Hackaton

Team 8

November 28th to December 5th

Contents

1 Organization

Github link: https://github.com/Hackathon-team8

For this project, we were asked to to work on two different code source. The first one was a Stencil code that only needed to be optimized and the other one was the Saturn code that had to be ported on ARM architecture and optimized if possible.
To complete our task we divided the workload and structured it in a way that everyone can have something to do. During our measures for the Stencil, our approach was to improve the code step by step and compare our result with the default one provided by Steve Messenger

2 Stencil

The goal of the Stencil is to execute the program as fast as possible. For that we did different optimizations that are tied to either the architecture of the cluster we had access to, the algorithm of the code or both. The machine on which we did our tests has 16 cores, 1 thread per cores a L1 cache of 64KB, a L2 cache of 1024K and a L3 cache of 32768K. This information is needed to understand how our data will be stored, therefore read in the future.

2.1 Compilation

The very first thing we did was to look at the compile options. We used the GNU compiler GCC, or to be more precise G++, for our tests as not only are we familiar with it, it offers the most options among the other ones available on the cluster. And it's the one who produced the best code by default.

```
#!/bin/bash
g++ -Ofast stencil.cxx -o stencil -fopenmp -mcpu=native -finline-functions -funroll-loops -ftree-loop-vectorize -ftree-vectorize
```

```
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
Acceleration:
10.54390707697388
```

On the second picture we can see the difference of accuracy between the program that we have now and the program at the beginning, then we can how mush faster we are.
We decided to use the Ofast compile flag as the loss in precision we have is negligible compared to the speed up gain it brings. With the O3 flag, we have on precision nigh no precision loss however we observed that our overall speed up was divided by two. As mentioned before, this is not something we judged as necessary for our case.

2.2 Data Structure

After giving the compiler the flags to direct him in how we want it to compile our code, the next step is to help it do so. Thus, the next thing we did was to rearrange how the data was structured in memory. While our tensors are stored in RAM due to their size, which will bring down performance, we want them to be stored continuously in memory to facilitate vectored operations.

```
double *__restrict matA;
double *__restrict matB;
double *__restrict matC;
```

```
matA = (double *)aligned_alloc(64,s);
assert( matA!=NULL);
matB = (double *)aligned_alloc(64,s);
assert( matB!=NULL);
matC = (double *)aligned_alloc(64,s);
assert( matC!=NULL);
```

```
[0, 3.6070128953549106e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 3.6070128953549106e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 3.6070128953549106e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 3.6070128953549106e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 3.6070128953549106e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
Acceleration:
12.564605788876861
```

The restrict tells the compiler that tensors don't cross in the memory, (every values of A can't be in the B tensor memory), and the aligned_alloc make all the values of the tensor contiguous in memory. As we can see our program is even faster than before, the compiler was able to vectorize some calculation in the **One Iteration** function. Instead of doing the calculations one by one he did it four by four. Our speed up was not as good as it could be, which implies that the compiler didn't vectorize every thing in the program.

2.3 Pow

For this improvement we noticed that there was a division with powers of 17, these values being the same for each value of o (from 1 to 8) we decided to calculate these values in advance in a vector by multiplying each time the previous value by 17 (which gives us all our values of 17 power o) in order not to recalculate them each time what allows us to make less calculation. Especially since the pow function is a very expensive function which slows down the program a lot.

```
vector<double> power_17;
```

```
power_17.push_back(1.0);
for(unsigned int i = 1; i <= order; i++)
        power_17.push_back(power_17[i-1]*17);
```

```
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x+o,y,z)]*matB[DIMXYZ(x+o,y,z)] / power_17[o];
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x-o,y,z)]*matB[DIMXYZ(x-o,y,z)] / power_17[o];
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y+o,z)]*matB[DIMXYZ(x,y+o,z)] / power_17[o];
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y-o,z)]*matB[DIMXYZ(x,y-o,z)] / power_17[o];
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y,z+o)]*matB[DIMXYZ(x,y,z+o)] / power_17[o];
matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y,z-o)]*matB[DIMXYZ(x,y,z-o)] / power_17[o];
```

```
Precision:
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
Acceleration:
13.826411804468417
```

As expected we have a speed up. Not only did we remove an expensive and badly optimized function pow, but we also allowed the compiler to vectorize the division by giving it values stored in L1 cache in read only mode.

2.4 OpenMP

We are now going to talk about one of the optimization that will improve the program the most, this optimization is the parallelisation of the program. The goal is to divide a huge workload over different cores. This allows us to use all the available cores in the architecture to reduce the burden of the main core and, if everything works accordingly, multiply our speed up by an amount close to the number of threads allocated.

```
#pragma omp parallel
	{
		omp_set_dynamic(0);
		const int n_threads = omp_get_num_threads();
		omp_set_num_threads(n_threads);

		#pragma omp for schedule(dynamic,1)
		for (ui64 z = 0; z < DIMZ; z++) {
			for (ui64 y = 0; y < DIMY; y++){
				for (ui64 x = 0; x < DIMX; x++){
					matC[DIMXYZ(x,y,z)] = matA[DIMXYZ(x,y,z)]*matB[DIMXYZ(x,y,z)] ;
					for (ui64 o = 1; o <= order; o++){
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x+o,y,z)]*matB[DIMXYZ(x+o,y,z)] / power_17[o];
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x-o,y,z)]*matB[DIMXYZ(x-o,y,z)] / power_17[o];
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y+o,z)]*matB[DIMXYZ(x,y+o,z)] / power_17[o];
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y-o,z)]*matB[DIMXYZ(x,y-o,z)] / power_17[o];
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y,z+o)]*matB[DIMXYZ(x,y,z+o)] / power_17[o];
						matC[DIMXYZ(x,y,z)]+= matA[DIMXYZ(x,y,z-o)]*matB[DIMXYZ(x,y,z-o)] / power_17[o];
					}
				}
			}
		}
		//  A=C
		#pragma omp for schedule(dynamic,1)
		for (ui64 z = 0; z < DIMZ; z++) {
			for (ui64 y = 0; y < DIMY; y++){
				for (ui64 x = 0; x < DIMX; x++){
					matA[DIMXYZ(x,y,z)] = matC[DIMXYZ(x,y,z)];
				}
			}
		}
	}
}
```

```
Precision:
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
Acceleration:
252.72867341489118
```

