CS3D Proof-of-Concept report

Document Information

Reference Number	POP2_POCR_016
Author	Andres CHARIF RUBIAL (UVSQ)
Contributor(s)	
Date	December 21th, 2020

Notices:
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "676553".

Table of Contents

1. Background

Applicant: Stéphane ABIDE, University of Perpignan (Core developer)
Name of the code: CS3D
Scientific/technical area: CFD
Programming: FORTRAN; MPI
Input case: 20 iterations 128x128x512 mesh
Goal : hundred thousand iterations 512x512x4096 mesh
Platform: User's cluster (Skylake Xeon Gold 40 cores nodes)
Compilers: Intel ifort 19 / IntelMPI 19
Scale: 1/3/5 nodes. 40 ranks per node
UVSQ collected the performance data (Extrae / MAQAO / ScoreP – CUBE – Vampir)

The user would like to be able to launch a 512x512x4096 case

The selected configuration for this PoC is:

Platform: User's cluster (Skylake Xeon Gold 40 cores nodes)
Compilers: Intel ifort 19 / IntelMPI 19
Setup: 3 nodes. 40 ranks per node

We used MAQAO for profiling and vectorization results. MALT tool (https://github.com/memtt/malt) was used to study memory allocations.

2. Previous assessments and recommendations

Based on the performance assessment, parallel performance is limited by parallel efficiency due to transfer efficiency.
Still, the program scales perfectly.

Memory consumption is actually the real issue.

From the computation point of view, something could be done at the vectorization level.

1.1. Memory consumption

The main issue is memory usage because the user can't launch cases bigger than 256x256x512. That is why the main objective of this study is to understand why and try to fix it. Getting a performance enhancement is nice but not useful if not being able to launch the desired target case (512x512x4096).

We saw (Figure 1) that the memory allocation tracing counts 3,1GB requested per Rank. Total memory consumption was 127GB (182 – 55) / 40 = 3,1GB. 40 being the total number of ranks.

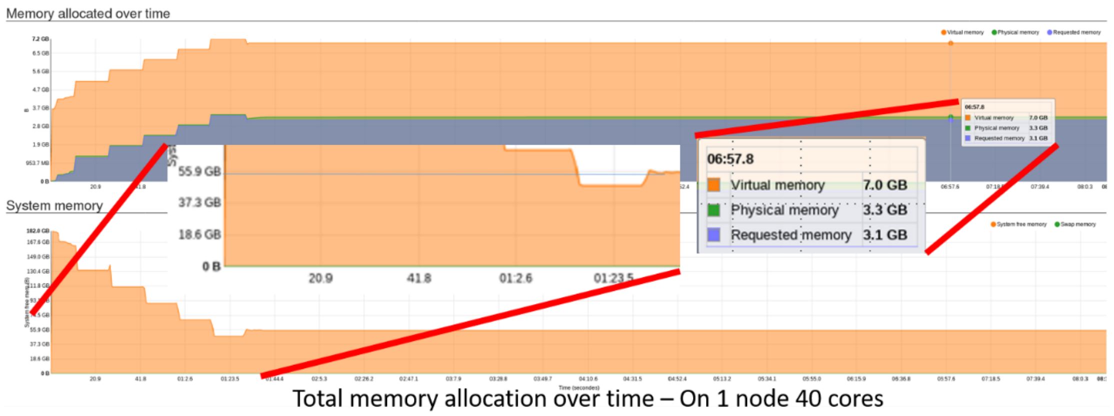


Figure 1: Memory consumption using the old version - 40 cores 128x128x512 use case

Memory consumption seems high given the size of the problem (128² x 512 x 8 = 67MB). We could expect a factor of 10. The user explains that there are many data structures which lead to a factor of around 40 (no link with the number of ranks).

However, user confirms that the total consumed memory is bigger than expected.

