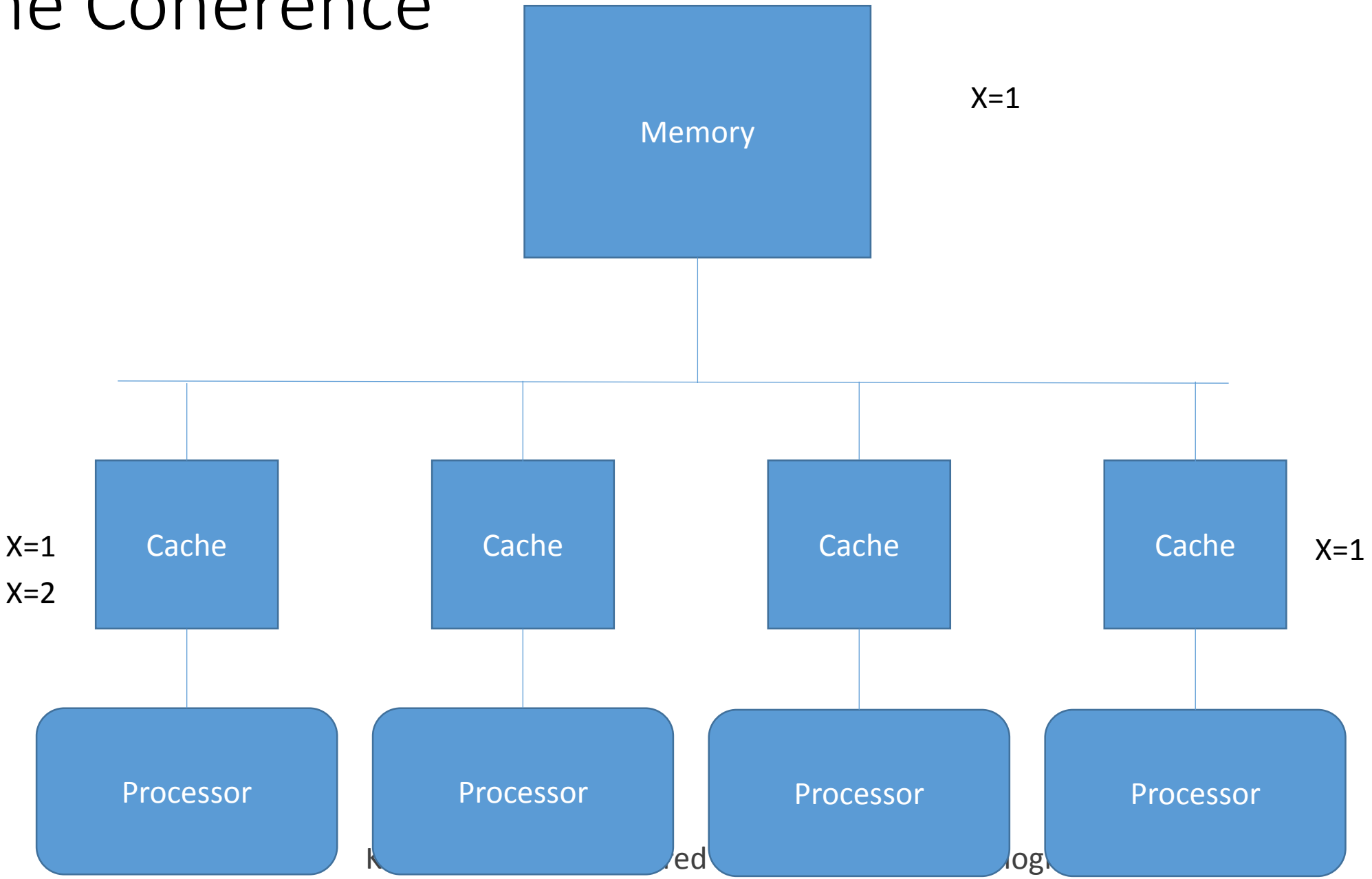
What is a shared memory system?

