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

```
1  for (int i=0; i < n-1; i++) {
2          x[i] = (y[i] + x[i+1]) / 7; // S
3  }
```

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

```
1  #pragma omp parallel for schedule(static)
2  for (int i=0; i < n-1; i++) {
3          x2[i] = (y[i] + x[i+1]) / 7;
4  }
```

1.2

1.2.1 Serial

```
1  for (int i=0; i < n; i++) {
2          a = (x[i] + y[i]) / (i+1); // S1
3          z[i] = a; // S2
4  }
5
6  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

```
1  #pragma omp parallel for lastprivate(a)
2  for (int i=0; i < n; i++) {
3          a = (x[i] + y[i]) / (i+1);
4          z[i] = a;
5  }
6
7  f = sqrt(a + k);
```

1.3

1.3.1 Serial

```
1  for (int i=0; i < n; i++) {
2          x[i] = y[i] * 2 + b * i; // S1
3  }
4
5  for (int i=0; i < n; i++) {
6          y[i] = x[i] + a / (i+1); // S2
7  }
```

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

```
1  #pragma omp parallel for schedule(static)
2  for (int i=0; i < n; i++) {
3          x[i] = y[i] * 2 + b * i;
4  }
5
6  #pragma omp parallel for schedule(static)
7  for (int i=0; i < n; i++) {
8          y[i] = x[i] + a / (i+1);
9  }
```

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

```
1  #!/bin/bash
2  # usage: sbatch [slurm_options] <executable> <number_of_measurements>
3
4  # Execute job in the partition "lva" unless you have special requirements.
5  #SBATCH --partition=lva
6  # Name your job to be able to identify it later
```

```
7   #SBATCH --job-name csba4017
8   # Redirect output stream to this file
9   #SBATCH --output=output.log
10  # Maximum number of tasks (=processes) to start in total
11  #SBATCH --ntasks=1
12  # Maximum number of tasks (=processes) to start per node
13  #SBATCH --ntasks-per-node=1
14  # Enforce exclusive node allocation, do not share with other jobs
15  #SBATCH --exclusive
16
17  ./benchmark.sh $1 $2
```

2.1.2 Benchmark Script

```
1   #!/bin/bash
2   # Usage: ./benchmark.sh <executable> <number_of_measurements>
3
4   ITER=100000000  # number of loop iterations
5   results=$1".dat"
6   echo "x y ey" > $results # create header
7   make toTable
8   make $1
9   for i in {1,4,8,12} # number of threads
10  do
11          measurements=$i"_"$1".log"
12          export OMP_NUM_THREADS=$i
13          for j in $(seq 1 $2) # repeat measurement £2 times
14          do
15                  ./$1 $ITER >> $measurements # store measurement results in <executable>.
16          done
17          ./toTable $measurements $results $2 $i #store table in <executable>.dat
18          rm $measurements
19  done
20  make clean
```

2.2

2.2.1 Serial

```
1  #ifndef A_SER_H
2  #define A_SER_H
3
4  double factor = 1; // S1
5
6  for (int i=0; i < n; i++) {
7          x[i] = factor * y[i]; // S2
8          factor = factor / 2; // S3
9  }
10 #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

```
1  #ifndef A_PAR_H
2  #define A_PAR_H
3
4  #pragma omp parallel for schedule(static)
5  for (int i=0; i < n; i++) {
6          x[i] = 1.0/pow(2,i) * y[i];
7  }
8  #endif
```

It is used that the different instances of statement S2 do not depend on each other and
that there exist a recurrence formula to determine 'factor'.

2.2.3 Experiment Results
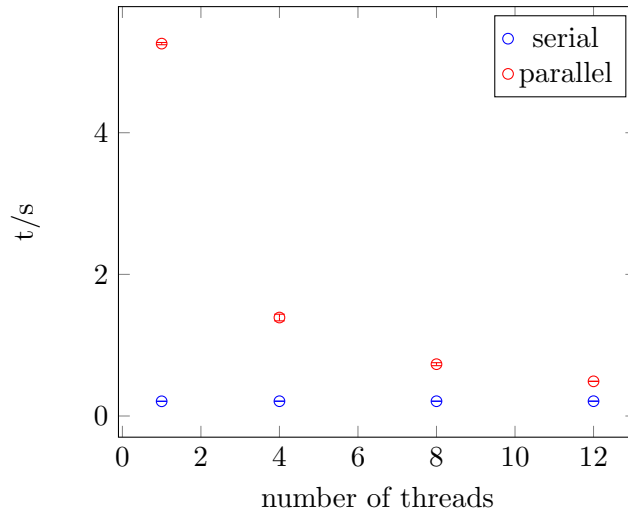
The walltime of the serial implementation is not affected by the number of threads, while the walltime of the parallel implementation decreases exponentially. Therefore both implementations behave as expected.

