Programmierung paralleler Rechnerarchitekturen, Winter 2012/13

Exercise 2

Task 2.1 (OpenMP theoretical concepts)**.** Recall what you have learned in the classes.

a) Indicate and describe the types of scheduling that OpenMP provides. Indicate the situations for which they are indicated.

Solution: **Static:** Iterations divided into #threads blocks, which are assigned to threads. If chunk is defined, chunks are assigned to threads in a round robin fashion. **Dynamic:** Iterations divided in blocks of chunksize (1 if not defined), assigned to threads in the order they finish the previous blocks. **Guided:** Similar to dynamic but the side of the blocks reduces over time.

b) What are the two main disadvantages of dynamic scheduling?

Solution: **Locality issues:** work is not assigned contiguously, which might impair locality usage. **Overhead:** Assignment of work is costly (need to verify what thread is idle / thread has to signal scheduler, etc), in comparison to static scheduling.

c) What forms of synchronization are provided by OpenMP? Specify and fully describe each primitive/construct/clause.

Solution: **Critical:** All threads execute that piece of code, but one at a time. **Single:** Piece of code executed only by one (unspecified) thread only. **Atomic:** Single memory location executed by all threads but one at a time. **Master:** Piece of code executed by the master thread. **Barrier:** Threads wait at the barrier point until all threads reach it. **Nowait:** Threads do not wait in the end of a for construct, where an implicit barrier is present.

Task 2.2. Complete the scoping of the variables in the code using their types accordingly.

```
#pragma omp parallel for
for (i=0; i<m; i++){
      y[i]=0;
      for (j=0; j<n; j++){
           y[i]= y[i] + A[i][j]*x[j];
      }
 }
```

Solution:

```
#pragma omp parallel for private(i,j) shared(y,m,n,A,x)
for (i=0; i<m; i++){
      y[i]=0;
      for (j=0; j<n; j++){
           y[i]= y[i] + A[i][j]*x[j];
      }
 }
```

Task 2.3. Eliminate any data race in the code. *fib(i)* is a function that outputs the i^{th} element in the Fibonacci sequence.

```
int y[55]={-1};
#pragma omp parallel for
for (i=0; i<10; i++){
    y[fib(i)]=fib(i)+i;
}
```

Solution: Exercise with multiple solutions. A possible one would be:

```
int y[55]={-1};

y[0] = 0;
y[1] = fib(1)+2;

#pragma omp parallel for
for (i=3; i<10; i++){
    y[fib(i)]=fib(i)+i;
}
```

Task 2.4. Indicate the type of dependence (if any) in each piece of code and parallelize them but removing the dependence accordingly.

a)
```
#pragma omp parallel for
for (i=0; i<n-1; i++){
    y[i]=y[i+1]*x[i+1];
}
```

Solution: Anti-dependence. Solvable with array duplication.

```
/* create a copy of vector y, named aux_y */

#pragma omp parallel for
for (i=0; i<n-1; i++){
    y[i]=aux_y[i+1]*x[i+1];
}
```

b)
```
#pragma omp parallel for
for (i=0; i<n; i++){
            y[i] = h[f(1)] * y[i];
            d = y[i];
}
printf("Result = %d.\n",d);
```

Solution: Output dependence. Solvable with *lastprivate*. Writing y[x] is an acceptable data-race.

```
#pragma omp parallel for lastprivate(d)
for (i=0; i<n; i++){
```

```
                y[i] = h[f(1)] * y[i];
                d = y[i];
    }
    printf("Result = %d.\n",d);
```

c)
```
    #pragma omp parallel for
    for (i=4; i<n-2; i+=2){
        a[i+1] = x;
        a[i+2] = y;
        h[i] = a[i];
        h[i+1] = a[i];
    }
```

Solution: Flow dependence on a[i+2]. There are no problems with accesses to h. A possible solution, with loop skewing (double loop for commodity - may affect performance if compiler is not able to join them):

```
#pragma omp parallel for
for (i=4; i<n-2; i+=2){
    a[i+1] = x;
    a[i+2] = y;
}

#pragma omp parallel for
for (i=4; i<n-3; i+=2){
    h[i+1] = a[i+1];
    h[i+2] = a[i+1];
}
```

Discussion of this exercise on 12^{th} November 2013.