- Activity!
- Consider the board as a shared memory
- Consider a sheet of paper in front of you as a local cache
- Assume that each row is just one processor

Issues

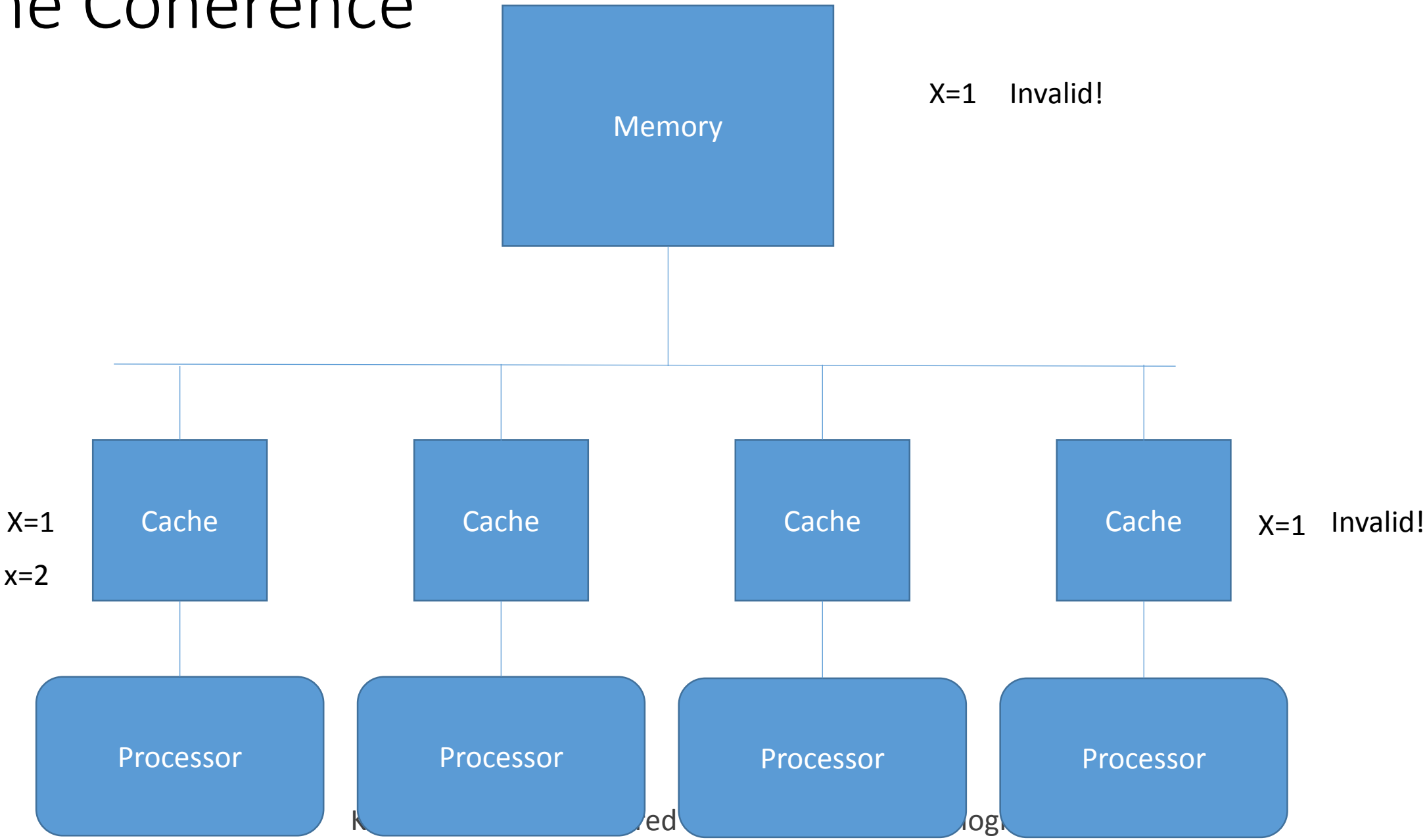
- Counting across rows results in an error,
- Such error is because the caches are not coherent

Cache Coherence



That is usually handled with invalidation protocols

Cache Coherence



But…

- Let’s try another activity
- Assume that the cache line contains 10 row counts what happens?