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; // S1
6          y[i] = y[i] +z[i] * 3; // S2
7  }
8  #endif
```

Statement S1 truly depends on S2: $S2 \delta S1$. This dependency is loop-carried. This snippet can be parallelized by having two sequentially executed loops. The loops themselves are executed in parallel.

2.3.2 Parallel

```
1   #ifndef B_PAR_H
2   #define B_PAR_H
3
4   #pragma omp parallel for schedule(static)
5   for (int i=1; i<n; i++) {
6           y[i] = y[i] +z[i] * 3;
7   }
8
9   #pragma omp parallel for schedule(static)
10  for (int i=1; i<n; i++) {
11          x[i] = (x[i] + y[i-1]) / 2;
12  }
13  #endif
```

2.3.3 Experiment Results
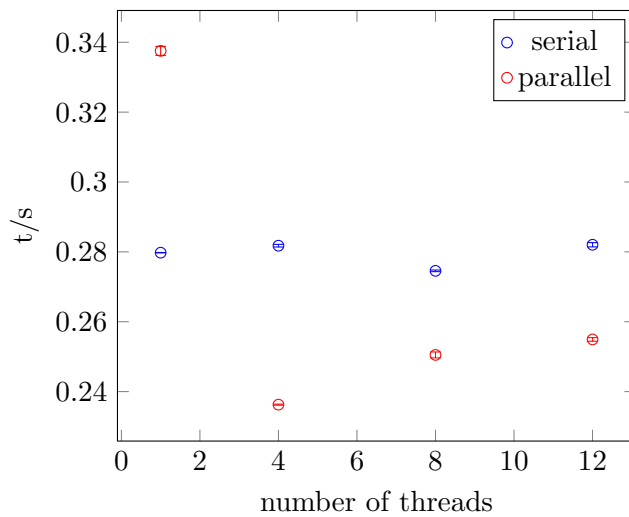
The walltime of the serial implementation is barely affected by the number of threads (as expected), while the walltime of the parallel implementation is minimal when the number of threads is four (and not twelve), which is surprising.

7

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]; // S1
5  for (int i = 1; i<n; i++) {
6         x[i] = x[i] + 5 * y[i]; // S2
7         if ( twice /* S3 */) {
8                 x[i-1] = 2 * x[i-1]; // S4
9         }
10 }
11 #endif
```

Statement S4 has a control dependency on S3 and a data dependency on S2. The first instance of statement S4 has a data dependency on S1. This code snippet can be parallelized by replacing the loop by two sequential loops so that all statements within a loop are independent from each other. The control dependency is retained by wrapping the second loop in an if-instruction.

2.4.2 Parallel

```
1  #ifndef C_PAR_H
2  #define C_PAR_H
3
4  x[0] = x[0] + 5 * y[0];
5  #pragma omp parallel for schedule(static)
6  for (int i = 1; i<n; i++) {
7         x[i] = x[i] + 5 * y[i];
8  }
9
10 if (twice ) {
11        #pragma omp parallel for schedule(static)
12        for (int i = 1; i<n; i++) {
13                x[i-1] = 2 * x[i-1];
14        }
15 }
16 #endif
```

2.4.3 Experiment Results
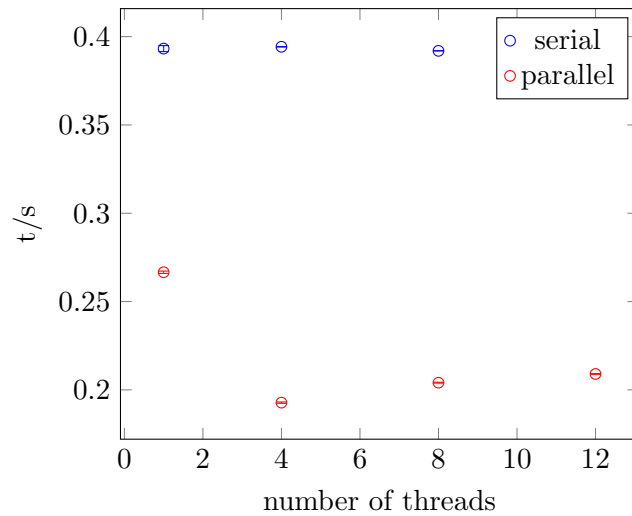
The walltime of the serial implementation is not affected by the number of threads (as expected), while the walltime of the parallel implementation is minimal when the number of threads is four (and not twelve), which is surprising.

3 ... And One More Loop to Parallelize

3.1 Serial

```
1  for (int i = 0; i < 4; ++i) {
2          for (int j = 1; j < 4; ++j) {
3                  a[i + 2][j - 1] = b * a[i][j] + 4;
4          }
5  }
```

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]	$(2, -1)$	$(<, >)$
$S[1][3]\delta S[3][2]$	a[3][2]	$(2, -1)$	$(<, >)$

There are four loop-carried true data dependencies in the code snippet. All write accesses happen before all read accesses. Therefore the loop can be divided into two loops, so that both loops themselves can be executed in parallel. This solution only works because the number of iterations is low enough. A solution that scales is to only parallelize the inner loop.

3.2 Parallel

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < 2; ++i) {
        for (int j = 1; j < 4; ++j) {
                a[i + 2][j - 1] = b * a[i][j] + 4;
        }
}


#pragma omp parallel for schedule(static)
for(int i = 2; i < 4; ++i) {
        for (int j = 1; j < 4; ++j) {
                a[i+2][j-1] = b * a[i][j] + 4;
```

```
12          }
13     }
```