Sheet 07

# PS Parallel Programming

Patrick Wintner

May 5, 2025

## 1 Parallelizing Loops

The dependencies and parallelisation posibilities of code snippets are analysized.

### 1.1

#### 1.1.1 Serial

```
for (int i=0; i < n-1; i++) {
        x[i] = (y[i] + x[i+1]) / 7; // S
}
```

Statement S anti-depends (Write-After-Read) on itself: $S\delta^{-1}S$. Anti-dependencies can be eliminated through variable renaming.

#### 1.1.2 Parallel

```
#pragma omp parallel for schedule(static)
for (int i=0; i < n-1; i++) {
        x2[i] = (y[i] + x[i+1]) / 7;
}
```

## 1.2

### 1.2.1 Serial

```
for (int i=0; i < n; i++) {
        a = (x[i] + y[i]) / (i+1); // S1
        z[i] = a; // S2
}

f = sqrt(a + k); // S3
```

Statement S2 truly depends (Read-After-Write) on S1 and S3 truly depends on the last instance of S1: $S1\delta S2, S2\delta S3$. The depency is obviously not loop-carried, therefore the loop can be parallelized by making 'a' private within the loop.

### 1.2.2 Parallel

```
#pragma omp parallel for lastprivate(a)
for (int i=0; i < n; i++) {
        a = (x[i] + y[i]) / (i+1);
        z[i] = a;
}

f = sqrt(a + k);
```

## 1.3

### 1.3.1 Serial

```
for (int i=0; i < n; i++) {
        x[i] = y[i] * 2 + b * i; // S1
}

for (int i=0; i < n; i++) {
        y[i] = x[i] + a / (i+1); // S2
}
```

Statement S2 both truly and anti-depends on S1: $S1\delta S2, S1\delta^{-1}S2$. There is no dependence within the loops, therefore the loops themselves can be parallelized.

### 1.3.2 Parallel

```
#pragma omp parallel for schedule(static)
for (int i=0; i < n; i++) {
        x[i] = y[i] * 2 + b * i;
}

#pragma omp parallel for schedule(static)
for (int i=0; i < n; i++) {
        y[i] = x[i] + a / (i+1);
}
```

## 2 Parallelizing more Loops

The dependencies of code snippets are analysized and the code snippets themselves are parallelized. The wall time of both the serial and parallel versions is measured.

### 2.1 Measurement Method

All measurements were done on the LCC3 cluster by calling `sbatch job.sh <executable> 3`, e. g. `sbatch job.sh a_ser 3` (3 is the number of measurements) with the number of loop iterations set to 100000000.

The following scripts are involved in the experiment.

#### 2.1.1 SLURM Script

```
#!/bin/bash
# usage: sbatch [slurm_options] <executable> <number_of_measurements>

# Execute job in the partition "lva" unless you have special requirements.
#SBATCH --partition=lva
# Name your job to be able to identify it later
```

```
#SBATCH --job-name csba4017
# Redirect output stream to this file
#SBATCH --output=output.log
# Maximum number of tasks (=processes) to start in total
#SBATCH --ntasks=1
# Maximum number of tasks (=processes) to start per node
#SBATCH --ntasks-per-node=1
# Enforce exclusive node allocation, do not share with other jobs
#SBATCH --exclusive

./benchmark.sh $1 $2
```

### 2.1.2 Benchmark Script

```
#!/bin/bash
# Usage: ./benchmark.sh <executable> <number_of_measurements>

ITER=100000000  # number of loop iterations
results=$1".dat"
echo "x y ey" > $results # create header
make toTable
make $1
for i in {1,4,8,12} # number of threads
do
        measurements=$i"_"$1".log"
        export OMP_NUM_THREADS=$i
        for j in $(seq 1 $2) # repeat measurement £2 times
        do
                ./$1 $ITER >> $measurements # store measurement results in <executable>.
        done
        ./toTable $measurements $results $2 $i #store table in <executable>.dat
        rm $measurements
done
make clean
```

## 2.2

### 2.2.1 Serial

```
#ifndef A_SER_H
#define A_SER_H

double factor = 1; // S1

for (int i=0; i < n; i++) {
        x[i] = factor * y[i]; // S2
        factor = factor / 2; // S3
}
#endif
```

The zero-th instance of both statements S2 and S3 truly depends on S1: $S1\delta S2, S1\delta S3$. Furthermore, S2 has true loop-carried dependence on S3 and S3 has a true loop-carried dependence on itself: $S3\delta S2, S3\delta S3, S3\delta^{-1}S3$.