False Sharing

- It significantly affects performance, almost no scaling

OpenMP

KFUPM training delivered by Brightskies Technologies

Flynn’s Taxonomy

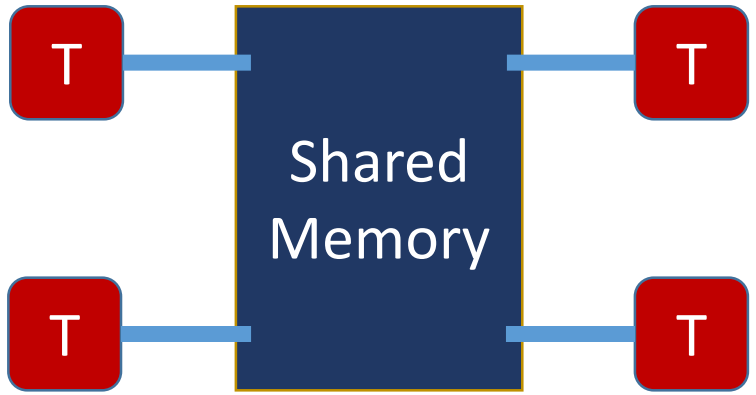
- Classifies computer architectures to:

1. SISD
2. SIMD
3. MISD
4. MIMD

Memory architecture

- Memory types can be classified into:

1. Shared memory
2. Distributed memory



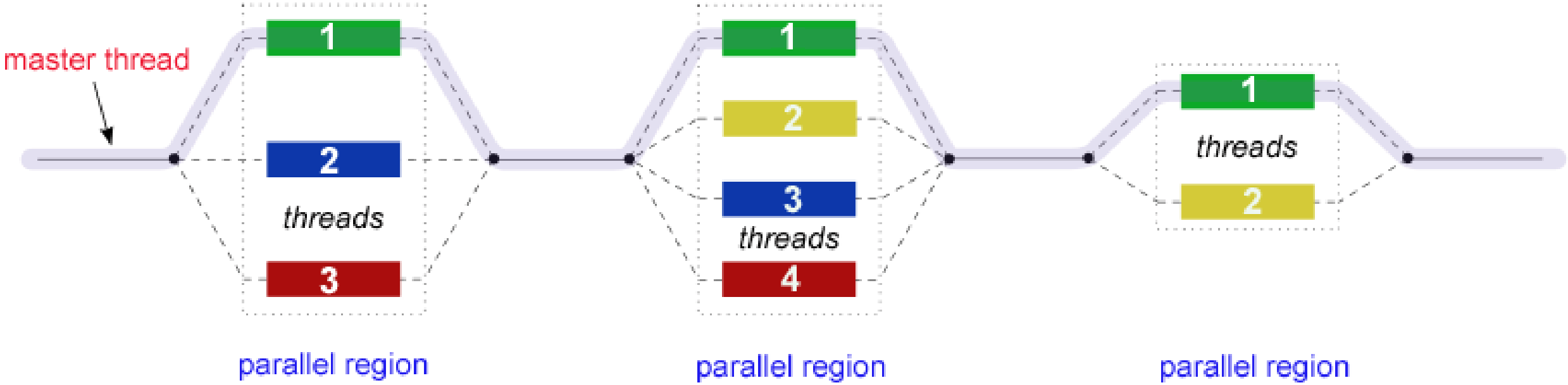
What is OpenMP?

Open Multi-Processing

An API for writing shared memory multithreaded applications.

