Assignment 3

High Performance Parallel Computing 2022

Kimi Cardoso Kreilgaard & Linea Stausbøll Hedemark

March 2022

1 OpenMP parallelise the program

Before using OpenMP on the code we first analyse which parts can be parallised, and find that the `Propagator` function is full of for loops that can easily be parallised with `#pragma omp parallel`. Furthermore we find that there are to tasks in the `FFT` function that can be solved independently of each other, and this function is therefor an obvious candidate to use `#pragma omp task` on. The implementation is described for each function below:

- The `Propagator` function:
  - First we open a parrallel scope with `#pragma omp parallel`
  - Within the function almost all loops are parallised with `#pragma omp for` which is just a standard way one can do this. There are however a few exceptions, where we have to be careful of which pragma we are using.
  - Twice we add `nowait` to `#pragma omp for` if the next loop to be parallised is independent of the current loop, it is not necessary to have a barrier here making all threads wait for the result of the current loop.
  - When a value only has to be computed once, i.e. when there is no loop, we add `#pragma omp single` which ensures that only one thread will be working on the next line, despite it being within a parallel scope. The advantage to doing this and not ending the scope right before, is that we only have to start up the threads once. We also use single before the `FFT` functions.
  - We use `#pragma omp for reduction(+: mean_wave)` once, in the loop where all iterations write to the same variable mean wave, to ensure that this would not be a problem.
  -
- The `FFT` function:
  - This function is always used by a single thread due to the `#pragma omp single` input right before this line.

- The `FFT` function used the `FFT` function within itself and can therefore be said to be recursive, meaning that the parallisation is not as straight forward as we saw in the other function. We first identify that `fft(even)` and `fft(odd)` within the function can run independent of each other so we add `#pragma omp task shared(even/odd)` right before this function to tell the main thread that it can delegate this task. We also state that the variable within, either even or odd, has to be shared so all threads has access to it.
- We add `#pragma omp taskwait` right after the two tasks, to tell the main thread that it has to wait for the update in place from the two tasks before it can run the next part of the code.
- Since the function is recursive it continues to run new `FFT` function within until the length of the input array is 1. This means that we will run `FFT` more times than we have threads. We therefor add an if statement ensuring that there are free threads available before it delegates a task. There are free threads if the following condition is upheld: $N > nsamp/ncores$.

Finally we check that the results obtained, including the checksum, are the same for the sequential code and the parallised code.

2 Strong and Weak Scaling Using SLURM

To benchmark the performance of the parallisation we run the parallel code with SLURM using the provided `job.sh` file. We change `cpus-per-task` to the number of threads we want. We use note the performance for 1,2,4,8,16,32 and 64 threads.

2.1 Strong Scaling

Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size. More specifically we will be looking at the speedup as a function of number of threads used. We fix the problem size by keeping $nfreq = 512 \cdot 1024$. The speedup here is defined as: $\textit{speedup}(N) = t(1)/t(N)$, where $t(1)$ is the elapsed time for one thread, i.e. the elapsed time for the sequential code, and $t(N)$ is the elapsed time for N threads. Since there are multiple components to the code we plot the observed scaling for the entire code, for the FFT part of the code and finally for the code without the FFT part. The result is plotted in Fig. 1 below. Since the result is not linear, which is what we would expect for the ideal case, it becomes clear that there is a sequential fraction of the code that cannot be parallised. This is what is known as Amdahl's law, and it tells us that the problem will not continue to scale forever with increasing number of threads since there is a sequential part that will fill more and more of the run time of the code. Comparing the different graphs in the plot we see that this problem is bigger for the FFT part of the code, where there is a bigger fraction of sequential code.