Figure 2 shows the portion of code which allocates the main data structures.

```
  allocate(   this%px(ni,ni,nk)  ,  this%pxm1(ni,ni,nk)  ,  this%vpx(ni,nk)  ,
this%flx(ni,2,nk) , this%fx(ni,2,nk) )
  allocate(  this%py(nj,nj   ) , this%pym1(nj,nj   ) , this%vpy(nj   ) , this%fly(nj,2
) , this%fy(nj,2   ) )
  allocate(  this%vpz(nk) )
```

Figure 2 (m_solver_diag_iih_cyl.f90:329)

This code is common to all MPI ranks. We clearly see that the same data structures (arrays) are duplicated over processes. Analyzing how the data structures are further used helps understanding why the user made this choice.
There are three axes (cylindrical coordinates). On the X axis there is a square matrix coupled to a Z point resulting in a 3-dimentional array. Y is independent. Nothing prevents cutting the Z axis in subgroups.

As consequence, the memory issue (problem size that can be computed) which the application suffers from could be fixed by implementing this split along Z-Axis.

Note that fixing this issue should also provide some speedup because because many computation (diagonalization phase) can be avoided.

1.2. Vectorization opportunities

We noticed that the vectorization efficiency could be enhanced by enabling the proper flags in the compilation chain.

3. Scope of the activity

3.1 Addressed recommendations and code refactoring

Our goal is to find out why so much memory is consumed and if it is legitimate.

To achieve this goal, we had to dig into the code. This was done thanks to the user involvement. We needed to understand the way memory was allocated with respect to the algorithms.

We also verified if vectorization efficiency could be enhanced.

3.2 Use-case and evaluation metrics

We will first use a 20 iterations 128x128x512 mesh on 40 cores. Then we will try a 512x512x2048 mesh on 200 cores.

The main metric will be the size of the problem that can be launched. It corresponds to the total memory footprint.
We will also consider walltime for performance speedup evaluation.

3.3 Target system

We will use the user’s system with the same environment.

4. Implementation

We will now detail how we fixed the memory allocation issue and how we enhanced vectorization efficiency.

4.1. Memory allocation issue

As explained in the previous section we found a way to fix the memory size issue by splitting one of the data structures (Z axis).

Figure 3 shows the new allocation scheme. This idea is to consider the X axis (radial) as the main axis and then split each part of Z dimension across the available ranks. So, the chunks are dynamically computed to fit the available MPI ranks.

```
   allocate(      self%pz_           (ph%zst(3):ph%zen(3),ph%zst(3):ph%zen(3),
sp%zst(1):sp%zen(1) ) )
   allocate             (self%pzm1_(ph%zst(3):ph%zen(3),ph%zst(3):ph%zen(3),
sp%zst(1):sp%zen(1) ) )
   allocate
(self%vpz_ (ph%zst(3):ph%zen(3), sp%zst(1):sp%zen(1) ) )
   allocate
(self%flz(ph%zst(3):ph%zen(3),2),self%fz(ph%zst(3):ph%zen(3),2))
```

Figure 3 (m_solver_diag_hii_cyl.f90:578)

As a consequence, the global memory footprint will decrease as the number of cores increase. In the previous version all the MPI ranks were requesting the total memory footprint. We then understand easily why the higher use cases quickly saturated the available memory.
Now not only we can split the total memory footprint but each MPI rank only allocates memory for one part of the computations.

```
   do i = sp%zst(1) , sp%zen(1)

     if (i.ne.sp%xen(1)) then

       tmp = -nu*d2
       do j=zstart(3),zend(3)
         tmp(j,j) = tmp(j,j) + real(self%wave(i))*dg_rm1(j,j)**2
       end do
       if (present(opt)) then
         if (opt) then
           do j=zstart(3),zend(3)
             tmp(j,j) = tmp(j,j) - dg_rm1(j,j)**2*(-nu)
           end do
         end if
       end if
       call reduction_not_periodic( staggered, zsize(3), cs_gz(3), tmp, b1, bn,
p, pm1, vpr,vpi, fl, f, cl )
       self%pz_(:,:,i) = cmplx(p(:,:),0.0)
       self%pzm1_(:,:,i) = cmplx(pm1(:,:),0.0)
       self%vpz_(:,i) = cmplx(vpr,vpi)
       !> comment flz ne varie pas avec k
       self%flz = fl
       self%fz = f
       self%clz = cl
       deallocate(p, pm1, vpr,vpi, fl, f)
     end if
   end do
```

Figure 4 (m_solver_diag_hii_cyl.f90:584)

Figure 4 depicts the rest of the processing in which we iterate over the Z portion for the current MPI rank and deal with its coupled matrix on the X axis.