First of all, we **export** the number of threads we want in the sbatch. Then, we use **pragma omp parallel** to create the parallel region.After multiple tests, we settled for **schedule(dynamic, 1** for our scheduling. This makes sense as the assembly code generated by our compiler tells us that only the last loop has been unrolled and each iterations will have a fast execution time so having a **guided** schedule would be detrimental due to the greater overhead time.

2.5 Intrinsic

We decide to try using intrinsic calls. Not only can they help the compiler to vectorize some parts of the code, but it's also a simplified assembly code which makes it easier to ask the compiler to do what we want. The problem is that sometimes, the compiler won't always follow our instructions. Moreover, while exploring the **objdump** of the executable we noticed that the compiler was doing write after write in the same register, therefore wasting precious time doing operations he could have skipped. So we decided to try to guide him, and make him just use the vectorized fma operations.

```
inline void compute(const ui64 x,
                    const ui64 y,
                    const ui64 z,
                    const ui64 o,
                    svbool_t pg){
        svfloat64_t A = svld1(pg,&matA[DIMXYZ(x+o,y,z)]);
        svfloat64_t B = svld1(pg,&matB[DIMXYZ(x+o,y,z)]);
        svfloat64_t C = svld1(pg,&matC[DIMXYZ(x,y,z)]);
        svfloat64_t pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);

        A = svld1(pg,&matA[DIMXYZ(x-o,y,z)]);
        B = svld1(pg,&matB[DIMXYZ(x-o,y,z)]);
        pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);

        A = svld1(pg,&matA[DIMXYZ(x,y+o,z)]);
        B = svld1(pg,&matB[DIMXYZ(x,y+o,z)]);
        pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);

        A = svld1(pg,&matA[DIMXYZ(x,y-o,z)]);
        B = svld1(pg,&matB[DIMXYZ(x,y-o,z)]);
        pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);

        A = svld1(pg,&matA[DIMXYZ(x,y,z+o)]);
        B = svld1(pg,&matB[DIMXYZ(x,y,z+o)]);
        pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);

        A = svld1(pg,&matA[DIMXYZ(x,y,z-o)]);
        B = svld1(pg,&matB[DIMXYZ(x,y,z-o)]);
        pow = svld1rq(pg,&power_17[o]);
        B = svdiv_m(pg,B,pow);
        C = svmad_x(pg, A, B, C);
        svst1(pg,&matC[DIMXYZ(x,y,z)],C);
    }
```

```
uint64_t x = 0;
svbool_t pg = svwhilelt_b64(x, DIMX);
do{
        matC[DIMXYZ(x,y,z)] = matA[DIMXYZ(x,y,z)]*matB[DIMXYZ(x,y,z)] ;
        compute(x, y, z, 1, pg);
        compute(x, y, z, 2, pg);
        compute(x, y, z, 3, pg);
        compute(x, y, z, 4, pg);
        compute(x, y, z, 5, pg);
        compute(x, y, z, 6, pg);
        compute(x, y, z, 7, pg);
        compute(x, y, z, 8, pg);

        matC[DIMXYZ(x+1,y,z)] = matA[DIMXYZ(x+1,y,z)]*matB[DIMXYZ(x+1,y,z)] ;
        compute(x+1, y, z, 1, pg);
        compute(x+1, y, z, 2, pg);
        compute(x+1, y, z, 3, pg);
        compute(x+1, y, z, 4, pg);
        compute(x+1, y, z, 5, pg);
        compute(x+1, y, z, 6, pg);
        compute(x+1, y, z, 7, pg);
        compute(x+1, y, z, 8, pg);

        matC[DIMXYZ(x+2,y,z)] = matA[DIMXYZ(x+2,y,z)]*matB[DIMXYZ(x+2,y,z)] ;
        compute(x+2, y, z, 1, pg);
        compute(x+2, y, z, 2, pg);
        compute(x+2, y, z, 3, pg);
        compute(x+2, y, z, 4, pg);
        compute(x+2, y, z, 5, pg);
        compute(x+2, y, z, 6, pg);
        compute(x+2, y, z, 7, pg);
        compute(x+2, y, z, 8, pg);

        matC[DIMXYZ(x+3,y,z)] = matA[DIMXYZ(x+3,y,z)]*matB[DIMXYZ(x+3,y,z)] ;
        compute(x+3, y, z, 1, pg);
        compute(x+3, y, z, 2, pg);
        compute(x+3, y, z, 3, pg);
        compute(x+3, y, z, 4, pg);
        compute(x+3, y, z, 5, pg);
        compute(x+3, y, z, 6, pg);
        compute(x+3, y, z, 7, pg);
        compute(x+3, y, z, 8, pg);
        x += svcntd();
        pg = svwhilelt_b64(x,DIMX);
}while(x < DIMX);
```

```
Precision:
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.0856380132123286e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.1852879923951595e-14, 0, 0, 0, 0, 0]
Acceleration:
119.14643818538238
```

We sadly did not manage to get better results with this method. Either the intrinsic calls we did were not the best one or the compiler did not understand what we wanted to do.

