OpenMP Final project

Unai Iborra

This document will explain how the provided code seismic.c has been parallelized and optimized to use multiple threads versus a sequential execution.

All the discussed code of this document, both sequential and parallel has been compiled with the gcc standard optimization level -O2. More performance might be obtained from using -O3 or -Ofast, but as the document focuses in analyzing how the performance of the parallelized code has been obtained, only -O2 has been used.

The provided zip includes a Makefile and two scripts for executing and testing the code. In order to build the code, there is no need to use the make command, it has been automatized using a shell script named test.sh. Just give execution permissions to the file and execute it. This will compile the code and run the parallel version of the code with the time command, in order to analyze the total performance of the code. It will also tell if the results are exactly the same as for the sequential code, comparing the created stdout saved in /results with the sequential pre-run stdout. Some parameters have been defined inside test.sh in order to adjust the thread number and preprocessor defines. More information about this script will be explained later.

This essay will talk about benchmarks and time results. The shown times have been taken with the average of five executions. The number of threads used will be specified for each result.

With all the introduction explained, the first step in order to analyze what parts of the code should be optimized is to get the total time results of each part of the code. Gprof has been used for that task.

Analysis of the sequential code with gprof

In order to analyze what parts of the code are the most computationally costly parts. Thanks to the use of gprof, a summary of that was obtained. The obtained summary has been provided into the task zip named as gprof_ft3.txt. This summary shows how the most computationally costly part of all the sequential program execution is clearly the smvp function. The second most time consuming part of the execution is the main function body itself (not considering function calls from main). The other parts of the execution have negligible cost for the total execution time, and are mostly the initialization part of the code (allocation, variable initialization, etc.). Those parts are not very relevant for the optimization of the code as they are only performed once in the initialization stage.

Because of this, the first step has been to parallelize the smvp function.

Analysis of the smvp function and parallelization

The need to parallelize the smvp function is clear, the first step to parallelizing smvp I took was to analyze if it was actually parallelizable, if it had dependencies, and if it had them if they could be managed.

After looking at the function I realised that it had race conditions if was just parallelized with just a #pragma omp parallel for. When accumulating over the variable w, both in the while loop and each iteration of the for loop, w may be attempted to be written by more than one thread, thus creating race conditions that makes the final result of the simulation wrong.

The first thing that was made in order to parallelize smvp was to allocate an array for each available thread, allowing for each thread to accumulate the values of w over its own array. The decision to allocate once for each thread instead of one big allocation that then the threads would index into, was to facilitate each core to have the fastest memory access possible. This is because the Finisterrae III processors use a NUMA architecture, so by allocating and first accessing each allocation by each thread, I attempted to make the virtual memory page correspond with the phyisical memory closests to each core.

In order to allocate each array and access it from each core the following code was made inside the mem_init function:

```
/* OMP VARIABLES INITIALIZATION */

unsigned int num_threads = omp_get_max_threads();

// double ** local_w init
local_w = malloc(num_threads * sizeof(double *));
if (!local_w) {
	printf("Bad allocation of local_w");
	exit(1);
}

#pragma omp parallel private(i)
{
	unsigned int thread_n = omp_get_thread_num();

	// disp[3][ARCHnodes][3]
	size_t local_w_size = ARCHnodes * 3;

	local_w[thread_n] = malloc(local_w_size * sizeof(double));

	for (i = 0; i < local_w_size; i++) {
	local_w[thread_n][i] = 0.0; // Initialize the memory in the numa core correctly
}
```

Then, in order to use it in the smvp function, each thead, by using its thread id, could get their asigned array and accumulate over It:

```
	unsigned int t_id = omp_get_thread_num();
```

```
        double *lw = local_w[t_id];
```

For the full parallelized function, see Exhibit 1.1. As it can be seen, first, the parallel region was created, then, each thread gets its index to the previously allocated array and used it to accumulate over it. Then, each array would be zeroed with a memset, then accummulated and finally, by an accumulation over the real w variable, performed the total accumulation (it's a reduction over an array). A simple reduction with omp pragmas could not be done as it is a reduction over an array and not over a single value.