From the computation performance perspective, this means that we have as many Z-groups times less computations (matrix diagonalization) to perform in this initialization phase. Whereas before, each rank had to perform the same computations.

Figure 5 exhibits the memory allocation profile obtained with the new version. The memory footprint is 38GB (182 – 144) / 40 = 950MB.

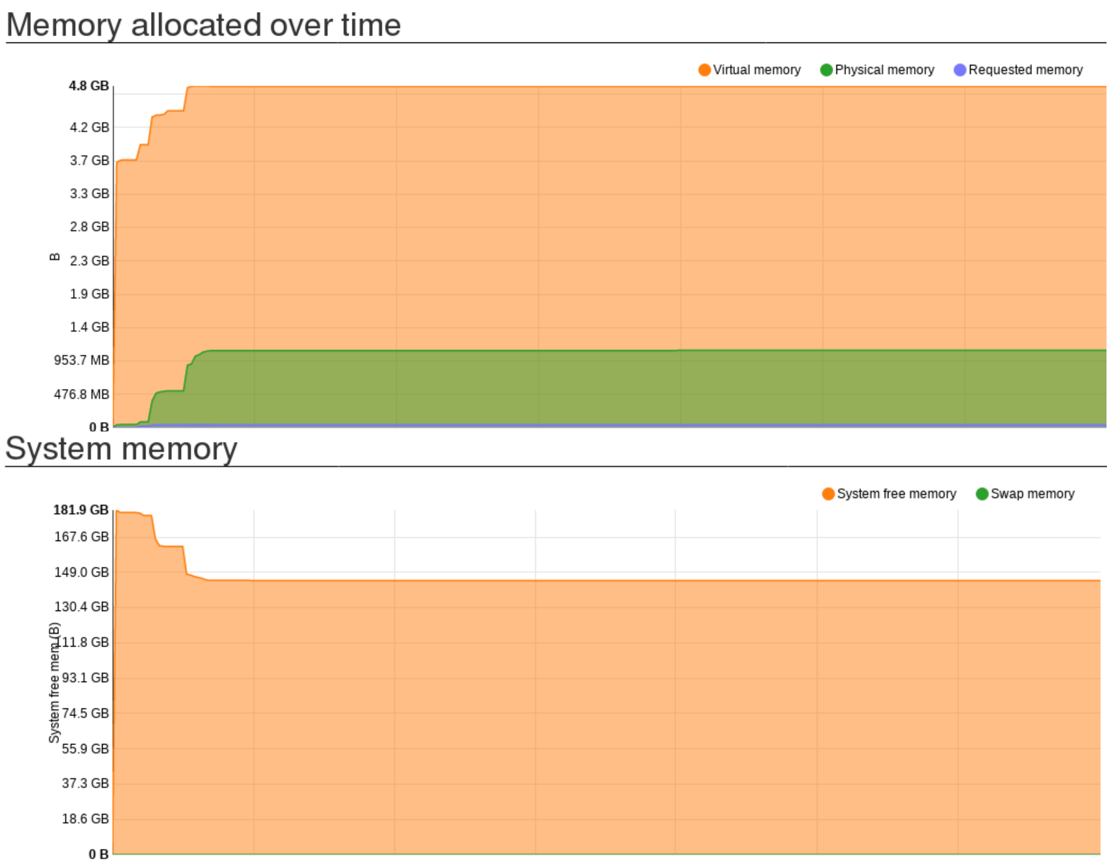


Figure 5: Memory consumption using the new version - 40 cores. 128x128x512 use case
Cores are mapped in 8x 5y 1z fashion.
Memory is spread across 8 groups (x axis).

The memory footprint is drastically reduced.

Figure 6 shows a much bigger use case, 512x512x2048.