2.2 Weak Scaling

Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size per processor. To keep the problem size fixed per thread used, we let the problem size $nfrec$

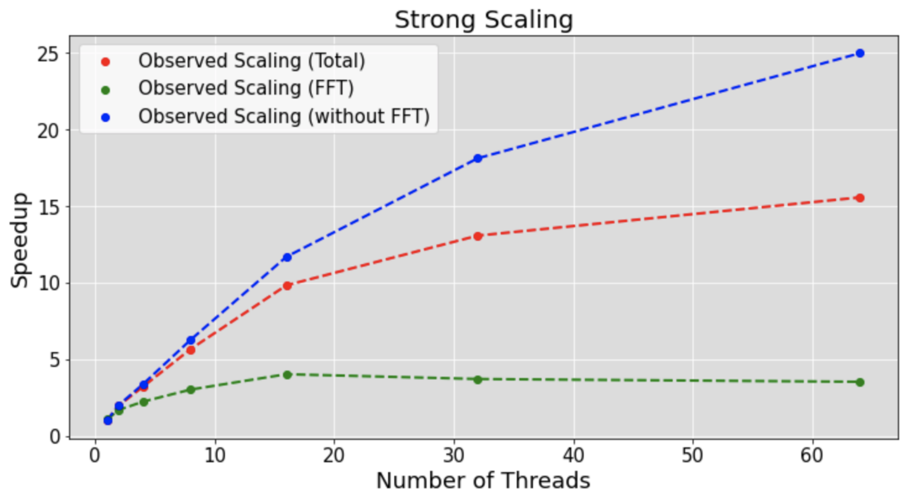

Figure 1: Strong scaling

depend on the number of threads: $nfrec = N \cdot 128 \cdot 1024$. The speed up for weak scaling is defined as: $speedup(N) = t(1)/t(N) \cdot (nfrec(N)/nfrec(1))$. Where the last term scales the speed up according to the increased workload for a larger number of threads. The results are plotted in Fig. 2 below. We see here again that the FFT part is not linear in cost as a function of $nfreq$, meaning that the full code will not be linear either. When subtracting the FFT time from the total runtime we see that the scaling is almost linear, as expected.