Fork - Join Model

OpenMP uses fork-join model of parallel execution

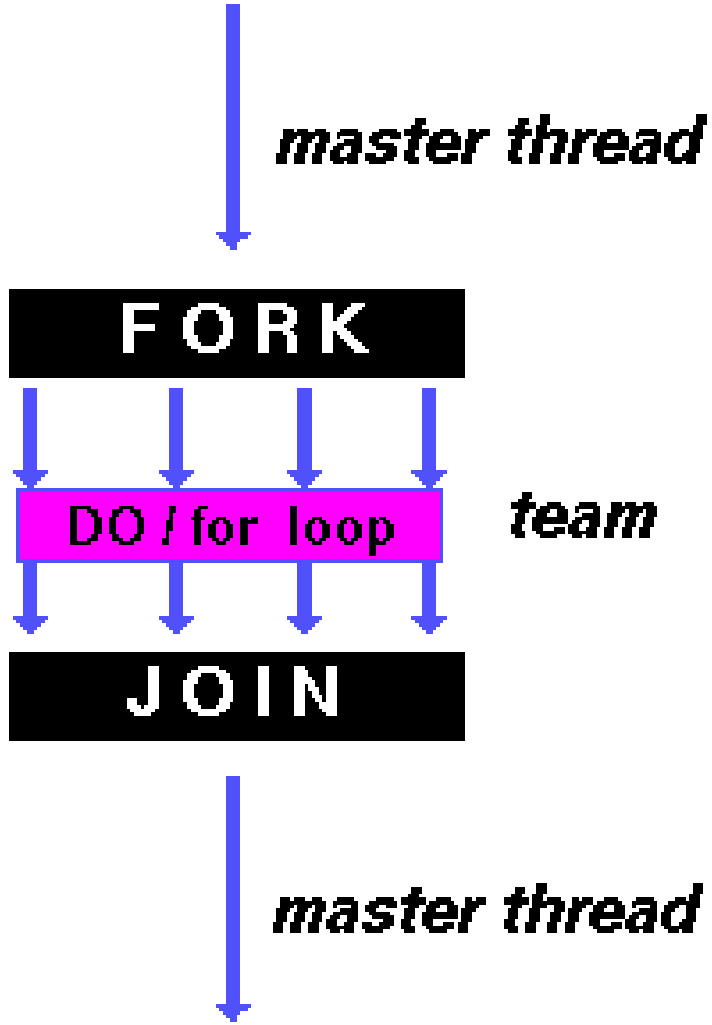


Work-Sharing Constructs

- Work-sharing constructs control how work is divided among a team of threads.
- Work-sharing constructs communicate with the compiler through compiler directives.

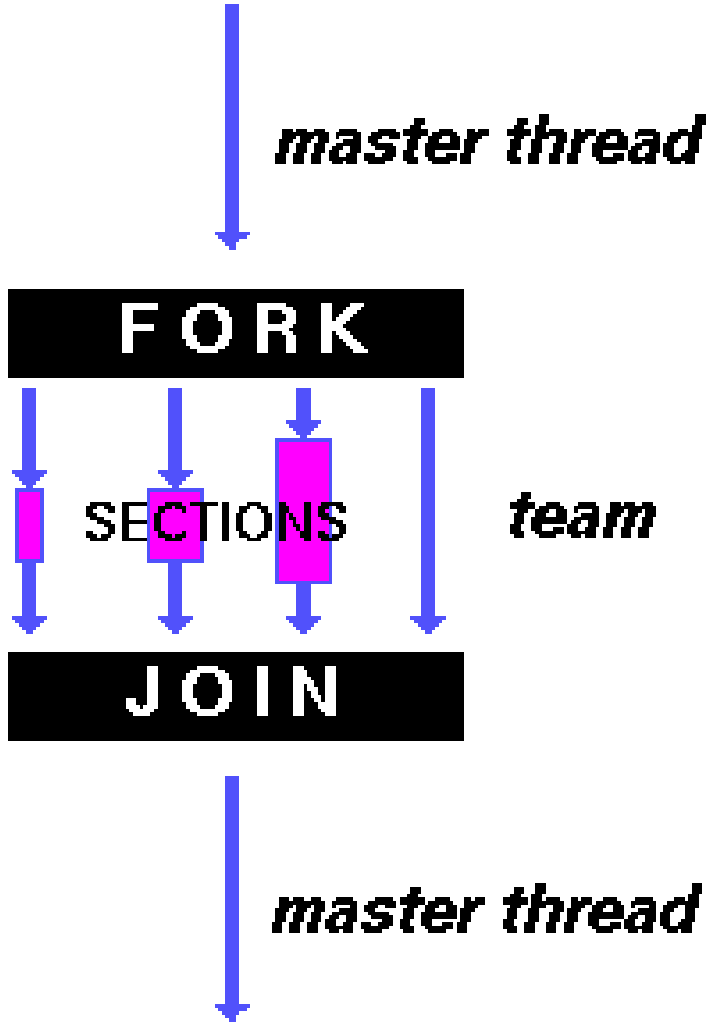
Work-Sharing Constructs

Do / for
Distributes execution of iterations among team of threads
Data parallelism



