

# Concurrent and Distributed Systems

Talk Eight

OpenMP Technology

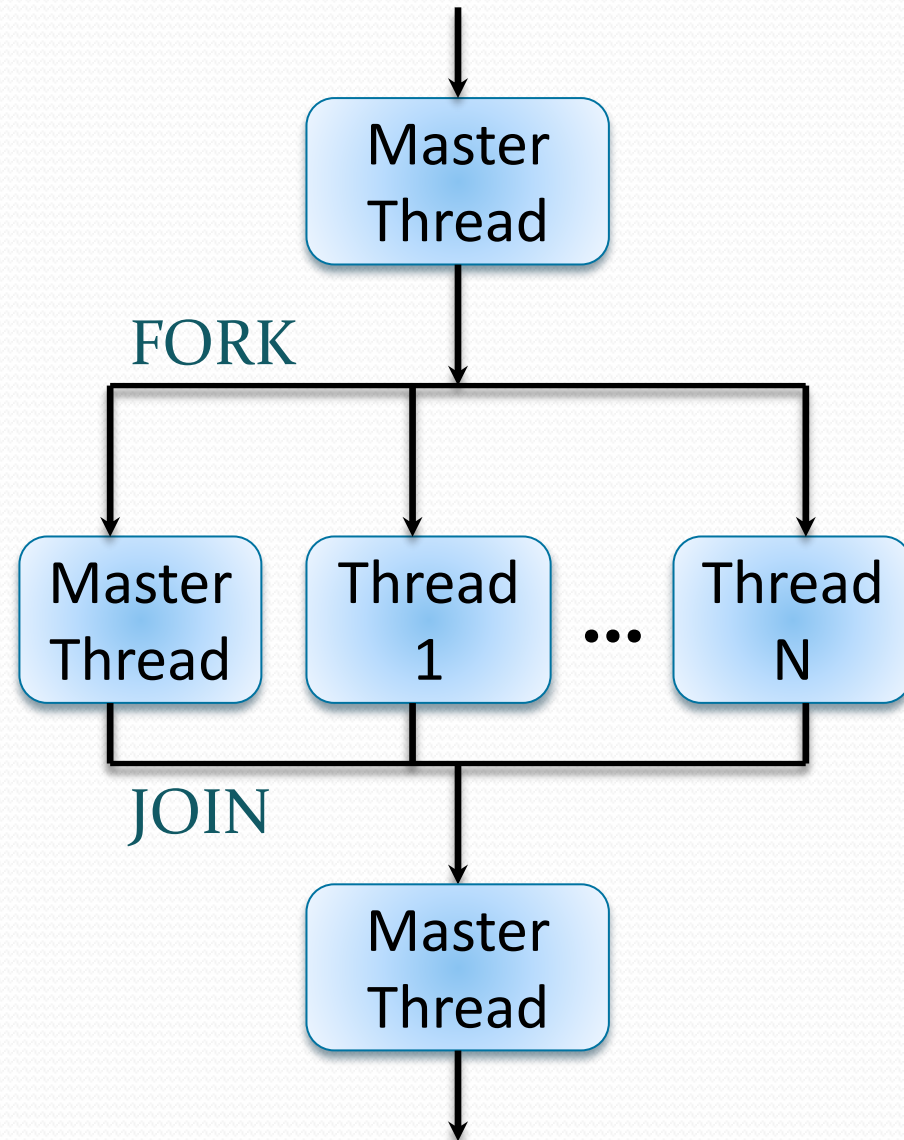
Maxim Mozgovoy

# OpenMP (Open Multi-Processing)

- OpenMP demonstrates another approach to introduce concurrency into conventional programs in C/C++ & Fortran.
- OpenMP adds special parallel instructions into these languages in form of special **compiler directives**.
- OpenMP is an **API** jointly developed and maintained by major hardware vendors, such as Intel, IBM, Fujitsu, Nvidia...
- The basic goal of OpenMP is **parallel computing** (speedup).
- Read more on this technology at <http://www.openmp.org>
- There is no official support for OpenMP in Java, but we will use an independent experimental implementation **omp4j**:  
<http://www.omp4j.org>

# OpenMP Paradigm

- OpenMP implements concurrent threads in a **shared memory environment**.
- OpenMP paradigm works best for the following scenario:
  - a master thread forks several worker threads
  - the system divides the work among them
  - the threads run concurrently
- The runtime environment allocates threads to different processors.