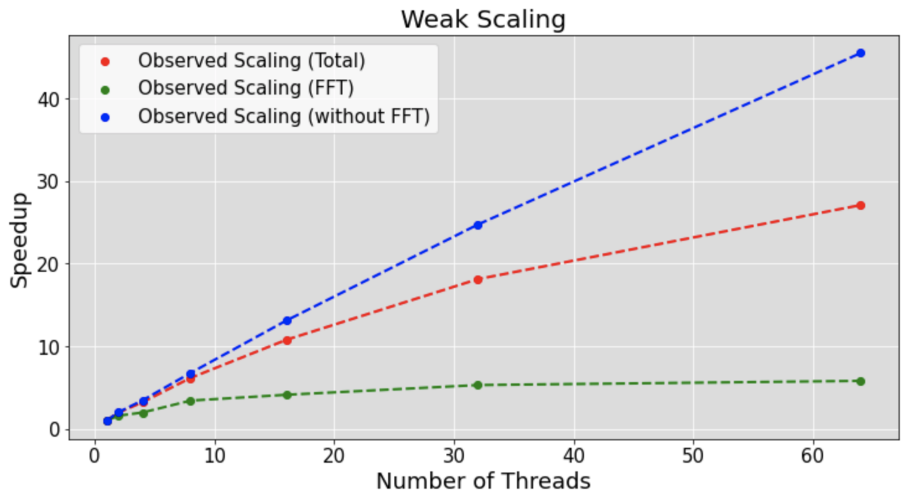

Figure 2: Weak scaling

3 The Code

```
/*
  Assignment: Make an OpenMP parallelised wave propagation
  model for computing the seismic repsonse for a wave
  propagating through a horizontally stratified medium
*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <random>
#include <chrono>
#include <thread>
#include <complex>
#include <cmath>

/* INCLUDE OMP */
#include <omp.h>

// shorthand name for complex number type definition below
typedef std::complex<double> Complex;

// ======================================================
// The number of frequencies sets the cost of the problem
const long nfreq=512*1024; // frequencies in spectrum WAS 16*1024 BEFORE
// ======================================================

// Initialize Basic Constants
const double dT=0.001; // sampling distance
const long nsamp=2*nfreq; // samples in seismogram

// Frequency resolution (frequency sampling distance)
double dF = 1/(nsamp*dT);

// read data file with one number per line
std::vector<double> read_txt_file(std::string fname) {
    std::vector<double> data; // vector of data points
    std::string line; // string to read in each line
    std::ifstream file(fname); // open file
    while (std::getline(file, line)) // loop over lines until end of file
        data.push_back(std::stod(line)); // convert string to double and store in array
    return data;
}

// CooleyTukey FFT (in-place computation)
void fft(std::vector<Complex>& x)
{
        const long N = x.size();
        if (N <= 1) return;

        // divide
        std::vector<Complex> even(N/2), odd(N/2);
        for (long i=0; i<N/2; i++) {
            even[i] = x[2*i];
            odd[i] = x[2*i+1];
        }

    // Get number of cores
    int ncores = omp_get_num_threads();

        // conquer

    // Check if there are any cores free to perform a task, if yes do parrallel
    if( N > nsamp/ncores) {

        // omp task says that this task can be done independently of the next task
        // the variable even is shared so all can access it
        #pragma omp task shared(even)
        fft(even);

        // same here
        #pragma omp task shared(odd)
        fft(odd);

        // For the next part we need both results, so we wait for the tasks to complete
        #pragma omp taskwait
    }

    // If no cores are free, don't parallise it
    else {
        fft(even);
        fft(odd);
    }

        // combine
        for (long k = 0; k < N/2; k++)
        {
                Complex t = std::polar(1.0, -2 * M_PI * k / N) * odd[k];
                x[k ] = even[k] + t;
                x[k+N/2] = even[k] - t;
        }
}

// inverse fft (in-place)
void ifft(std::vector<Complex>& x)
{
    double inv_size = 1.0 / x.size();
    for (auto& xx: x) xx = std::conj(xx); // conjugate the input
        fft(x);  // forward fft
    for (auto& xx: x)
        xx = std::conj(xx) // conjugate the output
            * inv_size; // scale the numbers
}


// Main routine: propgate wave through layers and compute seismogram
std::vector<double> propagator(std::vector<double> wave,
                               std::vector<double> density,
                               std::vector<double> velocity) {
    const long nlayers = density.size();
    std::vector<double> imp(nlayers); // impedance
    std::vector<double> ref(nlayers-1); // reflection coefficient
    std::vector<Complex> half_filter(nfreq/2+1,1); // half filter
    std::vector<Complex> filter(nfreq+1); // full filter
    std::vector<double> half_wave(nfreq+1,0); // half wave
    std::vector<Complex> wave_spectral(nsamp); // FFT(wave)
    std::vector<Complex> U(nfreq+1,0); // Upgoing waves
    std::vector<Complex> Upad(nsamp,0); // FFT(seismogram)
    std::vector<double> seismogram(nsamp); // final seismogram
    long n_wave = wave.size(); // size of wave array
    long lc = std::lround(std::floor(nfreq*0.01)); // low-cut indices
    double mean_wave = 0.; // wave zero point

    std::chrono::time_point<std::chrono::system_clock> tstart1,tstart2,tend1,tend2;

    // Start a new parallel scope, meaning we only start up the threads here
    #pragma omp parallel
    {
        // Compute seismic impedance
        #pragma omp for
        for (long i=0; i < nlayers; i++)
            imp[i] = density[i] * velocity[i];

        // Reflection coefficients at the base of the layers
        // The next loop if not dependent on ref so we don't have to wait here
        #pragma omp for nowait
        for (long i=0; i < nlayers-1; i++)
            ref[i] = (imp[i+1] - imp[i])/(imp[i+1] + imp[i]);

        // Spectral window (both low- and high cut)
        #pragma omp for
        for (long i=0; i < lc+1; i++)
            half_filter[i]= (sin(M_PI*(2*i-lc)/(2*lc)))/2+0.5;

        #pragma omp for
        for (long i=0; i < nfreq/2+1; i++)