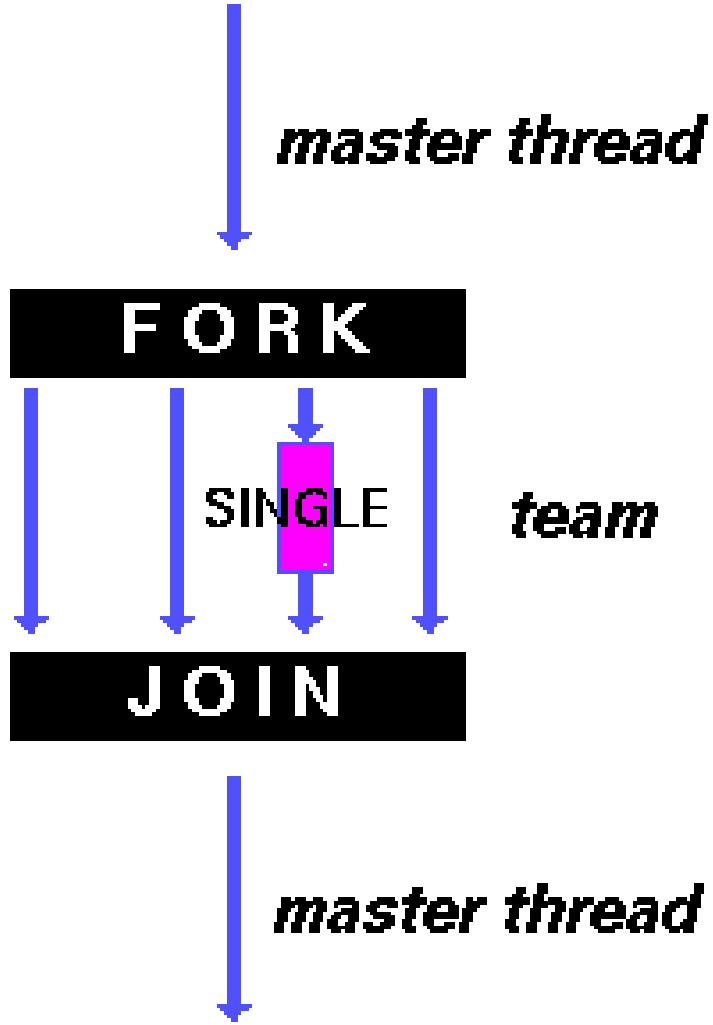
Work-Sharing Constructs

Sections
Breaks code into separate sections. each section is executed by a threads.
Functional parallelism



Work-Sharing Constructs

Single
Serializes a section of code
Useful when dealing with not thread-safe parts of code like I/O



OpenMP components

OpenMP consists of:

- Compiler directives
- Runtime Library Routines
- Environment Variable

Compiler Directives

Compiler directives are the main driver in OpenMP applications.
Used to give the compiler hints or commands.
Most commonly used directives:

- #pragma omp parallel (starts a parallel region)
- #pragma omp for (distributes iterations among threads)

Run-time Library Routines

Used for setting and querying information such as number of threads, thread ID, parallel region level, etc.

Most commonly used directives:

- OMP_SET_NUM_THREADS (sets number of threads used in next parallel section)
- OMP_GET_THREAD_NUM (gets the thread number of the current thread among the team)

Environment Variables

Used for setting information such as number of threads, thread affinity, etc.
One of the most commonly used directives:

- OMP_NUM_THREADS (sets number of threads used in next parallel section)

Compiling OpenMP code

- Using Intel compilers (icc, icpc, ifort), by adding ‘-openmp’
- Using GNU compilers (gcc, g++, g77, gfortran), by adding ‘-fopenmp’

Hello World!

Let's start parallelizing the common Hello World example.

- Code :

```
main () {
 printf("Hello World!\n");
}
```

- Output:

```
Hello World!
```

Hello World!

Now let's add omp parallel pragma to create a parallel region with 8 threads.

- Code :

```
main () {
#pragma omp parallel num_threads (8)
 {
  printf("Hello World!\n");
 }
}
```

- Output:

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Hello World!

Let's try to get more information from each thread.

- Code :

```
main () {
#pragma omp parallel num_threads(8)
 {
  int tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n",
tid);
 }
}
```

- Output:

```
Hello World from thread = 0
Hello World from thread = 7
Hello World from thread = 3
Hello World from thread = 6
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 4
Hello World from thread = 5
```

Hello World!

What if we wanted 1 thread to do something?