2.6 Interesting modification

This part gathers several optimizations that have been done and fall into the category of tricks and pattern recognition. Together, they have given us an interesting acceleration.

First of all, we noticed that we were calling several times the same vector containing the values (see part on pow above) so we used variables that contains the values of the vector so we don't have to load them from the vector every time we need to divide. The variables are **const**, to force a read-only behavior on the address. The compiler can do more optimizations with **const** variables, thus increasing the efficiency and consistency of prior optimizations.

For the second optimization, we reduced the number of write in the tmp tensor(previously C). We did every calculation before which allows us to store less often, speeding up the program.

```
const double val = power_17[1];
const double val2 = power_17[2];
const double val3 = power_17[3];
const double val4 = power_17[4];
const double val5 = power_17[5];
const double val6 = power_17[6];
const double val7 = power_17[7];
const double val8 = power_17[8];
```

```
Precision:
[0, 2.708379924885843e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 4.977310148740515e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
Acceleration:
1051.403556572074
```

As mentioned before, these optimizations are not that impactful by themselves, but together and alongside the previous ones, it creates a virtuous cycle that allows the compiler to produce a better executable.

2.7 multiply before

Here we saw that each time we multiply both A tensor and B tensor in the same nested loop (multiply A[X] with B[x]). The idea is to isolate this operation so it can be more easily vectorized and avoid having to read from both structures again and only from tmp. Indeed, this application has a memory issue. The fact that our data sets are stored in RAM prevents us from gaining more speed up.

```
omp_set_dynamic(0);
const int n_threads = omp_get_num_threads();
omp_set_num_threads(n_threads);

#pragma omp for schedule(dynamic, 1)
for (ui64 z = 0; z < DIMZ; ++z) {
        for (ui64 y = 0; y < DIMY; ++y){
                for (ui64 x = 0; x < DIMX; ++x){
                        tmp[DIMXYZ(x,y,z)] = matA[DIMXYZ(x,y,z)]*matB[DIMXYZ(x,y,z)];
                }
        }
}

#pragma omp for schedule(dynamic, 1) // test with guided
for (ui64 z = 0; z < DIMZ; ++z)
{
        for (ui64 y = 0; y < DIMY; ++y)
        {
                for (ui64 x = 0; x < DIMX; ++x)
                {
                        matA[DIMXYZ(x,y,z)] = tmp[DIMXYZ(x,y,z)];

                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+1,y,z)] + tmp[DIMXYZ(x-1,y,z)] + tmp[DIMXYZ(x,y+1,z)] + tmp[DIMXYZ(x,y-1,z)] + tmp[DIMXYZ(x,y,z+1)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+2,y,z)] + tmp[DIMXYZ(x-2,y,z)] + tmp[DIMXYZ(x,y+2,z)] + tmp[DIMXYZ(x,y-2,z)] + tmp[DIMXYZ(x,y,z+2)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+3,y,z)] + tmp[DIMXYZ(x-3,y,z)] + tmp[DIMXYZ(x,y+3,z)] + tmp[DIMXYZ(x,y-3,z)] + tmp[DIMXYZ(x,y,z+3)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+4,y,z)] + tmp[DIMXYZ(x-4,y,z)] + tmp[DIMXYZ(x,y+4,z)] + tmp[DIMXYZ(x,y-4,z)] + tmp[DIMXYZ(x,y,z+4)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+5,y,z)] + tmp[DIMXYZ(x-5,y,z)] + tmp[DIMXYZ(x,y+5,z)] + tmp[DIMXYZ(x,y-5,z)] + tmp[DIMXYZ(x,y,z+5)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+6,y,z)] + tmp[DIMXYZ(x-6,y,z)] + tmp[DIMXYZ(x,y+6,z)] + tmp[DIMXYZ(x,y-6,z)] + tmp[DIMXYZ(x,y,z+6)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+7,y,z)] + tmp[DIMXYZ(x-7,y,z)] + tmp[DIMXYZ(x,y+7,z)] + tmp[DIMXYZ(x,y-7,z)] + tmp[DIMXYZ(x,y,z+7)] +
                        matA[DIMXYZ(x,y,z)]+= (tmp[DIMXYZ(x+8,y,z)] + tmp[DIMXYZ(x-8,y,z)] + tmp[DIMXYZ(x,y+8,z)] + tmp[DIMXYZ(x,y-8,z)] + tmp[DIMXYZ(x,y,z+8)] +
                }
        }
```

```
Precision:
[0, 2.708379924885843e-14, 3.14073869201969e-14, 5.32057291761917e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 5.32057291761917e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 5.32057291761917e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 5.32057291761917e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
[0, 2.708379924885843e-14, 3.14073869201969e-14, 5.32057291761917e-15, 1.9536619748829894e-14, 2.095727009100278e-14, 0, 0, 0, 0, 0]
Acceleration:
2004.77314324282
```

We got a good speed up due to the fact that the multiplication happens beforehand, the second loop is less congested and we also need less memory access because we only use tmp instead of both A and B tensors.

2.8 Conclusion

We did a lot of optimization for the Stencil and as we can all of them aren't as efficient as the other but all together we can go 2000 time faster than the beginning. We tried some optimization which did not have a big effect. There are other methods we could have tried to improve the stencil like block algebra, however the lack of time restricted us.

3 Saturn

Introduction of the Saturn code and its purpose.

In order to ensure a clean environment for the following tests we need to make sure that our environment is clean with **module purge**.