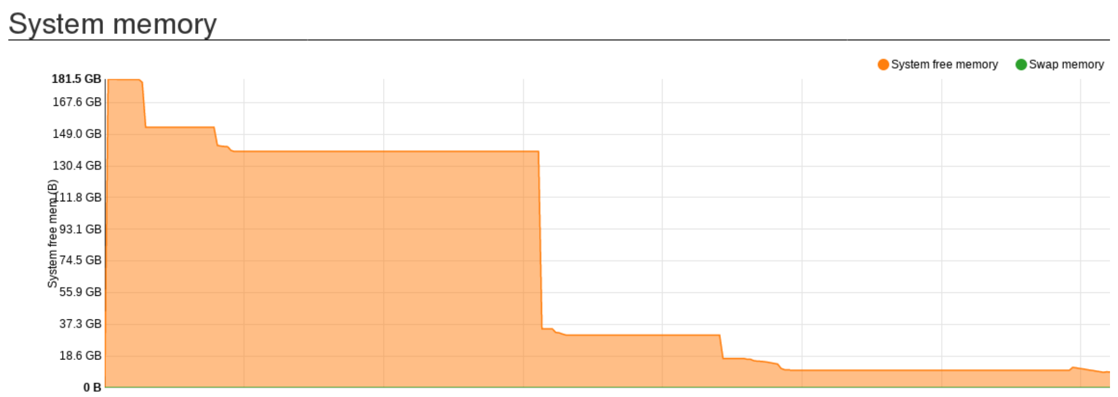
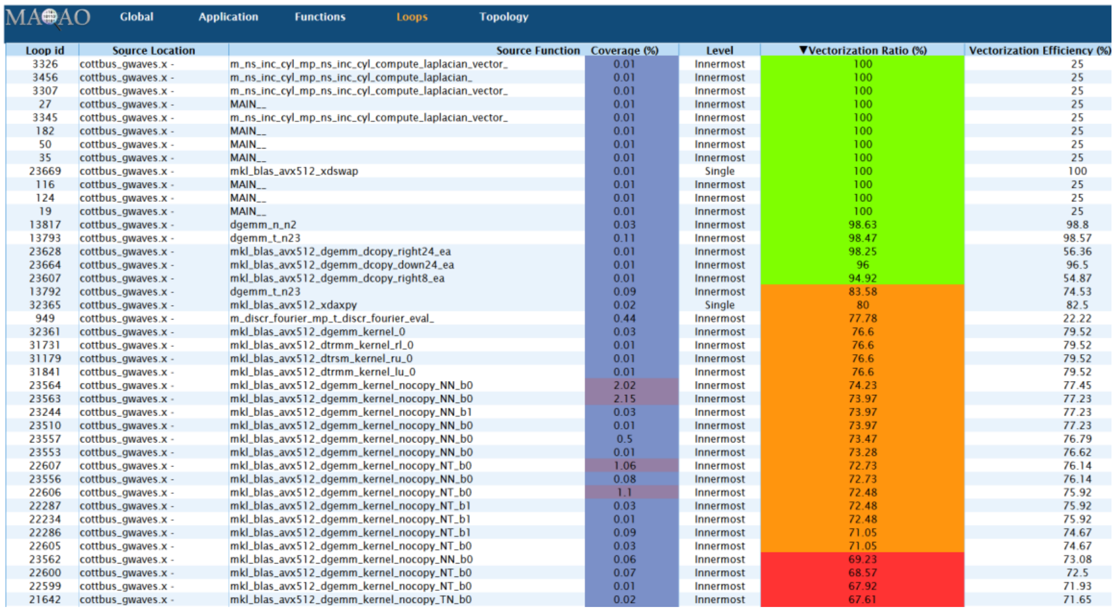


**Figure 6: Memory consumption using the new version – 200 cores. 512x512x2048 use case
Cores are mapped in 20x 10y 1z fashion.
Memory is spread across 20 groups (x axis).**

The user was able to run the target 512x512x4096 on 8 nodes. We did not include here profiling information because the user performed a production run lasting many hours.

4.2. Vectorization opportunities

Many loops in the user’s code were vectorized but no fully efficient as shown in Figure 7. While vectorized (100% ratio), the efficiency is only 25%.