The parallelization of this function this way, although making the function parallel, added some overhead to the overall structure of the function, as zeroeing each local array and then accumulating it over the w variable was needed.

In order to test the speedup gotten from this parallelization, some benchmarks were made in the FT3. The exexution results were around 21.408s with 8 threads (mean of 5 executions, they were very close from each other) and 18.809s with 32 threads. The results were an improvement from the 47.215s gotten from the sequential benchmark, but didn't clearly scale well and were not a fully functioning solution yet. The main hypothesis about why the parallelization was not scaling well was that the overhead from the memset and accumulating over the w variable was the main problem. Another theorized cause was that the code was memory bound and was constantly accessing memory, causing performance problems.

Parallelyzing the time integration loop

The previous gprof results showed the smvp function using 73.32% of the program execution time. Almost all the remaining time was spent inside the main function, using 21.45% of the time. The next logical step was to analyze the main function, where it can be parallelized and if that parallelization was made in a repeated zone.

After some time analyzing the main function, the conclusion that the time integration loop was where the main function spent most time. This was obvious by analyzing that the smvp function was called there and that the loop was where each time step was calculated.

The iteration loop couldn't be parallelized as each step is dependant from the previous, but all the loops inside that time step could be parallelized. The parallelization each step seemed in a first glance a lot easier than the smvp function. Most loops inside the integration loop did not have dependencies and could be easily parallelizable inside a single parallel region. The provided code along with this document contains the optimized version of the time integration loop.

The changes made for this parallelization were the following:

- Create one single parallel region: In the first versions of the parallelization of a step, parallel regions were created and destroyed continuously, mainly to avoid calling the smvp function from the parallel region. The solution to avoid the overhead from constantly creating and destroying parallel regions was to change slightly the smvp function and delete its parallel region creation. This is because as it was going to be

called from a parallel region, there was no need for creating it again (it would create a thread pool for each thread that called, and would not be correct).

- Parallelizing each for loop: as loops didn't have dependencies I parallelized them just using a pragma parallel for. I also added nowait, as always the same i and j was accessing different data of the different arrays/matrices.

I didn't expect those changes to affect much in the total execution time. My assumption was wrong. Just by making those changes, the code with 8 cores had results of 8.412s and 4.098s with 32 cores.

The explanation of why the performance improved that much is probably because even when the main function took 21.45% of the total execution time, overhead from creating and destroying multiple regions was not done, and mainly because of optimizing those for loops.

With 32 cores a speedup of ~11.52x was decent but not exceptional. The hypothetical speedup for 32 cores is 32x (if everything could be parallelizable) so I thought there was still some room for optimization.

Final optimization

The next step was to attempt to reduce the overhead introduced in the smvp function when parallelized. The main theorized overhead from the implementation of the smvp parallelized function is zeroing the arrays asigned for each thread and the final reduction over the w variable.

Two attemps were made that attempted to skip those overheads:

1. Using critical regions: This solution gave correct results but it was even worse performing than the sequential execution. It was discarded as it was obviously not a valid solution.
2. Using atomics for each operation performed over the w variable: This solution is the final optimization of the code and gives the best results.

smvp array reduction using atomics

The atomic implementation over the smvp function used almos the same code as the sequential version, just parallelizing the loop and protecting the reductions made in the w variable with atomics:

```
#pragma omp atomic
w[col][0] += A[Anext][0][0] * v[i][0] + A[Anext][1][0] * v[i][1] +
A[Anext][2][0] * v[i][2];
#pragma omp atomic
w[col][1] += A[Anext][0][1] * v[i][0] + A[Anext][1][1] * v[i][1] +
A[Anext][2][1] * v[i][2];
#pragma omp atomic
```

```
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++;
}

#pragma omp atomic
w[i][0] += sum0;
#pragma omp atomic
w[i][1] += sum1;
#pragma omp atomic
w[i][2] += sum2;
```

Using atomics, also adds overhead but as it seems from the benchmark results, its overhead is far smaller than the overhead added from the previous implementation of smvp.