Listing 1 – Script to load compile and run ARMPL tool-chains

```
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

module use /opt/nvidia/hpc_sdk/modulefiles
module use /opt/tools-hackathon/arm/compilers/modulefiles
module use /opt/tools-hackathon/arm/forge/modulefiles
module use /opt/tools-hackathon/openmpi/modulefiles

module load arm-forge
module load acfl
module load armpl
module load openmpi/gcc
module load gnu

alias 'python'='python3'

export CC="mpicc"
export CXX="mpic++"
export FC="gfortan"
export DEST=$HOME/code_saturne-7.2.0

export ARMPLROOT="/opt/tools-hackathon/arm/compilers/armpl-22.1.0_AArch64_RHEL-7_arm-linux-co

alias 'code_saturne'="~/code_saturne-7.2.0/bin/code_saturne"
```

3.1 Warm up

To get used to our working environment, we first checked the hardware that was at our disposition. Slurm is used to manage the different node. To get info about these node we used the command **sinfo** as follow:

```
[alaplanche@ip-10-8-15-90 source]$ sinfo -o "%N %.6D %P %.11T %.4c %.8z %.6m %.8d %.6w %.8f %20E"
NODELIST  NODES PARTITION       STATE CPUS    S:C:T MEMORY TMP_DISK WEIGHT AVAIL_FE REASON
basic-slurm-dy-c7g-4xlarge-[1-4]      4 basic-slurm*       idle~   16   16:1:1  31129        0      1 dynamic, none
[alaplanche@ip-10-8-15-90 source]$
```

We can get more information about the node using the command **srun -N1 -n1 -p basic-slurm –pty lscpu** As

```
[cwyeumobarkwende@ip-10-8-15-90 ~]$ srun --ntasks=1 --cpus-per-task=1 -p basic-slurm --exclusi
ve --pty lscpu
Architecture:        aarch64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              16
On-line CPU(s) list: 0-15
Thread(s) per core:  1
Core(s) per socket:  16
Socket(s):           1
NUMA node(s):        1
Vendor ID:           ARM
Model:               1
Stepping:            r1p1
BogoMIPS:            2100.00
L1d cache:           64K
L1i cache:           64K
L2 cache:            1024K
L3 cache:            32768K
NUMA node0 CPU(s):   0-15
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asi
mdrdm jscvt fcma lrcpc dcpop sha3 sm3 sm4 asimddp sha512 sve asimdfhm dit uscat ilrcpc flagm s
sbs paca pacg dcpodp svei8mm svebf16 i8mm bf16 dgh rng
```

we can see on this picture, there is 16 cores available on each node.

There is different type of SIMD in ARM: like NEON or SVE. SVE is more performante than NEON. We used the following command to get information about wether NEON or SVE was used in these processor:

```
[alaplanche@ip-10-8-15-90 source]$ cat /proc/cpuinfo | grep "Features"
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 sm3 sm4
asimddp sha512 sve asimdfhm dit uscat ilrcpc flagm ssbs paca pacg dcpodp svei8mm svebf16 i8mm bf16 dgh rng
```

These command give us all the flags of our processor, which means most of the characteristics of the processor. We can see that theses processors support SVE vectorization.

Now we want to have information about on which OS the cluster is running. To get this information we use the command **cat /etc/os**.

Before using Saturn, we want to load a certain amount of modules: for example our compilers or a math lib. But it can be redundant and can cause issue when we forget to load them. To make it automatic, we put the following command in our .bashrc:

Listing 2 – Example of .bashrc to automatically use these different path

```
module use /opt/nvidia/hpc_sdk/modulefiles
module use /opt/tools-hackathon/arm/compilers/modulefiles
```

```
[alaplanche@ip-10-8-15-90 source]$ cat /etc/os-release
NAME="Amazon Linux"
VERSION="2"
ID="amzn"
ID_LIKE="centos rhel fedora"
VERSION_ID="2"
PRETTY_NAME="Amazon Linux 2"
ANSI_COLOR="0;33"
CPE_NAME="cpe:2.3:o:amazon:amazon_linux:2"
HOME_URL="https://amazonlinux.com/"
```

```
module use /opt/tools-hackathon/arm/forge/modulefiles
module use /opt/tools-hackathon/openmpi/modulefiles
```

After telling module that we use the different path to store modules, we want to see what modules are available. We used the following command and get this output:

```
[alaplanche@ip-10-8-15-90 source]$ module avail

------------------------- /opt/tools-hackathon/arm/compilers/modulefiles/gnu/../../moduledeps/gnu/11.2.0 -------------------------
armpl/22.1.0

------------------------------------------- /opt/tools-hackathon/openmpi/modulefiles -------------------------------------------
openmpi/acfl/4.1.4 openmpi/gcc/4.1.4

---------------------------------------- /opt/tools-hackathon/arm/forge/modulefiles ----------------------------------------
arm-forge/22.1

---------------------------------------- /opt/tools-hackathon/arm/compilers/modulefiles ----------------------------------------
acfl/22.1       binutils/11.2.0 gnu/11.2.0

------------------------------------------------ /opt/nvidia/hpc_sdk/modulefiles ------------------------------------------------
nvhpc/22.9              nvhpc-byo-compiler/22.9 nvhpc-nompi/22.9

------------------------------------------------ /usr/share/Modules/modulefiles ------------------------------------------------
armpl/21.0.0                module-git                  null
dot                         module-info                 openmpi/4.1.4
libfabric-aws/1.16.0~amzn3.0 modules                    use.own
```

The following command can be used to get information about where the loaded MPI is installed:

```
[alaplanche@ip-10-8-15-90 ~]$ which mpicc
/opt/tools-hackathon/openmpi/acfl/bin/mpicc
```

To compile the Code Saturne , we need to use GNU, ARM CLANG or NVIDIA compiler.

