Sheet 05

PS Parallel Programming

Patrick Wintner

April 8, 2025

1 Missing Flush Directives

Effects of missing flush directives are observed.

1.1 Source Code

```c
#include <omp.h>
#include <stdio.h>

int main() {

    int data;
    int flag = 0;

    #pragma omp parallel num_threads(2)
    {

      if (omp_get_thread_num() == 0) {

          data = 42;

          flag = 1;

      } else if(omp_get_thread_num() == 1) {

          int flag_val = 0;
```

```
21
22              while (flag_val < 1) {
23
24                  flag_val = flag;
25
26              }
27
28              printf("flag=%d data=%d\n", flag, data);
29
30          }
31
32      }
33
34      return 0;
35  }
```

The program spawns two threads. Thread 0 does some work (setting the value of the variable data) before setting a flag. Thread 1 should print the values of the flag and the variable after the other thread has finished his work.

1.2 Experiment Method

The experiment was done on the LCC3 cluster by calling

salloc –exclusive –tasks-per-node=1 –cpus-per-task=1 srun –pty bash .

Followed by calling

./main.sh

manually. The following scripts are involved in the experiment.

1.2.1 Main Script

```
1  #!/bin/bash
2  # Usage: ./main.sh
3  make
4  for i in {0..999}
```

```
do
        ./ex1
done
make clean
```

1.3 Experiment Results

The program neither terminates nor prints any output. This is probable the case because thread 1 does not fetch the updated value of flag from shared memory. and is thus not able to leave the loop.

1.4 Discussion

The program does indeed require several flush directives, see the code below. Those are placed either after a variable (that is read in another thread) is written to or before a variable (that is written in another thread) is read. It does not require any atomic directives, because there is no variable to which is written in both threads.

```
#include <omp.h>
#include <stdio.h>

int main() {

    int data;
    int flag = 0;

    #pragma omp parallel num_threads(2)
    {

      if (omp_get_thread_num() == 0) {

          data = 42;
          #pragma omp flush(data)

          flag = 1;
          #pragma omp flush(flag)

      } else if(omp_get_thread_num() == 1) {

          int flag_val = 0;
```

```
23
24              while (flag_val < 1) {
25
26                 #pragma omp flush(flag)
27                 flag_val = flag;
28
29              }
30
31              #pragma omp flush(data)
32              printf("flag=%d data=%d\n", flag, data);
33
34          }
35
36      }
37
38      return 0;
39  }
```

2 Parallelising Code Snippets

2.1

2.1.1 Original

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
    b[i] = a[i] - a[i-1];
}
```

2.1.2 Fixed

```
a[0] = 0;
#pragma omp parallel for ordered
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
    #pragma omp ordered {
    b[i] = a[i] - a[i-1];
```

```
    }
}
```

The ordered construct is necessary to guarantee that the instructions are executed strictly in order.

2.2

2.2.1 Original

```
a[0] = 0;
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<N; i++) {
        a[i] = 3.0*i*(i+1);
    }
    #pragma omp for
    for (i=1; i<N; i++) {
        b[i] = a[i] - a[i-1];
    }
}
```

2.2.2 Fixed

```
a[0] = 0;
#pragma omp parallel
{
    #pragma omp for
    for (i=1; i<N; i++) {
        a[i] = 3.0*i*(i+1);
    }
    #pragma omp for
    for (i=1; i<N; i++) {
        b[i] = a[i] - a[i-1];
    }
}
```

If threads do not wait after the first loop, it is possible that some elements of b are set with values of some elements of a that are not yet initialisized.

2.3

2.3.1 Original

```
#pragma omp parallel for default(none)
for (i=0; i<N; i++) {
    x = sqrt(b[i]) - 1;
    a[i] = x*x + 2*x + 1;
}
```

2.3.2 Fixed

```
#pragma omp parallel for default(none) private(i, x) shared(a, b) ordered
for (i=0; i<N; i++) {
    #pragma ordered {
        x = sqrt(b[i]) - 1;
    }
    a[i] = x*x + 2*x + 1;
}
```

Setting default to none requires the programmer to specify the storage attribute for each variable explicitly.

2.4

2.4.1 Original

```
f = 2;
#pragma omp parallel for private(f,x)
for (i=0; i<N; i++) {
    x = f * b[i];
    a[i] = x - 7;
}
a[0] = x;
```

2.4.2 Fixed

```
f = 2;
#pragma omp parallel for firstprivate(f) lastprivate(x)
for (i=0; i<N; i++) {
    x = f * b[i];
    a[i] = x - 7;
}
a[0] = x;
```

The value of private variables is undefined on entry and exit of parallel regions.

2.5

2.5.1 Original

```
sum = 0;
#pragma omp parallel for
for (i=0; i<N; i++) {
    sum = sum + b[i];
}
```

2.5.2 Fixed

```
sum = 0;
#pragma omp parallel for reduction(+: sum)
for (i=0; i<N; i++) {
    sum = sum + b[i];
}
```

The reduction cause generates a local copy of sum and initialisizes it with 0. Updates occur on local copies (this would become a problem if sum were shared), which are afterwards combined to a single value.

2.6

2.6.1 Original

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
    #pragma omp for
    for (j=0; j<N; j++) {
        a[i][j] = b[i][j];
    }
}
```

2.6.2 Fixed

```
#pragma omp parallel for private(j)
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = b[i][j];
    }
}
```

Avoids nesting of parallel regions. Only the loop variable of the parallel for loop is by default privat, therefore it is necessary to specify j as private. Another solution is to allow nesting of parallel regions by using the runtime routine omp_set_nested().