The final results from the benchmark using the atomic and previous implementation have been taken by using an exclusive node and setting export OMP_NUM_THREADS=X in the slurm script. All the executions were ran in the Finisterrae III. The critical implementation has not been benchmarked as it performs worse than the sequential implementation.

Threads	**First implementation time[1]**	**Atomic implementation time[2]**	**Speedup for atomic implementation[3]**
1 (sequential, compiled without -fopenmp)	47.215s		-
2	27.310s	30.013s	~1.573
4	14.709s	15.764s	~2.995
8	8.343s	8.509s	~5.548
16	5.188s	4.614s	~10.232
32	4.120s	2.671s	~17.676
64	3.861s	1.942s	~24.31

The atomic implementation results are far better performing than the previous implementation to the atomic one. The explanation for this must be that in the previous implementation, for each thread, for each call of the smvp function all the data from the thread had to be stored into the w variable, causing a memory bound code. With atomics, the scaling of the speedup seems far better, and reaching a final speedup of ~24.31 with 64 threads. The scaling is not obviously 100% scaling with the number of threads but that is expected.

1 Average of 5 executions
2 Average of 5 executions
3 Compared to the 47.215s gotten in the sequential executions

The previous implementation, however, seems to work better when being used with less threads (2-8). Probably due to not needing to reduce as many data back to the w variable and performing less operations over the bus.

How to compile and test the code

In order to compile and run the code, the script test.sh has been made. There what implementation to test can be decided changing the DEFINE variable (options are ATOMIC, CRITICAL and NUMA (first implementation)). The number of threads can also be defined in that script. It will also run the sequential code in order to test if the results are correct and print --- OK --- if they are.

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
```

Has been defined in the script, as this article showed how to use them and what they do in NUMA processors. Although, I did not notice any significant change in performance by using them.

Exhibit 1

1.1 smvp first optimization for use with good numa access

```
void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v,
double **w)
{
	// double t0 = omp_get_wtime();

#pragma omp parallel // in order to privatize easily the declarations
		{
			int i;
			int Anext, Alast;
			size_t col, lw_size;
			double sum0, sum1, sum2;

			unsigned int t_id = omp_get_thread_num();

			// local_w is in another numa core probably, so I
			// take it once to the thread stack
			double *lw = local_w[t_id];
			lw_size = nodes * 3 * sizeof(double);

			/* Displacement array disp[3][ARCHnodes][3] */
			// disp = (double ***)malloc(3 * sizeof(double **));
			memset(lw, 0.0,
			nodes * 3 * sizeof(double)); // clean the local w for next use

#pragma omp for nowait
			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];

				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];

					lw[col * 3] += A[Anext][0][0] * v[i][0] +
					A[Anext][1][0] * v[i][1] +
					A[Anext][2][0] * v[i][2];

					lw[col * 3 + 1] += A[Anext][0][1] * v[i][0] +
					A[Anext][1][1] * v[i][1] +
					A[Anext][2][1] * v[i][2];

					lw[col * 3 + 2] += A[Anext][0][2] * v[i][0] +
```

```
                                A[Anext][1][2] * v[i][1] +
                                A[Anext][2][2] * v[i][2];
                                Anext++;
                }

                lw[i * 3 + 0] += sum0;
                lw[i * 3 + 1] += sum1;
                lw[i * 3 + 2] += sum2;
        }
}

#pragma omp parallel
{
        int num_threads = omp_get_num_threads();
#pragma omp for schedule(static)
        for (int i = 0; i < nodes; i++) {
                double s0 = 0.0, s1 = 0.0, s2 = 0.0;

                for (int t = 0; t < num_threads; t++) {
                        double *lw = local_w[t];
                        s0 += lw[i * 3 + 0];
                        s1 += lw[i * 3 + 1];
                        s2 += lw[i * 3 + 2];
                }

                w[i][0] += s0;
                w[i][1] += s1;
                w[i][2] += s2;
        }
}
}
```