- GNU compiler: **gcc**, **g++** and **gfortran** to compile respectively C, C++ and Fortran
- ARM CLANG compiler: **armclang**, **armclang++** and **armflang** to compile respectively C, C++ and Fortran
- NVIDIA compiler: **nvcc**, **nvc++** and **nvfortran** to compile respectively C, C++ and Fortran

After testing and loading multiple module we need to clean up our working environment. In order to do so, we execute the command **module purge**. It unload each loaded module.

ARM Alinea Studio comes with a mathematical librairie called ARMPL (ARM Performance Librairie) which implement an optimized version of the libraries BLAS, LAPACK and FFT. We are now ready to work on the code Saturne now that we get used to our working environment.

3.2 Porting to ARM

3.2.1 ACFL

The first things we have to do was to download the source code of Saturne. We use the command **wget**. To install Saturne we first have to export the libraries we want to use. To make it automatic we created a file that some command, we just need to use the command **source** on this file to execute the following command:

Listing 3 – File that was used to prepare the configuration process

```
# Load the module required to compile code Saturne with armclang
module load arm-forge
module load acfl
module load openmpi/acfl
# Load the ARM Performance Librairie
module load armpl

# Create environment variable to facilitate the configuration process
export CC="mpicc"
export CXX="mpic++"
export FC="armflang"
export DEST=$HOME/code_saturne-7.2.0

# Create an environment variable to store the ARMPL location
export ARMPLROOT="/opt/tools-hackathon/arm/compilers/armpl-22.1.0_AArch64_RHEL
  -7_arm-linux-compiler_aarch64-linux/include/"
```

Then we get inside the code Saturn sources (**cd code_saturne-7.2.0**/). We now have to run the command **./sbin/bootstrap** to create the configure file. Now we create a directory named build and we run the command:

```
../configure CC="$CC" CXX="$CXX" --with-blas=$ARMPLROOT --prefix=$DEST --disable-gui
--without-med --without-hfd5 --without-cgns --without-metis --disable-salome
--without-salome --without-eos --disable-static --enable-long-gnum
```

We get the following output:

```
checking for MKL libraries... no
checking for threaded ATLAS BLAS... no
checking for ATLAS BLAS... no
checking for legacy C BLAS... no
configure: error: in `/userdir/home/alaplanche/raw_saturne/build':
configure: error: BLAS support is requested, but test for BLAS failed!
See `config.log' for more details
```

As we can see, Saturn don't recognize our version of BLAS, we got the error "BLAS support is requested; but test for BLAS failed". We have to implement test for ARMPL BLAS. To do so we modify the file **m4/cs_blas.m4** and add the following lines:

Listing 4 – Line added to cs_blas.m4

```
# Test for ARMPL BLAS

 if test "x$with_blas_type" = "x" -o "x$with_blas_type" = "xARMPL" ; then

   if test "$1" = "yes" -o "x$with_blas_libs" = "x"; then # Threaded version ?

     BLAS_LIBS="-larmpl -lpthread"

     CPPFLAGS="${saved_CPPFLAGS} ${BLAS_CPPFLAGS}"
     LDFLAGS="${saved_LDFLAGS} ${BLAS_LDFLAGS}"
     LIBS=" ${BLAS_LIBS} ${saved_LIBS}"

     AC_MSG_CHECKING([for threaded ARMPL BLAS])
     AC_LINK_IFELSE([AC_LANG_PROGRAM([[#include <cblas.h>]],
                    [[ cblas_ddot(0, 0, 0, 0, 0); ]])],
                    [ AC_DEFINE([HAVE_ARMPL], 1, [ARMPL BLAS support])
                      cs_have_blas=yes; with_blas_type=ARMPL ],
                    [cs_have_blas=no])
     AC_MSG_RESULT($cs_have_blas)
   fi

   if test "$cs_have_blas" = "no" ; then # Test for non-threaded version
                                         # or explicitely specified libs second
     if test "x$with_blas_libs" != "x" -a "x$with_blas_type" = "xARMPL"; then
       BLAS_LIBS="$with_blas_libs"
     else
       BLAS_LIBS="-larmpl"
     fi

     CPPFLAGS="${saved_CPPFLAGS} ${BLAS_CPPFLAGS}"
     LDFLAGS="${saved_LDFLAGS} ${BLAS_LDFLAGS}"
     LIBS=" ${BLAS_LIBS} ${saved_LIBS}"

     AC_MSG_CHECKING([for ARMPL BLAS])
     AC_LINK_IFELSE([AC_LANG_PROGRAM([[#include <cblas.h>]],
                    [[ cblas_ddot(0, 0, 0, 0, 0); ]])],
                    [ AC_DEFINE([HAVE_ARMPL], 1, [ARMPL BLAS support])
                      cs_have_blas=yes; with_blas_type=ARMPL ],
                    [cs_have_blas=no])
     AC_MSG_RESULT($cs_have_blas)
   fi

 fi
```

After the change the configure command run as intended. But when we try top compile the code by typing **make** we got an error about a compilation flag that is unknown to ARMCLANG. When we read the full output, we saw "armflang is a NVIDIA compiler". The code Saturn isn't recognizing ARMCLANG as intended. We take a look at the file "cs_auto_flags.sh" which is supposed to detect the compiler used and add the right flags.