# Basic Example: *omp parallel* Directive (1)

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // sequential code  
  
        // omp parallel  
        {  
            // parallel code  
        }  
  
        // sequential code  
    }  
}
```

Inside *omp* regions  
two macros are available:

OMP4J\_NUM\_THREADS:  
Number of parallel threads

OMP4J\_THREAD\_NUM:  
Current thread ID

# Basic Example: *omp parallel* Directive (2)

```
public class HelloWorld {  
    public static  
    void main(String[] args) {  
        // omp parallel  
    {  
        System.out.println("Hello from thread " +  
                           OMP4J_THREAD_NUM + " of " +  
                           OMP4J_NUM_THREADS);  
    }  
}
```

Creates as many threads  
as there are cores on  
your machine

On a four-CPU machine prints:

Hello from thread 1 of 4  
Hello from thread 0 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4

Current thread ID

Total thread count

# Basic Example: *omp parallel* Directive (3)

- OpenMP is designed with **graceful degradation** in mind: if a certain compiler does not support it, OpenMP directives will be simply **ignored**.
- In compatible C++ environments OpenMP normally has to be enabled in compiler settings:
  - Use **-fopenmp** flag with GCC compiler.
  - Use **/openmp** flag with MSVC.
- The general syntax of OMP directives is:  
(C/C++)     **#pragma omp *directive-name* [*clauses*]**  
(omp4j)     **//omp *directive-name* [*clauses*]**

# Running “Hello, World” with omp4j

OpenMP is not build-in into Java, so its setup is more complex:

- Download [omp4j-1.2.jar](#) file from our Moodle.
- Compile your code as follows:  
`java -jar omp4j-1.2.jar YourProgram.java`
- Run it as usual:  
`java YourProgram`

# Primary OMP Directives (My Choice)

<b>critical</b>	Code will be executed on one thread at a time.
<b>parallel for</b>	The work in a for-loop inside a <b>parallel for</b> region will be divided among threads.
<b>single</b>	Only one thread will run a section of code.
<b>parallel</b>	Defines a region, where the code will be executed by multiple threads in parallel.
<b>sections</b>	Code sections will be divided among all threads.
<b>barrier</b>	Code will pause until all threads arrive at this point.
<b>master</b>	Only master thread will execute the specified region.

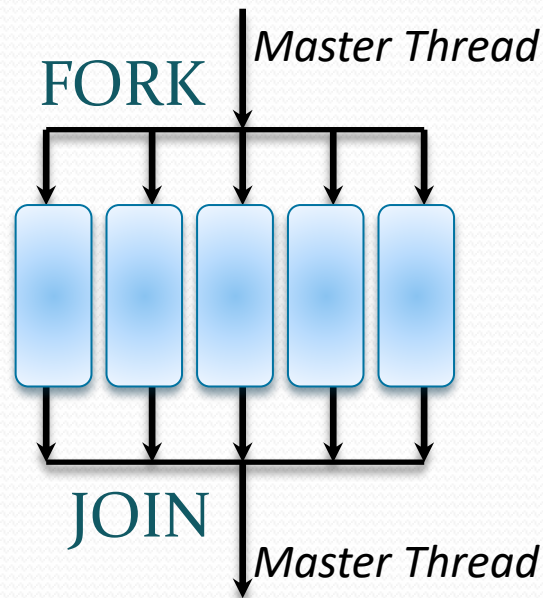
For a short summary of OpenMP directives, check

<http://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf>

# Simple For-loop Parallelization (1)

Let's review `parallel` directive operation.

When a thread faces `parallel` directive, it constructs a **team of threads** (including itself), and the operation continues in parallel till the end of the block.



The main thread is known here as a **master thread**. Typically the number of threads in a team is managed automatically.

# Simple For-loop Parallelization (2)

```
// omp parallel for
for(int i = 0; i < 5; ++i) {
    System.out.println("ID=" + OMP4J_THREAD_NUM +
        " i=" + i);
}
```

On my machine prints:

```
ID=0, i=0
ID=2, i=3
ID=3, i=4
ID=1, i=2
ID=0, i=1
```

// omp parallel for  
divides iterations between the threads  
in an **unpredictable** order.

The for-loop body must always be  
inside curly brackets.

Note that **parallel for** blocks cannot  
contain **break** statements.

# Simple For-loop Parallelization (3)

Note that a for-loop inside a **parallel** region is a **completely different** structure:

```
// omp parallel
{
    for(int i = 0; i < 4; ++i)
        System.out.println("ID=" + OMP4J_THREAD_NUM +
                           " i=" + i);
}
```

This fragment simply creates several parallel threads, each executing its own sequential (non-parallelized) for-loop printing values from 0 to 3.

# Passing Values (1)

OpenMP lets us control how variables are passed inside parallel sections (should a shared global variable be used or each process gets its own copy). By default, variables are shared, but we can change this behavior using **directive attributes**:

<code>private(a, b, c)</code>	variables <b>a</b> , <b>b</b> , <b>c</b> will be created for each thread using a <b>default</b> constructor
<code>firstprivate(a, b, c)</code>	variables <b>a</b> , <b>b</b> , <b>c</b> will be created for each thread using a <b>copy</b> constructor

**Note:** sharing mechanism works with **reference** (objects, arrays) types only, and shared variables should not be re-assigned inside parallel sections.

# Passing Values (2)

```
int global = 1;                // value (non-ref) type

// omp parallel for
for(int i = 0; i < 5; ++i) // will not be shared
    { global = 2; }
System.out.println(global); // prints 1
```

```
int[] global = {1};           // reference type

// omp parallel for
for(int i = 0; i < 5; ++i)    // shared among threads
    { global[0] = 2; }
System.out.println(global[0]); // prints 2
```

# Passing Values (3)

```
Integer global = new Integer(1); // ref type  
  
// omp parallel for  
for(int i = 0; i < 5; ++i)           // shared  
    { global = new Integer(2); }      // re-assigned  
System.out.println(global);           // prints 1
```

```
import java.util.concurrent.atomic.AtomicInteger;  
...  
AtomicInteger g = new AtomicInteger(0); // ref type  
  
// omp parallel for  
for(int i = 0; i < 5; ++i)           // shared among threads  
    { g.getAndIncrement(); }          // gets incremented  
System.out.println(g);                // prints 5
```

# Passing Values (4)

```
import java.util.concurrent.atomic.AtomicInteger;
...

AtomicInteger g = new AtomicInteger(10); // ref type

// omp parallel for private(g)
for(int i = 0; i < 5; ++i) { // shared among threads
    g.getAndIncrement();    // gets incremented
    System.out.println(g);  // prints 1 (*)
}
System.out.println(g);      // prints 10
```

(\*) Inside a loop a each thread gets its own **g** as follows:  
`g = new AtomicInteger();` // gets a zero value

To preserve the initial variable value,  
we need a **copy constructor** and a **firstprivate** attribute.

# Passing Values (5)

```
import java.util.concurrent.atomic.AtomicInteger;
...

class MyInteger extends AtomicInteger {
    public MyInteger(int val)      { super(val); }
    public MyInteger(MyInteger o) { super(o.get()); }
}

MyInteger g = new MyInteger(5); // ref type

// omp parallel for firstprivate(g)
for(int i = 0; i < 5; ++i) { // shared among threads
    g.getAndIncrement();      // gets incremented
    System.out.println(g);    // prints 6
}
System.out.println(g);       // prints 5
```

# Using Critical Sections (1)

Let's try parallelizing searching for the min value in an array:

```
int a[] = {7, 2, 5, 4, 1, 9};  
int min[] = {a[0]};  
  
// omp parallel for  
for(int i = 0; i < a.length; ++i)  
{  
    min[0] = Math.min(min[0], a[i]);  
}  
  
System.out.println("Result: " + min[0]);
```

# Using Critical Sections (2)

This code suffers from **race condition**, since **min** variable is not protected. Let's fix it with **critical** directive:

```
int a[] = {7, 2, 5, 4, 1, 9};
int min[] = {a[0]};

// omp parallel for
for(int i = 0; i < a.length; ++i)
{
    // omp critical
    { min[0] = Math.min(min[0], a[i]); }
}

System.out.println("Result: " + min[0]);
```

(Note that this code is very inefficient, though.)

# Using Section Directive (1)

**Section** directive allows to specify different jobs for different threads. Consider a modified version of sort-and-merge array sorting:

```
static final int N = 2000000;  
static int A[] = new int[N];  
  
static void Merge(int[] answer, int[] a, int[] b) {  
    /* merges two arrays into the resulting array */  
}
```

# Using Section Directive (2)

```
...
static public void main(String[] args) {
    // here fill the array
    // ...

