# Improving conjugate gradient and multi-grid applications

*Source code from this homework can be find in the archive directory, along with a README file.*

## Introduction

The goal of this project is to study the parallelization of conjugate gradient and multi-grid applications. Both applications were already optimized thanks to OpenMP primitives. These primitives have been modified in order to improve execution time.

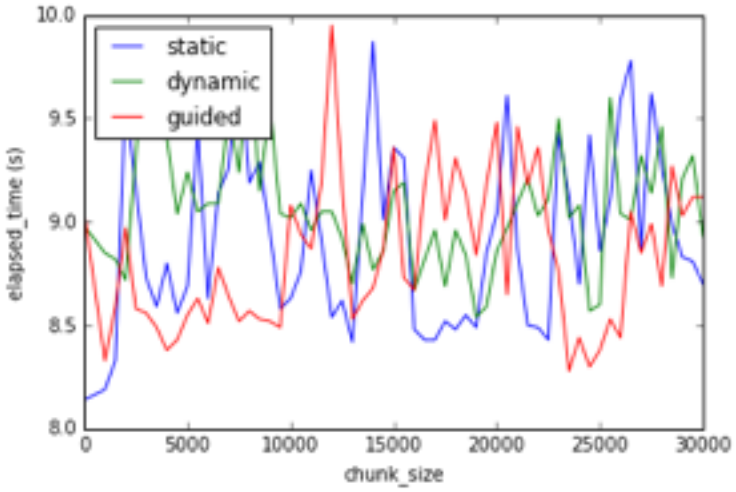
## Study of the given code

The code already contained lots of OpenMP primitives. Most of the for loops were preceded by `#pragma omg parallel for`. Besides, sections of codes outside loops represented a poor workload compared to loops themselves. Thus, the only solutions I founded in order to improve execution time were:
(1) Set loop scheduling;
(2) Try loop reductions.

## First method: setting loop scheduling

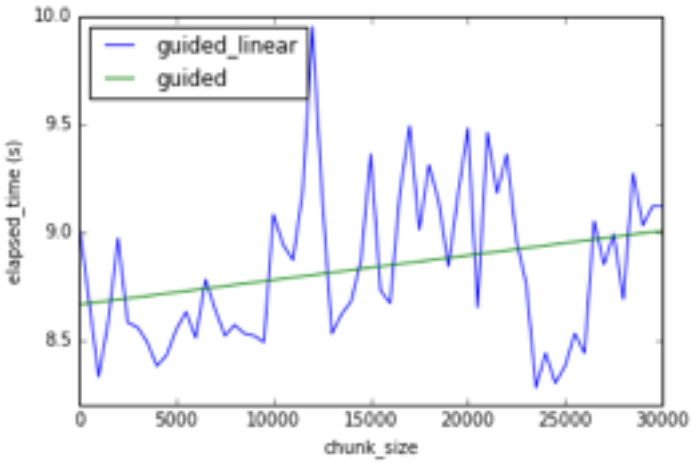
No loop was manually scheduled, which means that their schedule was `static` with a chunk size of `1`. Thanks to a test script, I tried to find the best scheduling trying every scheduling type for different chunk sizes. Studying the impact of scheduling does not seem to be possible thanks to library routines: `omp_get_schedule()` only works into parallel sections which makes it useless for computing global execution time. I had to set environment variables manually and define scheduling at runtime.

The results of these tests can be seen in the following figures.




Unfortunately, we can see that no scheduling seems to be significantly better than another one. However, computing the mean of elapsed times, we see that `guided` scheduling seems to be slightly better, I will then use it for the following tests.

Then, I had to pick the best chunk size for `guided` scheduling. Although data for `guided` scheduling are homogenous, they seemed to follow a trend, which I tried to highlight thanks to linear regression, as shown in the next figure.




We can see that, if we ignore the behavior around 25000 chunk size, the smaller chunk size the better. Thus, for following tests, I used a chunk size of `1` (default chunk size).

Surprisingly, execution time seems better when scheduling method is given directly into the code (and not at runtime). It is to say that

```
#pragma omp parallel for schedule(runtime)
...
$ ./bin/cg.C.x OMP_SCHEDULE="guided"
```

is slower than

```
#pragma omp parallel for schedule(guided)
...
$ ./bin/cg.C.x
```

I found no explanation for this improvement. However, I then used the second implementation in both applications.

## Second method: trying loop reduction

Loops with a possible reduction already had it implemented. However, nested loops in both programs, aven though they could be reducted, were not. I then tried to implement reduction in those loops, which was tough because of the private variable they used (`suml`).

```
#pragma omp for schedule(guided)
  for (j = 0; j < lastrow - firstrow + 1; j++) {
    suml = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
      suml = suml + a[k]*z[colidx[k]];
    }
    r[j] = suml;
}
```

The solution I founded was:

```
#pragma omp parallel shared(_suml)
{
      #pragma omp for schedule(guided) reduction(+:_suml)
      for (j = 0; j < lastrow - firstrow + 1; j++) {
        suml = 0.0;
        for (k = rowstr[j]; k < rowstr[j+1]; k++) {
          _suml = _suml + a[k]*z[colidx[k]];
        }
        r[j] = _suml;
      }
}
```

However, results were horrible, it took age for the program to end, and the number of operations per thread turned small.

# Conclusion

In conclusion, it was very hard to find any idea for optimizing this code which seemed already optimized. Most of directives were here, covering significant workload. I am curious to see what other students did, in order to deepen my understanding of OpenMP.