Synchronization

Reduction

Loop
Scheduling

Tasking

Conditional
Compilation

Conclusion

OpenMP Lecture 2

Vipin Vasu

Synchronization

Reduction

Loop
Scheduling

Tasking

Conditional
Compilation

Conclusion

Outline

1. Synchronization
2. Reduction
3. Loop Scheduling
4. Tasking
5. Conditional Compilation
6. Conclusion

Critical Regions

Critical regions can be marked by using the command

```
#pragma omp critical ( name )
{
    critical-section;
}
```

Atomicity

The atomic keyword in OpenMP specifies that the denoted action happens atomically. It is commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

```
#pragma omp atomic
sum += value;
```

Barrier

Barriers are used for execution synchronization. Upon reaching the barrier statement the thread waits for all other threads to reach that point. Once all threads have executed the barrier statement, further execution starts. There is an implicit barrier at the end of each parallel block, and at the end of each sections, for and single statement

```
first_set_of_lines;
#pragma omp barrier
second_set_of_lines;
```

nowait

It is used to avoid implicit barriers

```
#pragma omp parallel
{
#pragma omp for nowait
for(int n=0; n<10; ++n) Work();
SomeMoreCode();
}
```

Reduction

The reduction clause is a mix between the private, shared, and atomic clauses. It allows to accumulate a shared variable without the atomic clause, but the type of accumulation must be specified. It will often produce faster executing code than by using the atomic clause.

Factorial Function

```
int factorial(int number)
{
    int fac = 1;
    #pragma omp parallel for reduction(*:fac)
    for(int n=2; n<=number; ++n)
    fac *= n;
    return fac;
}
```

Reduction

To write the factorial function (shown above) without reduction, it probably would look like this:

```
int factorial(int number)
{
    int fac = 1;
    #pragma omp parallel for
    for(int n=2; n<=number; ++n)
    {
    #pragma omp atomic
    fac *= n;
    }
    return fac;
}
```

Loop Scheduling

The for construct splits the for-loop so that each thread in the current team handles a different portion of the loop.

```
#pragma omp parallel
{
    #pragma omp for
    for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

Static Scheduling

It divides the loop into contiguous chunks of (roughly) equal size. Each thread then executes on exactly one chunk. Upon entering the loop, each thread independently decides which chunk of the loop they will process.

```
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Dynamic Scheduling

Dynamic scheduling assigns a chunk of work, whose size is defined by the chunksize, to the next thread that has finished its current chunk. This allows for a very flexible distribution which is usually not reproduced from run to run.

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Guided Scheduling

Again, threads request new chunks dynamically, but the chunksize is always proportional to the remaining number of iterations divided by the number of threads.

```
#pragma omp for schedule(guided)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

Runtime Scheduling

For debugging and profiling purposes, OpenMP provides a facility to determine the loop scheduling at runtime. If the scheduling clause specifies "RUNTIME," the loop is scheduled according to the contents of the OMP_SCHEDULE shell variable. However, there is no way to set different schedulings for different loops that use the SCHEDULE(RUNTIME) clause.

Tasking

In early versions of the standard, parallel worksharing in OpenMP was mainly concerned with loop structures. However, not all parallel work comes in loops; a typical example is a linear list of objects , which should be processed in parallel. Since a list is not easily addressable by an integer index or a random-access iterator, a loop worksharing construct is ruled out, or could only be used with considerable programming effort. OpenMP 3.0 provides the task concept to circumvent this limitation. A task is defined by the TASK directive, and contains code to be executed. 1 When a thread encounters a task construct, it may execute it right away or set up the appropriate data environment and defer its execution. The task is then ready to be executed later by any thread of the team.

Tasking

```
struct node { node *left, *right; };
extern void process(node* );
void traverse(node* p)
{
    if (p->left)
    #pragma omp task
    traverse(p->left);
    if (p->right)
        #pragma omp task
        traverse(p->right);
    process(p);
}
```

Conditional Compilation

In some cases it may be useful to write different code depending on OpenMP being enabled or not. The directives themselves are no problem here because they will be ignored gracefully. Beyond this default behavior one may want to mask out, e.g., calls to API functions or any code that makes no sense without OpenMP enabled. This is supported in C/C++ by the preprocessor symbol _OPENMP, which is defined only if OpenMP is available.

The End