    Arrays.sort(A, 0, N / 2);           // sort two halves
    Arrays.sort(A, N / 2, N);

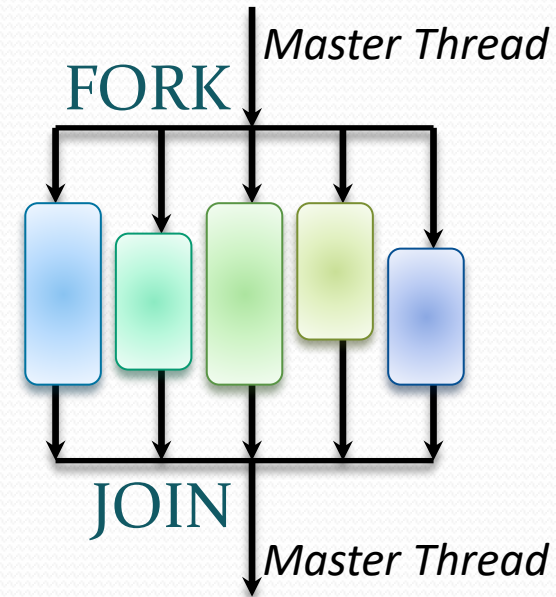
    int left[] = new int[N / 2];
    int right[] = new int[N - N / 2];
    System.arraycopy(A, 0, left, 0, N / 2);
    System.arraycopy(A, N / 2, right, 0, N - N / 2);

    Merge(A, left, right);              // then merge them
}
```

# Using Section Directive (3)

Let's parallelize the sorting part into two threads!

```
static public void main(String[] args) {  
    ...  
    // omp sections  
    {  
        // omp section  
        {  
            Arrays.sort(A, 0, N / 2);  
        }  
        // omp section  
        {  
            Arrays.sort(A, N / 2, N);  
        }  
    }  
    ...  
}
```



# Using Single and Barrier Directives (1)

Suppose we need to print a sum of all array elements.  
It can be done with this code:

```
import java.util.concurrent.atomic.AtomicInteger;
int a[] = { ... }; // source array
AtomicInteger sum = new AtomicInteger(0);

// omp parallel
{ // process a [left, right) array block
    int left = OMP4J_THREAD_NUM * a.length / OMP4J_NUM_THREADS;
    int right = (OMP4J_THREAD_NUM + 1) * a.length /
                OMP4J_NUM_THREADS;

    int partSum = 0;
    for(int i = left; i < right; ++i)
        partSum += a[i];

    sum.addAndGet(partSum);
}
System.out.println(sum.get());
```

# Using Single and Barrier Directives (2)

Structurally, this code contains a mix of parallel and non-parallel fragments. If we need to do more calculations and more reporting, it would look like this:

```
// omp parallel
{ // calculation
    ...
}
// report
...
```

```
// omp parallel
{ // calculation
    ...
}
// report
...
```

However, the same effect can be achieved inside **parallel** sections with **single** and **barrier** directives

# Using Single and Barrier Directives (3)

...

```
// omp parallel
```

```
{ // process a [left, right) array block
```

```
  int left = OMP4J_THREAD_NUM * a.length / OMP4J_NUM_THREADS;
```

```
  int right = (OMP4J_THREAD_NUM + 1) * a.length /  
                                                       OMP4J_NUM_THREADS;
```

```
  int partSum = 0;
```

```
  for(int i = left; i < right; ++i)  
    partSum += a[i];
```

```
  sum.addAndGet(partSum);
```

```
// omp barrier
```

```
{}
```

```
// omp single
```

```
{ System.out.println(sum.get()); }
```

```
}
```

Wait till all  
threads are here

Make sure only  
one thread  
prints the result

# Nested Parallel Sections

How will the following code work?

```
public static void main(String[] args) {  
    // omp parallel  
    {  
        // omp parallel  
        { System.out.println(OMP4J_THREAD_NUM); }  
    }  
}
```

Processing of parallel sections located inside other parallel sections in OpenMP depends on **nested parallelism** settings. Nested parallelism is always **enabled** in omp4j.

It means that on a 4-core machine `println()` will be called 16 times.

# Limiting Threads

If for some reason you need to set the number of threads created in a parallel section, use `threadNum(N)` attribute:

```
public static void main(String[] args) {  
    // omp parallel threadNum(16)  
    {  
        System.out.println(OMP4J_THREAD_NUM);  
    }  
}
```

Here `println()` will be called 16 times.

This tool is also useful for benchmarking: set the number of threads to one, measure performance of your code, then remove this limit and measure again to calculate speedup provided by OpenMP.

# Conclusions

- OpenMP aims to deliver a cross-platform language extension mechanism for painless high performance computing.
- A clear goal of OpenMP is fast parallel computing.
- The technology is mostly promoted and developed by the corresponding hardware vendors. Thus, adoption by language and platform varies.
- The best support is for C/C++ and Fortran by chip-provider tools. E.g., Intel C++ compiler supports OpenMP 4 (2013), while MSVC supports only OpenMP 2 (2000).