We notice that there is a part dedicated for ARM compatibility, but it is after the NVIDIA compiler part. Because of the bad test realized to test if the compiler was an NVIDIA compiler and because they put an else instead of an elif. If the compiler is not one of the first ones, it is set as an NVIDIA compiler, and it prevent the

following test to be made. To fix this issue we copy the part which concern ARM compiler, and past it before the test for NVIDIA compiler. We have to do that for each compiler i.e. C, C++ and FORTRAN. After these change the code Saturn compile as intended with ACFL compiler.

Now we want to check the dependencies of the binary to check if the compilation had no issue. So we type the command in the root of the code Saturn: **ldd libexec/code_saturne/cs_solver**. We get the following output:

Listing 5 – Output of ldd

```
...
libcs_solver-7.2.so => not found
libsaturne-7.2.so => not found
libple.so.2 => not found
...
```

As we can see here the linker failed to link with three libraries, which seems to contain the compiled code of code Saturn. So there is probably an issue with the linker. While exploring the file "cs_auto_flags.sh" we found out that there is a linker section at the end of the file and we never validate the condition to enter this block of code. If "cs_linker_set" is equal to no, some parameters are set to give flags to the linker. But "cs_linker_set" is set to yes in the C section of the ARM compiler (in cs_auto_flags.sh). So we remove this line and rebuild. Now when we check with **ldd**, the linked libraries everything is normal.

To create both mesh, we just need to go to the right directory and run the Saturn program. Then, to check whether or not the mesh was successfully generated we can use the command line : **grep -r -C2 "Extrusion" ***. This will allow use to verify that cells input and output are coherent with what was asked. And then to get the execution time, we used the command : **grep -r -C2 "Elapsed time:" *** which should be in the performance.log file of the run.

3.2.2 GNU

For the GNU compiler there is almost nothing to do. The only things to do is to execute these command or put in in a .bashrc to make them automatic:

Listing 6 – Things to do before installing Saturn with gcc

```
# Load the module required to compile code Saturne with gcc
module load arm-forge
module load armpl
module load openmpi/gcc
module load gnu

# Create environment variable to facilitate the configuration process
export CC="mpicc"
export CXX="mpic++"
export FC="gfortan"
export DEST=$HOME/code_saturne-7.2.0

# Create an environment variable to store the ARMPL location for gcc
export ARMPLROOT="/opt/tools-hackathon/arm/compilers/armpl-22.1.0_AArch64_RHEL
  -7_gcc_aarch64-linux/include/"
```

Now we build the code and verify it as been compile with GNU compilers. To do this we first check which

mpicc and mpic++ is used:

```
[alaplanche@ip-10-8-15-90 build]$ which mpicc
/opt/tools-hackathon/openmpi/gcc/bin/mpicc
[alaplanche@ip-10-8-15-90 build]$ which mpic++
/opt/tools-hackathon/openmpi/gcc/bin/mpic++
[alaplanche@ip-10-8-15-90 build]$
```

Then we check the output of "configure":

```
compiler 'mpicc' is GNU gcc-11.2.0
compiler 'mpic++' is GNU g++-11.2.0
compiler 'gfortran' is GNU gfortran-11.2.0
```

And the we type make and make install. We check the output of these commands:

```
mpicc -o cs_solver cs_solver.o -lcs_solver
```

```
libtool: compile:  gfortran -I../../src -I../../src/atmo
```

Now we check the dependence of the binary with the command **ldd libexec/code_saturne/cs_solver** and we get the following output:

Listing 7 – Output of ldd cs_solver compiled with GCC

```
linux-vdso.so.1 (0x0000ffffa2616000)
libcs_solver-7.2.so => /userdir/home/alaplanche/code_saturne-7.2.0/lib
  /libcs_solver-7.2.so
libsaturne-7.2.so => /userdir/home/alaplanche/code_saturne-7.2.0/lib
  /libsaturne-7.2.so
libple.so.2 => /userdir/home/alaplanche/code_saturne-7.2.0/lib/libple.so.2
libarmpl.so => /opt/tools-hackathon/arm/compilers/armpl-22.1.0_AArch64_RHEL-
  7_gcc_aarch64-linux/lib/libarmpl.so
libz.so.1 => /lib64/libz.so.1
libdl.so.2 => /lib64/libdl.so.2
libgfortran.so.5 => /opt/tools-hackathon/arm/compilers/gcc-11.2.0_Generic-AArch64_RHEL-
  7_aarch64-linux/lib64/libgfortran.so.5
libpthread.so.0 => /lib64/libpthread.so.0
libm.so.6 => /lib64/libm.so.6
libmpi.so.40 => /opt/tools-hackathon/openmpi/gcc/lib/libmpi.so.40
```

For a comparison between ACFL and GNU, we ran most of our tests with the ACFL configuration however, on the F128_01 run with 32 MPI process and 2 threads each, the GNU version is on average 15 seconds faster (283s for ACFL and 268s for GNU for the calculation time recorded in performance.log).

3.2.3 NVIDIA

The first things to do is to modify cs_auto_flags.sh:

Listing 8 – Modification in cs_auto_flags.sh to support NVIDIA compiler

```
...
820: fi
821: # Otherwise, are we using pgc++/nvc++ ?
822: #-------------------------------------
823: if test "x$cs_cxx_compiler_known" != "xyes"; then
824:     $CXX -V 2>&1 | grep 'NVIDIA' > /dev/null
...
```

We then build the program and get multiple error. To fix them we only add **-fPIC** in cs_auto_flags.sh in all the NVIDIA compiler section. Then the program build and run as intended.

Since NVBLAS is a CUDA library, and there is no cuda in code Saturn nor graphics card in this cluster, there is no need to implement NVBLAS. So code Saturn will be compile without BLAS when using the NVIDIA compiler.