Code :

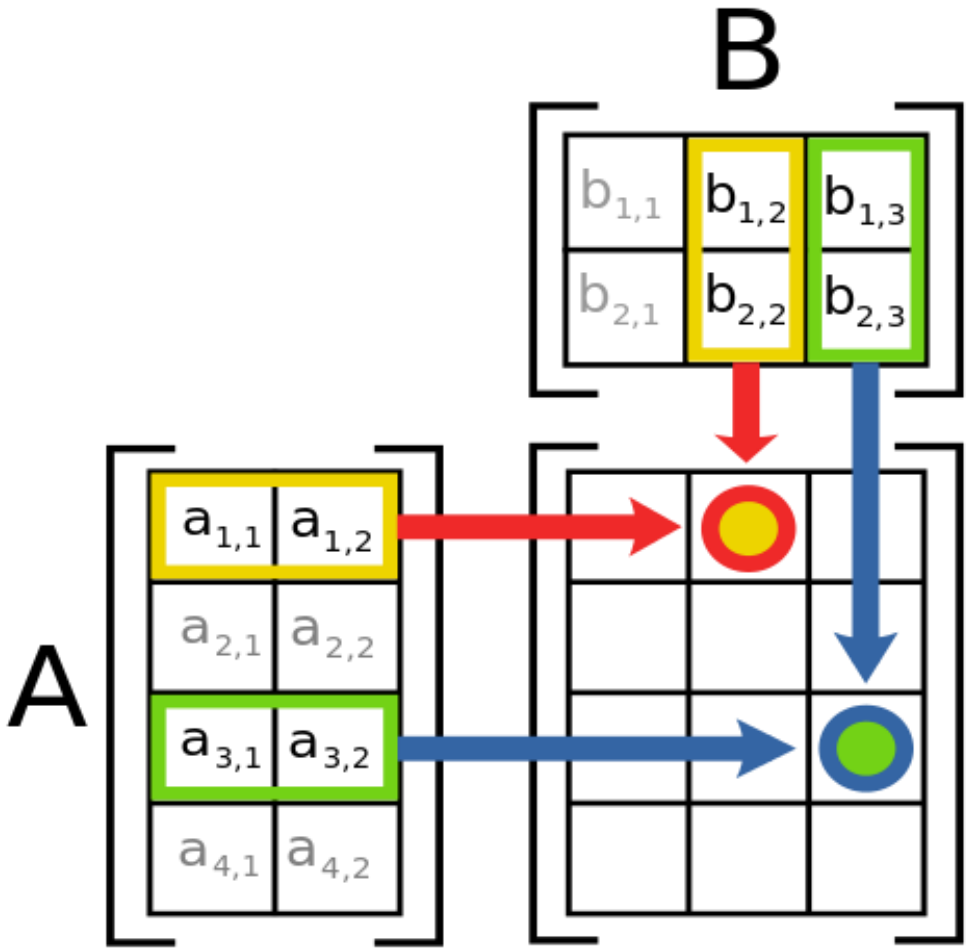
```
main () {
#pragma omp parallel num_threads(8)
 {
  int tid = omp_get_thread_num();
#pragma omp single
  {
   int nthreads = omp_get_num_threads();
   printf("Total # of threads = %d\n", nthreads);
  }
  printf("Hello World from thread = %d\n", tid);
 }
}
```

- Output:

```
Total # of threads = 8
Hello World from thread = 4
Hello World from thread = 0
Hello World from thread = 2
Hello World from thread = 6
Hello World from thread = 1
Hello World from thread = 7
Hello World from thread = 5
Hello World from thread = 3
```

Matrix Multiplication

- Sum of rows from first matrix multiplied by columns from second matrix.



Matrix Multiplication

- How to implement a parallel version of it?
- Let’s first start with a serial version.

Matrix Multiplication

Serial version of matrix multiplication.

- Code :

```
for (int c = 0; c < m; c++) {
 for (int d = 0; d < m; d++) {
  int sum = 0;
  for (int k = 0; k < m; k++) {
   sum = sum + FIRST[c][k]*SECOND[k][d];
  }
  MULTIPLY[c][d] = sum;
 }
}
```

- MULTIPLY = FIRST · SECOND
- FIRST, SECOND and MULTIPLY are all 2500x2500 matrices
- Runtime = 16.20 seconds

Matrix Multiplication

Serial version of matrix multiplication.

- Code :

```
for (int c = 0; c < m; c++) {
 for (int d = 0; d < m; d++) {
  int sum = 0;
  for (int k = 0; k < m; k++) {
   sum = sum + FIRST[c][k]*SECOND[k][d];
  }
  MULTIPLY[c][d] = sum;
 }
}
```

Which loop of the three to parallelize?
The red, the green, or the blue?

