UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 3

Solution

Lecture Parallel Processing
Winter Term 2024/25

Exercise 1: Dependence analysis

a) Correct: S1 writes `a[7]` in iteration `7`, S2 reads `a[7]` in all following iterations.
b) Correct: S1 reads `b[i]`, S2 writes `b[i]`.
c) Incorrect: as in each iteration, S2 is executed **after** S1, there can never be a dependence within a single loop iteration from S2 to S1.
d) Correct: e.g. S2 writes `b[4]` in iteration 4, S1 reads `b[4]` in iteration 5.
e) Correct: S1 writes `a[7]` in iteration `7`, S2 reads `a[7]` in all previous iterations.
f) Incorrect: S1 writes only `a`, which is not read by S3. S3 writes only `c` which is not read by S1.
g) Correct: e.g. S2 reads `c[3]` in iteration 3, S3 writes to `c[3]` in iteration 4.
h) Incorrect: S2 and S3 write to different arrays.
i) Incorrect: S3 writes to `c[i-1]` and S2 reads from `c[i]`. Thus, it we e.g. look at `c[5]`, it is written in iteration 6, but is read in iteration 5, i.e., earlier. Thus, this is an **anti**-dependence.
j) Correct: each statement writes to a different array.

Exercise 2: Loop parallelization

Here are the correct parallelizations:

```
void loop1()
{
    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        a[i] = b[i] + c[0];
        b[i] = a[i] - c[i];
    }
}

void loop2()
{
    // True dependence: a[i] (write, i=1) --> a[i-1] (read, i=2)
    for (int i=1; i<N; i++) {
        a[i] = a[i-1];
        b[i] = a[i] + c[i];
    }
}

void loop3()
{
    // Anti dependence: a[i+2] (read, i=1) --> a[i] (write, i=3)
    // So use renaming. Here, the variable 'a' is renamed as 'aa' when we store into
    // it. Later, we copy 'aa' into 'a' again.
    double aa[N];
    #pragma omp parallel
    {
```

```c
        #pragma omp for
        for (int i=1; i<N-2; i++) {
            aa[i] = b[i] + a[i+2];
            c[i] = b[i-1];
        }
        #pragma omp for
        for (int i=1; i<N-2; i++)
            a[i] = aa[i];
    }
}

void loop4()
{
    // True dependence (among others): a[i] (write, i=N/2) --> a[N/2] (read, i=N/2+1)
    for (int i=0; i<N; i++) {
        a[i] = a[i] - 0.9 * a[N/2];
    }
}

void loop5()
{
    // True dependence: a[i+N/3] (write, i=0) --> a[i] (read, i=N/3)
    for (int i=0; i<N/2; i++) {
        a[i+N/3] = (c[i] - a[i])/2;
    }
}

void loop6()
{
    // No dependence, since i >= N/3 is not possible!
    #pragma omp parallel for
    for (int i=0; i<N/3; i++) {
        a[i+N/3] = (c[i] - a[i])/2;
    }
}

void loop7()
{
    // Dependeces cannot be analyzed at compile time.
    // Whether or not we have dependences depends on the contents of map[].
    for (int i=0; i<N; i++) {
        a[map[i]] = a[i] + b[i];
    }
}

void loop8()
{
    // The outer loop cannot be parallelized, since it carries dependences.
    // In the inner loop, however, there are no dependences (for a fixed i)!
    for (int i=1; i<M-1; i++) {
        #pragma omp parallel for
        for (int j=1; j<M-1; j++) {
            m[i][j] = (m[i-1][j-1] + m[i-1][j+1] + m[i+1][j-1] + m[i+1][j+1]) / 4;
        }
    }
}
```

Exercise 3: Parallelization of a simple optimization code with OpenMP (Compulsory Exercise! Submit until Tuesday, November 26$^{\text{th}}$, 10:00 via moodle)

Exercise 4: Parallelization of a numerical integration using OpenMP (Compulsory Exercise! Submit until Tuesday, November 26$^{\text{th}}$, 10:00 via moodle)