Su16 Guerilla Session 6

Parallel Programming

Flynn’s Taxonomy (Types of parallelism)

# Instructions/# Values	Single	Multiple
Single	Not parallel	SIMD - Perform same operation on many data of the same type. (ex: Intel Intrinsics)
Multiple	MISD - Perform multiple operations on same data (rarely seen in practice)	MIMD - Perform multiple operations on many data of same type

Amdahl’s Law

Relates the speedup of a particular section of code to the overall speedup of the whole task.

From wikipedia: the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

- *S*_latency is the theoretical speedup in latency of the execution of the whole task;
- *s* is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- *p* is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system *before the improvement*.

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

What is OpenMP?

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. - Wikipedia

In other words, OpenMP is an API which allows the programmer to utilize multiple processors in parallel (thread level parallelism) to speed up completion of a task.

OMP Directives

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

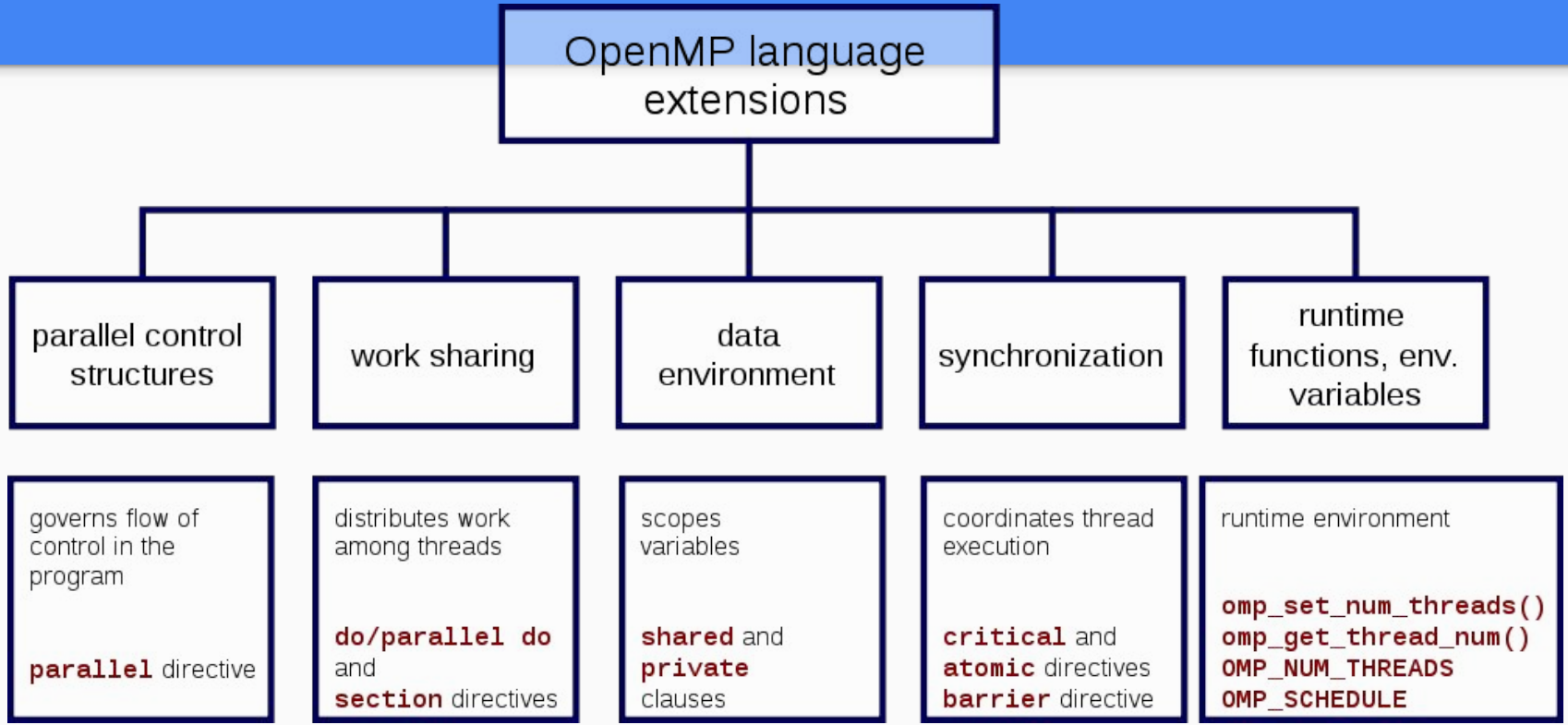
- By default, variables outside parallel sections are shared, to make private, need to declare with pragma: #pragma omp parallel private (x)
- #pragma omp parallel
- {

```
        #pragma omp for
        for(i=0;i<len;i++) { … }
}
```

<-->

```
#pragma omp parallel for
for(i=0;i<len;i++) { … }
```

A Great Summary From Lecture



OMP Usage Example

```
double avg, sum=0.0, A[MAX];

int i;

#pragma omp for reduction(+ : sum)

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

avg = sum/MAX;
```

What to Watch Out For

Data Races - Two threads access and write to same variable at the same time, so whichever thread finishes writing second overwrites the result of the first thread's write.

False Sharing - Two processors are accessing different variables in the same cache block. Every time one processor writes to the block, the second processor has to waste time reading from memory and updating its copy of the block in the cache even though the threads are writing to different variables.

Repeated work - Attempting to split work between threads but instead making each thread do all the work.

Spark RDD as described in Spark Programming Guide

Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Once created, the distributed dataset (`distData`) can be operated on in parallel. For example, we can call `distData.reduce(lambda a, b: a + b)` to add up the elements of the list. We describe operations on distributed datasets later on.

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, `map` is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, `reduce` is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel `reduceByKey` that returns a distributed dataset).

Spark Methods

rdd.map(func) - applies a single argument function to all elements of input RDD and outputs a new RDD.

rdd.flatMap(func) - applies a single argument function to all elements of input RDD and outputs a new RDD after flattening result

rdd.groupbykey() - Groups the values associated with the same key, and returns tuples of the form (key, value1, value2, … valueN)

rdd.reducebykey(func) - Groups the values associated with the same key, performs a function on all the values, and returns tuples of the form (key, func(val1, val2,..,valN))

rdd.reduce(func) - returns a single value which is the result of calling func on each element of the input RDD

rdd.count() - returns the number of elements in the input RDD

parallelize(collectionData, numParitions) - returns a RDD representation of your input collection