Matrix Multiplication

Parallel version of matrix multiplication.

- Code :

```
#pragma omp parallel
#pragma omp for
 for (int c = 0; c < m; c++) {
  for (int d = 0; d < m; d++) {
   int sum = 0;
   for (int k = 0; k < m; k++) {
    sum = sum + FIRST[c][k]*SECOND[k][d];
   }
   MULTIPLY[c][d] = sum;
  }
 }
```

- Let’s try to parallelize the red loop
- Runtime = 1.47 seconds
- Speedup = 11x

Matrix Multiplication

Parallel version of matrix multiplication.

- Code :

```
 for (int c = 0; c < m; c++) {
#pragma omp parallel
#pragma omp for
  for (int d = 0; d < m; d++) {
   int sum = 0;
   for (int k = 0; k < m; k++) {
    sum = sum + FIRST[c][k]*SECOND[k][d];
   }
   MULTIPLY[c][d] = sum;
  }
 }
```

- Let’s try to parallelize the green loop
- Runtime = 6.90 seconds
- Speedup = 2.34x
- Why did it perform worse than parallelizing the red loop?

Matrix Multiplication

Parallel version of matrix multiplication.

- Code :

```
 for (int c = 0; c < m; c++) {
  for (int d = 0; d < m; d++) {
   int sum = 0;
#pragma omp parallel
#pragma omp for
   for (int k = 0; k < m; k++) {
    sum = sum + FIRST[c][k]*SECOND[k][d];
   }
   MULTIPLY[c][d] = sum;
  }
 }
```

- Let’s try to parallelize the red loop
- This gives a wrong answer, why?!

Is it always that simple?

Data Dependences

Threads operate on shared address mode, they communicate by sharing variables.

Not all loops can be parallelized. You have to check data dependencies and unintended sharings before adding OpenMP directives.

Data Dependences

Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, …
Here is a serial implementation of Fibonacci sequence
Each iterations depend on last two iterations.
e.g. iteration 5 depends on depends on iterations 3 and 4.
Running the code 8 times, all produce the same correct result.

```
int main () {
 int long fibo[81];
 fibo[0] = 0;
 fibo[1] = 1;
 for (long i = 2; i < 81; i++) {
  fibo[i] = fibo[i-1] + fibo[i-2];
 }
 printf ("fibo[80] = %ld\n", fibo[80]);
}
```

```
Output:
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
fibo[80] = 23416728348467685
```

Data Dependences

- Adding omp pragmas before the for loop to parallelize it.
- What if 5 is calculated before 3 or 4?!
- Running the code 8 times, results in different numbers and all are incorrect!

```
#pragma omp parallel
#pragma omp for
 for (long i = 2; i < 81; i++) {
  fibo[i] = fibo[i-1] + fibo[i-2];
 }
 printf ("fibo[80] = %ld\n",
fibo[80]);
```

```
Output:
fibo[80] = 704407161128600
fibo[80] = 422264347243877
fibo[80] = 14300623258714
fibo[80] = 7713680008944624
fibo[80] = 7721580125263840
fibo[80] = 95113505610746 5416
fibo[80] = 876173885461
fibo[80] = 29072994390360
```

Race condition

Here we have an array of 10,000 ones.
When we add them serialy, we expect the result to be 10,000

```
main ()  {
 int ones[8192];
 for (int i=0; i<8192; i++)
 {
  ones[i] = 1;
 }
 int count = 0;
 for (int i=0; i<8192; i++)
 {
  count+=ones[i];
 }
 printf("Count = %d\n", count);
}
```

```
Output:
Count = 10000
```

Race condition

Parallelizing the summation loop and running the application 8 times.
We get wrong results. Why?

```
#pragma omp parallel
#pragma omp for
 for (int i=0; i<8192; i++)
 {
  count+=ones[i];
 }
 printf("Count = %d\n", count);
```

```
Output:
Count = 9375
Count = 10000
Count = 9375
Count = 9375
Count = 9375
Count = 8750
Count = 9375
Count = 10000
```

Race condition

How do we solve race condition?