            filter[i] = half_filter[i];

        #pragma omp single
        filter[nfreq/2+1] = 1;

        // The next loop is not dependent on filter so we dont have to wait here
        #pragma omp for nowait
        for (long i=nfreq/2+2; i < nfreq+1; i++)
            filter[i] = half_filter[nfreq+1-i];

        #pragma omp for
        for (long i=0; i < n_wave/2; i++)
            half_wave[i] = wave[n_wave/2-1+i];

        // Make reduction so not all threads try to write to mean_wave at the same time (race cond.)
        #pragma omp for reduction(+: mean_wave)
        for (long i=0; i < 2*nfreq; i++) {

            if (i < nfreq) {
                wave_spectral[i] = half_wave[i];
            } else {
                wave_spectral[i] = half_wave[2*nfreq-i];
            }
            mean_wave += std::real(wave_spectral[i]);
        }

        // This is not a loop, so a single thread should perform this
        #pragma omp single
        mean_wave = mean_wave / nsamp;

        #pragma omp for
        for (long i=0.; i < 2*nfreq; i++)
            wave_spectral[i] -= mean_wave;

        // A single thread needs to set up the threads that are started within FFT
        #pragma omp single
        {
            // Fourier transform waveform to frequency domain
            tstart1 = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)
            fft(wave_spectral);
            tend1 = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)
        }

        // spectrum U of upgoing waves just below the surface.
        // See eq. (43) and (44) in Ganley (1981).
        #pragma omp for
        for (long i=0; i < nfreq+1; i++) {
            Complex omega{0, 2*M_PI*i*dF};
            Complex exp_omega = exp( - dT * omega);
```

```
            Complex Y = 0;
            for (long n=nlayers-2; n > -1; n--)
                Y = exp_omega * (ref[n] + Y) / (1.0 + ref[n]*Y);
            U[i] = Y;
        }

        // Compute seismogram
        #pragma omp for
        for (long i=0; i < nfreq+1; i++) {
            U[i] *= filter[i];
            Upad[i] = U[i];
        }

        #pragma omp for
        for (long i=nfreq+1; i < nsamp; i++)
            Upad[i] = std::conj(Upad[nsamp - i]);

        #pragma omp for
        for (long i=0; i < nsamp; i++)
            Upad[i] *= wave_spectral[i];

        // A single thread needs to set up the threads that are started within FFT
        #pragma omp single
        {
            // Fourier transform back again
            tstart2 = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)
            ifft(Upad);
            tend2 = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)
        }

        #pragma omp for
        for (long i=0; i < nsamp; i++)
            seismogram[i] = std::real(Upad[i]);
    }

    // Print results
    std::cout << "Wave␣zero-point␣␣␣␣␣␣␣␣:␣" << std::setw(9) << std::setprecision(5)
              << mean_wave<< "\n";
    std::cout << "Seismogram␣first␣coeff␣:␣" << std::setw(9) << std::setprecision(5)
              << seismogram[0] << ",␣" << seismogram[1] << ",␣" << seismogram[2] << ",␣" << seismogram[3] <<"
                  ↪ \n";
    std::cout << "Elapsed␣time␣for␣FFTs␣␣:" << std::setw(9) << std::setprecision(4)
              << ((tend1 - tstart1).count() + (tend2 - tstart2).count())*1e-9 << "\n";

    return seismogram;
}

//
    ↪ ================================================================
      ↪
//========================= Main function
      ↪ =================================================================
//
      ↪ ==========================================================================
      ↪
int main(int argc, char* argv[]){
    // Load the wave profile and the density and velocity structure of the rock from text files
    std::vector<double> wave = read_txt_file("wave_data.txt"); // input impulse wave in medium
    std::vector<double> density = read_txt_file("density_data.txt"); // density as a function of depth
    std::vector<double> velocity = read_txt_file("velocity_data.txt"); // seismic wave velocity as a function of depth

    auto tstart = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)

    // Propagate wave
    std::vector<double> seismogram = propagator(wave,density,velocity);

    auto tend = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)

    // write output and make checksum
    double checksum=0;
    std::ofstream file("seismogram.txt"); // open file
    for (long i=0; i < nsamp; i++) {
        file << seismogram[i] << '\n';
        checksum += abs(seismogram[i]);
    }

    // Print more results
    std::cout << "Elapsed time:" << std::setw(9) << std::setprecision(4)
              << (tend - tstart).count()*1e-9 << "\n";
    std::cout << "Checksum    :" << std::setw(20) << std::setprecision(15)
              << checksum << "\n";
}
```