Final Project OpenMP 2 - SEISMIC

Student: Tiago de Souza Oliveira

Toolkit:

- icc -g -pg -qopenmp -o seismic_omp seismic_omp.c
- gprof -l seismic_omp > seismic_omp_gprof.out
- more seismic_omp_gprof.out
- For sbatch:

```
openmp > $ job_seismic_omp.sh

#!/bin/bash
#SBATCH -o %x-%J.out
#SBATCH -e %x-%J.error

#SBATCH -J omp_job_seismic       # Job name
#SBATCH -o omp_job_seismic.o%j   # Name of stdout output file(%j expands to jobId)
#SBATCH -e omp_job_seismic.o%j   # Name of stderr output file(%j expands to jobId)

#SBATCH --time=0-00:05:00 #requested time to run the job
#SBATCH -c 32 #(64 cores per job)
#SBATCH -t 00:10:00 #(10 min of execution time)
#SBATCH --mem=16GB #(4GB of memory)
#SBATCH --exclusive

export OMP_NUM_THREADS=8
time ./seismic_omp
```

By sticking to the Incremental Parallelization strategy, I tried to focus on the most expensive parts of the application. From that perspective, the gprof report gives some hints as it shows below in terms of routine and subroutine time consumption, computational cost, call frequency and so on.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 65.77    296.39   296.39                             main (seismic_omp.c:321 @ 401fdb)
 21.53    393.41    97.03                             main (seismic_omp.c:320 @ 401fa1)
 10.01    438.53    45.12                             main (seismic_omp.c:319 @ 401f5d)
  1.04    443.24     4.71                             main (seismic_omp.c:300 @ 401da8)
  0.57    445.79     2.55                             main (seismic_omp.c:298 @ 401d4d)
  0.18    446.62     0.83                             main (seismic_omp.c:301 @ 401dd2)
  0.14    447.24     0.62                             main (seismic_omp.c:275 @ 401b70)
  0.13    447.83     0.59                             main (seismic_omp.c:332 @ 402111)
  0.12    448.38     0.55                             main (seismic_omp.c:296 @ 401cf4)
```

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 450.95 seconds

index % time    self  children    called     name
                0.00    0.00       1/151500039     main (seismic_omp.c:165 @ 40134c) [222]
                0.00    0.00       1/151500039     main (seismic_omp.c:167 @ 401351) [223]
                0.00    0.00       1/151500039     main (seismic_omp.c:205 @ 4015cb) [241]
                0.00    0.00       1/151500039     main (seismic_omp.c:206 @ 4015ed) [242]
                0.00    0.00       1/151500039     main (seismic_omp.c:207 @ 401617) [243]
                0.00    0.00       1/151500039     main (seismic_omp.c:211 @ 401641) [244]
                0.00    0.00       1/151500039     main (seismic_omp.c:221 @ 40168f) [250]
                0.00    0.00       1/151500039     main (seismic_omp.c:228 @ 4016ff) [254]
                0.00    0.00       1/151500039     main (seismic_omp.c:237 @ 401876) [262]
                0.00    0.00       2/151500039     main (seismic_omp.c:236 @ 40184c) [261]
                0.00    0.00      14/151500039     main (seismic_omp.c:226 @ 4016bb) [252]
                0.00    0.00      14/151500039     main (seismic_omp.c:229 @ 401729) [255]
                0.00    0.00 1500000/151500039     main (seismic_omp.c:280 @ 401c05) [284]
                0.00    0.00 150000000/151500039     main (seismic_omp.c:275 @ 401b70) [7]
[29]     0.0    0.00    0.00 151500039         main (seismic_omp.c:134 @ 4012e8) [29]
-----------------------------------------------
```

It is hard to interpret the regular gprof report. Therefore, I could not come to a conclusion based on those statistics. Nevertheless, I could reason about functions called from the long Main implementation, where I spot **smvp** as a classic vector product algorithm

For that, I performed another query to gprof, but this time only for getting information about the smvp function.
prof -p**smvp** -b seismic_omp gmon.out > analysis.txt

```
openmp > analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.01     16.50    16.50     3855     4.28     4.28  smvp
```

In order to better evaluate the parallelism approach that better fits from the efficiency and consistency perspectives, I plan to setup an OMP environment on my ubuntu machine with appropriate profiling tools like Intel Advisor, Valgrind/Callgrind, CloverLeaf/KCacheGrind and some other gprof parameters as well. Then I think I would be able to better design the parallelism, which I have been thinking would demand some changes in the code..

To do so, I have played the hiphotesis that if I could set the accumulator apart from the array and use a local variable instead, I would be able to leverage "#pragma omp parallel firstprivate (Anext) reduction (+ : variable1,…)", for instance. That's the incomplete implementation that can be seen in the code.
The tests show inconsistencies when calculating **epicenternode** values.

```c
double sum0, sum1, sum2;
double sumw0, sumw1, sumw2;

  for (i = 0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

    sumw0 = w[Acol[Anext + 1]][0], sumw1 = w[Acol[Anext + 1]][1], sumw2 = w[Acol[Anext + 1]][2];

  //Hipothesis that replacing array with a local variable, to use the cache better, avoid false sharing and improve locality
  #pragma omp parallel firstprivate ( Anext ) reduction ( + : sum0, sum1, sum2, sumw0, sumw1, sumw2)
  {
    Anext++;
    while (Anext < Alast) {
      col = Acol[Anext];

      sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
      sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
      sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];

      /*
      w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
      w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
      w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
      */

      sumw0 += w[col][0] + A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
      sumw1 += w[col][1] + A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
      sumw2 += w[col][2] + A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];

      Anext++;
    }


    w[i][0] = sumw0 + sum0;
    w[i][1] = sumw1 + sum1;
    w[i][2] = sumw2 + sum2;

  }
}
```

In summary, this is still a Work In Progress for me, and I should invest more time on a better profiling tool for a loop interaction to choose the right OMP parallelism instructions and the correct changes in the code to accommodate higher scalability.