1 Parallelizing Code with OpenMP

1.1 Chapter 3 of Introduction to Parallel Computing

1.1.1 Comparing code snippets

The code in listing 3.1 reads as follows:

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

int main() {
    printf("Hello, world:");
    #pragma omp parallel
        printf(" %d", omp_get_thread_num());
    printf("\n");
    return 0;
}
```

In this code, a team of thread get created my the *#pragma omp parallel* clause. Their task is to print their respective thread numbers. But since no order is defined, the order of their execution are "random", i.e. determined at runtime by the operating system.

Listing 3.2 reads as follows:

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

long fib(int n) {
    return (n < 2 ? 1 : fib(n-1) + fib(n-2));
}

int main() {
    int n = 45;
    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        printf("%d: %d\n", t, fib(n+t));
    }
    return 0;
}
```

While still not defining the order of the tasks, they yet seem to execute in order. Interestingly, this is not quite true, as the main reason the seem to do so is because the *terminate* in order. This is simply due to the fact that the work load for a thread with a higher thread number is substantially higher than for another one with a lower number. This is because fib(n+t) is invoked, but it takes very long to calculate because of the big n, and even a singe step by t causes fib(n+t) to take nearly twice as long.

1.1.2 Game of Life

The code in Listing 3.10 looks well designed. It collapses the two nested for loops of size $size$ into a single one of size $size^2$. Since each iteration approximately takes the same amount of work, the implicit static scheduling ensures a good distribution of the workload. This way, every thread has to compute $\frac{size^2}{\text{omp\_get\_num\_threads()}}$ iterations.

The outer most loop cannot be parallelized, as their iterations are all strictly dependent on the previous one. This is why the implicit barrier in line 13 is neccessary. It also shouldn't be that big of a bottleneck, as the threads should reach it almost simultaneously.

1.1.3 Optimizing Random Shooting

When trying to approximate π by the method of random shooting we use random numbers. They, however, don't need to be truly random, they just have to simulate it well enough. One way to do this simulation of randomness is the rnd function like it is defined in the book (or see the code below). It calculates a random number quite efficiently. Another benfit is that we do not need multiple instances of random number generators, and we also don't run the risk of misusing those by calling their methods in different threads. The code is shown in the figure below. It runs very quickly and calculates π very precisely. The running time on my particular machine was 0.0313137 s, in comparision to the previous version with a running time of 0.117771 s.

```cpp
#include <iostream>
#include <omp.h>
#include <random>

using namespace std;

double rnd(unsigned int *seed)
{
    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
    return ((double)(*seed)) / (1 << 24);
}

int main()
{
    int n = 100000000;              // number of points to generate
    int counter = 0;                // counter for points lying in the
        first quadrant of a unit circle
    auto start_time = omp_get_wtime(); // omp_get_wtime() is an OpenMP
        library routine

    // compute n points and test if they lie within the first quadrant
        of a unit circle
    #pragma omp parallel reduction(+:counter)
    {
        unsigned int seed = omp_get_thread_num();
        size_t local_n = (n / omp_get_num_threads()) + ((n %
            omp_get_num_threads() > omp_get_thread_num()) ? 1 : 0);

        for (int i = 0; i < local_n; ++i)
        {
            auto x = rnd(&seed); // generate random number between 0.0
                and 1.0
            auto y = rnd(&seed); // generate random number between 0.0
                and 1.0
            if (x * x + y * y <= 1.0)
            { // if the point lies in the first quadrant of a unit circle
                ++counter;
            }
        }
    }

    auto run_time = omp_get_wtime() - start_time;
    auto pi = 4 * (double(counter) / n);

    cout << "pi: " << pi << endl;
    cout << "run_time: " << run_time << " s" << endl;
    cout << "n: " << n << endl;
}
```