3.3 MPI and OpenMP

First we want compile the program without openmp for that we have to configure using the flag –**disable-openmp**.

```
../configure CC="$CC" CXX="$CXX" --with-blas=$ARMPLROOT --prefix=$DEST --disable-gui --without-
med --without-hfd5 --without-cgns --without-metis --disable-salome --without-salome --without-eos
--disable-static --enable-long-gnum --disable-openmp
```

As we can see in the next picture, the openmp support is still active which mean that the flag didn't work as intended.

```
MPI (Message Passing Interface) support: yes
  MPI I/O support: yes
  MPI3 nonblocking barrier support: yes
OpenMP support: yes
OpenMP accelerator support: no
OpenMP Fortran support: yes
```

We looked inside all the configuration files to see if we can find were is the problem about the openmp support. After some research we found the openmp support in the file "configure.ac"

Listing 9 – The original configure.ac file

```
AC_ARG_ENABLE(openmp,
  [AS_HELP_STRING([--disable-openmp], [disable OpenMP support])],
  [
    case "${enableval}" in
      yes) cs_have_openmp=yes ;;
      no)  cs_have_openmp=no ;;
      *)   AC_MSG_ERROR([bad value ${enableval} for --enable-openmp]) ;;
    esac
  ],
  [ cs_have_openmp=yes ]
)

AC_ARG_ENABLE(openmp_target,
  [AS_HELP_STRING([--enable-openmp-target], [enable OpenMP accelerator support])],
  [
    case "${enableval}" in
      yes) cs_have_openmp_target=yes
           cs_have_openmp=yes ;;
      no)  cs_have_openmp_target=no ;;
      *)   AC_MSG_ERROR([bad value ${enableval} for --enable-openmp-target]) ;;
    esac
  ],
  [ cs_have_openmp=yes ]
)
```

We can see here the problem is that by default openmp is activated if there is no flag, but if we had the flag, the openmp support is still activated because in every case the "configure.ac" file activate the openmp support, to change that we have to change the last "cs_have_openmp=yes" to "cs_have_openmp=no", we also need to change the order of the two blocks which gives us

Listing 10 – modification to the configure.ac file

```
AC_ARG_ENABLE(openmp_target,
  [AS_HELP_STRING([--enable-openmp-target], [enable OpenMP accelerator support])],
  [
    case "${enableval}" in
      yes) cs_have_openmp_target=yes
           cs_have_openmp=yes ;;
      no)  cs_have_openmp_target=no ;;
      *)   AC_MSG_ERROR([bad value ${enableval} for --enable-openmp-target]) ;;
    esac
  ],
  [ cs_have_openmp=no ]
)

AC_ARG_ENABLE(openmp,
  [AS_HELP_STRING([--disable-openmp], [disable OpenMP support])],
  [
    case "${enableval}" in
      yes) cs_have_openmp=yes ;;
      no)  cs_have_openmp=no ;;
      *)   AC_MSG_ERROR([bad value ${enableval} for --enable-openmp]) ;;
    esac
  ],
  [ cs_have_openmp=yes ]
)
```

Now we want to look at the dependencies of the program with and without the openmp support activated. To see if it change anything about the dependencies when we disable openmp.

dependencies	without openmp	with openmp
linux-vdso.so	X	X
libamath_aarch64.so	X	X
libcs_solver-7.2.so	X	X
libsaturne-7.2.so	X	X
libple.so.2	X	X
libarmpl.so	X	X
libz.so.1	X	X
libdl.so.2	X	X
libarmflang.so	X	X
libomp.so	X	X
librt.so.1	X	X
libastring_aarch64.so	X	X
libpthread.so.0	X	X
libm.so.6	X	X
libmpi.so.40	X	X
libc.so.6	X	X
libopen-rte.so.40	X	X
libopen-pal.so.40	X	X
libutil.so.1	X	X
libhwloc.so.5	X	X
libevent_core-2.0.so.5	X	X
libevent_pthreads-2.0.so.5	X	X
libstdc++.so.6	X	X
libgcc_s.so.1	X	X
/lib/ld-linux-aarch64.so.1	X	X
libnuma.so.1	X	X
libltdl.so.7	X	X

As we can see we have the same dependencies with and without openmp, it due to the fact that even if we disable openmp the program still include omp.h which mean that openmp is always a dependencies of the program.

1.

3.4 Benchmarks

Cores	Pure MPI	Pure OpenMP
16	26s	40s
8	41s	55s
4	69s	84s
2	134s	145s
1	251s	265s

Table 1 – Pure MPI and OpenMP testing on C016_04 ACFL compiled

First of all, we can see that MPI is more efficient than OpenMP for Saturn. Not only that, MPI is close to being strong scaling unlike OpenMP.

We measured these times with MPI and OpenMP enabled. We also made sure that each threads was allocated to specific cores by checking the cpu usage with **/usr/bin/time** and **htop**. What we can deduce from these recorded times is that Saturn isn't strong scaling, doubling the allocated resources does reduce the execution time but we're not close to dividing it by two. It's not weak scaling either as doubling the workload and the resources does not keep the execution time relatively the same.

Benchmarks	Dataset (s)									
	C016_04		F128_01					F128_02		F128_04
MPI_RANK:	8	4	64	32	16	8	4	32 (2 nodes)	32 (4 nodes)	32 (4 nodes)
OMP_NUM_THREADS:	1	2	1	2	4	8	16	2	2	2
Elapsed Time	15	12	430	286	322	390	443	712	523	523

Table 2 – Measured execution times with ACFL compilation

.