- Consider serlializing this part of code.
- Using thread synchronization (will be covered later)
- Using reduction clause.
- Reduction clause creates a private copy of its variable list for each thread, and the final result is written into the global shared variable.

```
#pragma omp for reduction(+:count)
 for (int i=0; i<10000; i++)
 {
  count+=ones[i];
 }
```

```
Output:
Count = 10000
Count = 10000
Count = 10000
Count = 10000
Count = 10000
Count = 10000
Count = 10000
Count = 10000
Count = 10000
```

Thread Control Directives

Thread Control Directives

To handle previously mentioned data dependency problems, OpenMP provides a set of thread control and synchronization directives:

- barrier
- critical
- single
- master
- atomic
- ordered

Barrier

barrier creates a synchronization point to all working threads.

Critical

critical directive specifies a region of code that should be executed by only one thread at a time.
Might be of help solving *race condition* problem.

```
#pragma omp for
 for (int i=0; i<10000; i++)
 {
  #pragma omp critical
  count+=ones[i];
 }
```

Critical

critical directive has an optional name clause.
All critical directives sharing the same name are treated as the same region.
All unnamed critical directives are treated as the same region.
Conditional jumps (e.g. break, continue, return) are not allowed within a critical region.

Single

single directive lets only 1 worker thread execute the code block. All other members of the team skip this section but wait for the worker thread to synchronize with them after the single block, unless nowait.

Helpful when handling IO requests, or sections that are not thread safe generally.

Master

master directive lets only master threads execute that block of code. All other members if the team skip this section.

Atomic

atomic directive acts as mini-critical section
It only applies on a single following statement.
Not all statements can be preceded by atomic directive.
Atomic directive has much lower overhead than critical directive and different atomic directives cannot be grouped by name like critical directives.

Ordered

ordered directive forces later iterations to wait for previous iterations to complete before executing their ordered section of code

Ordered directive solves the problem of fibonacci calculation. *But do we get any enhancement from parallelization in this case?*

```
#pragma omp parallel
#pragma omp for ordered
 for (int i = 2; i < 81; i++) {
#pragma omp ordered
  fibo[i] = fibo[i-1] + fibo[i-2];
 }
```

General tips

- Never parallelize computationally trivial blocks of code.
- Pay special attention to private and shared variables.
- Don’t rely on debugging your code using breakpoints, it changes the behaviour of threads.
- Excessive needless use of critical sections or private variables hurt performance and memory efficieny.

Locks

Locks

Locks offer great control over threads
A thread can set/release locks over a code section
locked sections are not accessible to other threads
Used to implement *critical directive*

Lock routines

Lock routines operate on a variable of type omp_lock_t

Lock routines

void omp_init_lock(omp_lock_t *lock)
Initializes *lock and unlocks it
void omp_set_lock(omp_lock_t *lock)
If the lock is unavailable, this subroutine forces the thread to wait until the lock is available then it acquires it, otherwise it acquires it right away.
void omp_unset_lock(omp_lock_t *lock)
This subroutine releases the acquired lock.

Lock routines

int omp_test_lock(omp_lock_t *lock)
Attempts to acquire and set the lock, but does not block if the lock is unavailable.
void omp_destroy_lock(omp_lock_t *lock)
Disassociates the lock variable from any locks.

Nested lock routines

Nested locks work pretty much the same, except it allows multiple locks for code sections for the owning thread.

The lock variable stores a counter that is incremented with every omp_set_nest_lock() and decrement the counted with every omp_unset_nest_lock().

The lock is considered unlocked only if that counter is zero, in other words if it was unset exactly the same number of sets it encountered.

Locks vs. critical directive

Locks can be exited with conditional jumps (e.g. break, continue, return, exception handling)

All critical sections wait for each other, unless they were named during compilation time. Locks can be manipulated during runtime.

Unlike critical directive, using locks you may limit access to blocks of code to certain number of threads. How?!!

Locks offer greater power, but with great power comes great responsibility.

Locks vs. critical directive vs. atomic directive

Never use critical directive where you can use atomic directive!! (hurts performance)

Never use locks where you can use critical directive!! (If misused may break you code semantically and might hurt performance)

General tips

- Never parallelize computationally trivial blocks of code.
- Pay special attention to private and shared variables.
- Don’t rely on debugging your code using breakpoints, it changes the behaviour of threads.
- Excessive needless use of critical sections or private variables hurt performance and memory efficieny.