### 2.2.2 Parallel

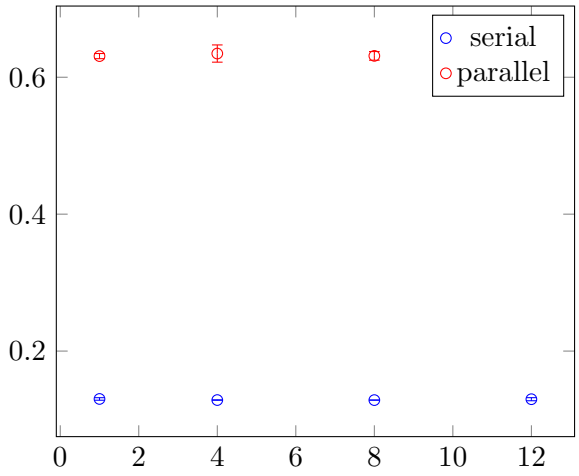
```
#ifndef A_PAR_H
#define A_PAR_H

double *factor = malloc(n*sizeof(*factor));
if(!factor) {
        free(x);
        free(y);
        free(z);
        perror("malloc");
        return EXIT_FAILURE;
}
factor[0] = 1;
for(int i=0; i < n-1; i++) {
        factor[i+1] = factor[i]/2;
}
#pragma omp parallel for schedule(static)
for (int i=0; i < n; i++) {
        x[i] = factor[i] * y[i];
}
```

```
20 free(factor);
21 #endif
```

It is used that the different instances of statement S2 do not depend on each other.

### 2.2.3 Experiment Results




## 2.3

### 2.3.1 Serial

```
1 #ifndef B_SER_H
2 #define B_SER_H
3
4 for (int i=1; i<n; i++) {
5         x[i] =(x[i] + y[i-1]) / 2;
6         y[i] = y[i] +z[i] * 3;
7 }
8 #endif
```

### 2.3.2 Parallel

```
1 #ifndef B_PAR_H
2 #define B_PAR_H
```

```
3
4  for (int i=1; i<n; i++) {
5          x[i] =(x[i] + y[i-1]) / 2;
6          y[i] = y[i] +z[i] * 3;
7  }
8  #endif
```

### 2.3.3 Experiment Results

## 2.4

### 2.4.1 Serial

```
1  #ifndef B_SER_H
2  #define B_SER_H
3
4  x[0] = x[0] + 5 * y[0];
5  for (int i = 1; i<n; i++) {
6          x[i] = x[i] + 5 * y[i];
7          if ( twice ) {
8                  x[i-1] = 2 * x[i-1];
9          }
10 }
11 #endif
```

### 2.4.2 Parallel

```
1  #ifndef C_PAR_H
2  #define C_PAR_H
3
4  x[0] = x[0] + 5 * y[0];
5  for (int i = 1; i<n; i++) {
6          x[i] = x[i] + 5 * y[i];
7          if ( twice ) {
8                  x[i-1] = 2 * x[i-1];
9          }
10 }
11 #endif
```

### 2.4.3 Experiment Results

# 3 ... And One More Loop to Parallelize

## 3.1 Serial

```
for (int i = 0; i < 4; ++i) {
        for (int j = 1; j < 4; ++j) {
                a[i + 2][j - 1] = b * a[i][j] + 4;
        }
}
```

### 3.1.1 Distance and Direction Vectors

| i | j | S: a[i+2][j-1] | S: a[i][j] |
|---|---|---|---|
| 0 | 1 | a[2][0]= | =a[0][1] |
| 0 | 2 | **a[2][1]=** | =a[0][2] |
| 0 | 3 | **a[2][2]=** | =a[0][3] |
| 1 | 1 | a[3][0]= | =a[1][1] |
| 1 | 2 | **a[3][1]=** | =a[1][2] |
| 1 | 3 | **a[3][2]=** | =a[1][3] |
| 2 | 1 | a[4][0]= | **=a[2][1]** |
| 2 | 2 | a[4][1]= | **=a[2][2]** |
| 2 | 3 | a[4][2]= | =a[2][3] |
| 3 | 1 | a[5][0]= | **=a[3][1]** |
| 3 | 2 | a[5][1]= | **=a[3][2]** |
| 3 | 3 | a[5][2]= | =a[3][3] |

| dependence relation | array element | distance vector | direction vector |
|---|---|---|---|
| $S[0][2]\delta S[2][1]$ | a[2][1] | $(2,-1)$ | $(<,>)$ |
| $S[0][3]\delta S[2][2]$ | a[2][2] | $(2,-1)$ | $(<,>)$ |
| $S[1][2]\delta S[3][1]$ | a[3][1] | $(2,-1)$ | $(<,>)$ |
| $S[2][3]\delta S[3][2]$ | a[3][2] | $(2,-1)$ | $(<,>)$ |

There are four loop-carried true data dependencies in the code snippet.

### 3.2 Parallel

```
for (int i = 0; i < 4; ++i) {
        for (int j = 1; j < 4; ++j) {
                a[i + 2][j - 1] = b * a[i][j] + 4;
        }
}
```