Sheet 08

PS Parallel Programming

Patrick Wintner

May 12, 2025

1 Compiler Dependence Analysis

The dependence analysis of compilers is examined.

1.1 Source Code

```c
#include <stdio.h>

#define SIZE 1024

int main(int argc, char** argv) {

        int a[SIZE];
        int b[SIZE];

        for(int i = 0; i < SIZE; ++i) {
                a[i] = argc;
        }

        for(int i = 0; i < SIZE; ++i) {
                b[i] = a[i];
        }

        for(int i = 4; i < SIZE; ++i) {
                a[i-4] = a[i];
        }
```

```
21
22          for(int i = 1; i < SIZE-1; ++i) {
23                  a[i] = a[i%argc];
24          }
25
26          // output data to prevent compiler from removing any code
27          for(int i = 0; i < SIZE; ++i) {
28                  printf("%d ", a[i]);
29                  printf("%d ", b[i]);
30          }
31          printf("\n");
32
33          return 0;
34  }
```

1.2 Makefile

```
CFLAGS = -O2 -ftree-vectorize -fopt-info-vec-all-internals

.PHONY: all
all: analysis

.PHONY: clean
clean:
        $(RM) analysis

analysis: analysis.c
```

1.3 Discussion of Compiler Output

The full compiler output can be found in the file 08/ex1/output.log.

2 Investigation of Code Snippets

2.1 Safety of Parallelization

The safety of parallelization of the following code is examined.

```
void copy(double* x, double* y) {
    for(int i = 0; i < 1024; i++) {
        x[i] = y[i];
    }
}
```

It is possible that the arrays x and y overlap each other. One way to parallelize this manually is by using a temporary array z with the 1024*sizeof(double) and splitting the loop into two so that all read accesses from y (storing in z) are done in the first and all write accesses to x (reading from z) are done in the second. Both loops can run in parallel.

The function cannot be parallelized by the compiler, because the compiler needs the __restrict__ keyword for both arguments, which implies that two pointers cannot point to overlapping memory regions.

2.1.1 Manually parallelized Loop

```
void copy(double* x, double* y) {
    double* z = malloc(1024*sizeof(*z));
    if(!z) {
            return;
    }
    #pragma omp parallel for
    for(int i = 0; i < 1024; i++) {
        z[i] = y[i];
    }
    #pragma omp parallel for
    for(int i = 0; i < 1024; i++) {
        x[i] = z[i];
    }
    free(z);
}
```

2.2 Loop Normalization

The following loop should be normalisized.

```
for (int i=4; i<=N; i+=9) {
    for (int j=0; j<=N; j+=5) {
```

```
3           A[i] = 0;
4       }
5   }
```

2.2.1 Normalisized Loop

```
1   for (int i=0; i<=(N-4)/9; i+=1) {
2       for (int j=0; j<=N/5; j+=1) {
3           A[i*9+4] = 0;
4       }
5   }
```

2.3 Parallelizability

It is examined if the following loop is parallelizable.

```
1   for(int i = 1; i < N; i++) {
2       for(int j = 1; j < M; j++) {
3           for(int k = 1; k < L; k++) {
4               a[i+1][j][k-1] = a[i][j][k] + 5;
5           }
6       }
7   }
```

The distance vector for all dependencies is $(1, 0, -1)$ and therefore the corresponding direction vector is $(<, =, >)$. Thus the outmostloop is not parallelizable. The dependency of the first inner loop is loop-independent, therefore that loop can be parallelized.

2.3.1 Parallelized Loop

```
1   for(int i = 1; i < N; i++) {
2       #pragma omp parallel for
3       for(int j = 1; j < M; j++) {
4           for(int k = 1; k < L; k++) {
5               a[i+1][j][k-1] = a[i][j][k] + 5;
6           }
7       }
8   }
```