3.5 Forge

FIrst of all we have to download forge. Once we have done this we have to create a reverse connect by following the guideline given in the pdf for Saturn. For that we only need the address of the server and the path to the forge directory on the server.

3.5.1 DDT

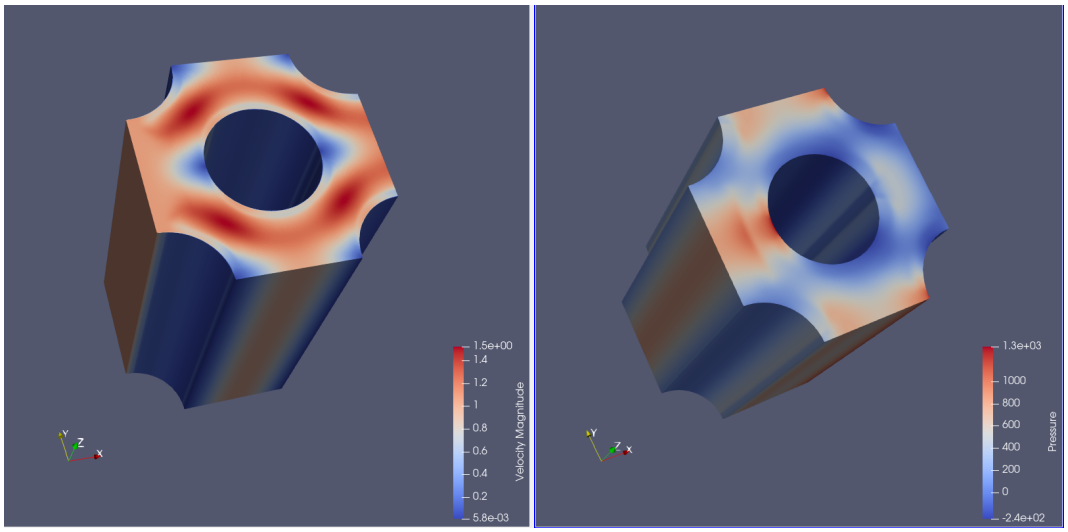
Now that we got the reverse connect, we can run the program under ddt by using the command line *ddt –connect python3 /code_saturne_7.2.0/bin/code_saturne run*. No problem occurs during the use of ddt with reverse connect. We were not able to use play, pause, Breakpoint, etc... because all the program is not compile with -g. Even we configure with the option *–enable-debug*.

3.5.2 MAP

With the reverse connect we can also profile the program with MAP using the command line *map –connect python3 /userdir/home/user/code_saturne_7.2.0/bin/code_saturne run*. No problem occurs while using map.

3.6 Visualization

To install paraview on arch linux, we typed the command *pacman -Sy paraview*. It automatically download a binary ready to use on our laptop. Once Paraview is installed we can load the result (*postprocessing/RESULTS_BOUNDARY.case*, *postprocessing/RESULTS_FLUID_DOMAIN.case*). We get the following results:

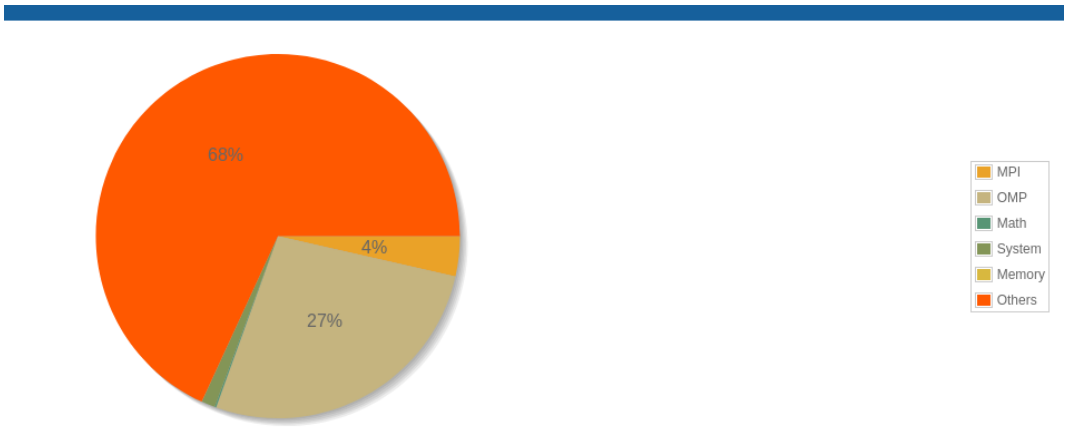


As we can see, we have the same result as those provided by EDF.

3.7 MAQAO

We used the MAQAO profiler to analyze Saturn and see where are the hot spots in the program. So I used it on C016_04 with 4 MPI process and 4 MPI threads.

Name	Module	Coverage run_0 (%)
kmp_flag_64<false, true>::wait(kmp_info*, int, void*)	libomp.so	26.83
.omp_outlined._debug__.57	libsaturne-7.2.so	4.75
.omp_outlined._debug__.76	libsaturne-7.2.so	4.19
_compute_cocgb_rhsb_lsq_v	libsaturne-7.2.so	3.85
.omp_outlined._debug__.72	libsaturne-7.2.so	3.82



MAQAO is a profiler that analyzes the binary directly of a program. It sets up breakpoints during the run time if we use the debugging compile option to see exactly how everything works deep down. The pie chart tell us that MPI and OpenMP are where most of the run time goes to and it also shows itself in the function tab where a function from the libomp library is responsible for more than 25% of the run time. It would be interesting to see how these numbers and graph evolve with different hybrid configurations, sadly we did not have enough time to see this through.