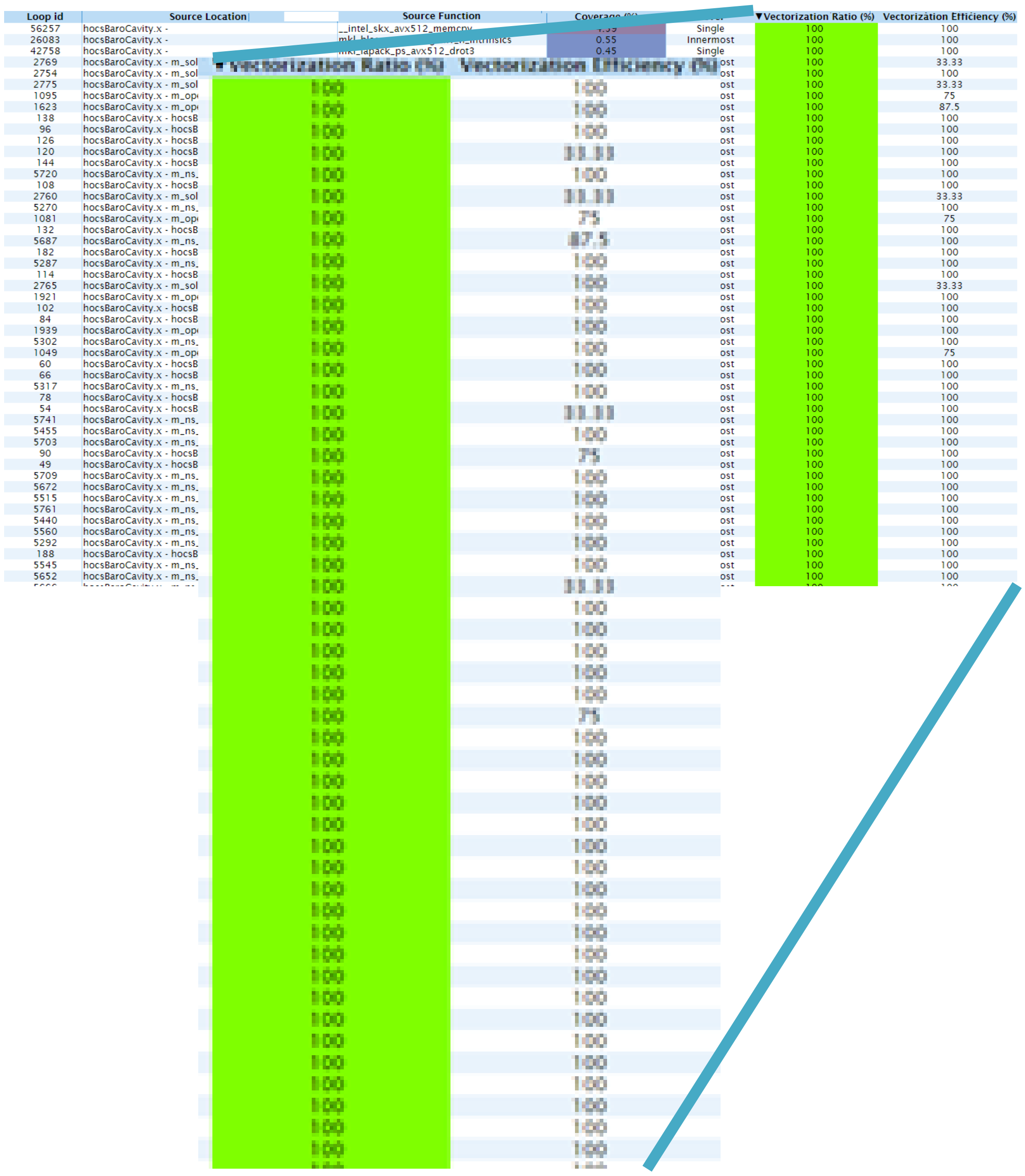
Loop id	Source Location	Source Function	Coverage (%)	Level	▼Vectorization Ratio (%)	Vectorization Efficiency (%)
3326	cottbus_gwaves.x -	m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_	0.01	Innermost	100	25
3456	cottbus_gwaves.x -	m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_	0.01	Innermost	100	25
3307	cottbus_gwaves.x -	m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_	0.01	Innermost	100	25
27	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
3345	cottbus_gwaves.x -	m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_	0.01	Innermost	100	25
182	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
50	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
35	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
23669	cottbus_gwaves.x -	mkl_blas_avx512_xdswap	0.01	Single	100	100
116	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
124	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
19	cottbus_gwaves.x -	MAIN__	0.01	Innermost	100	25
13817	cottbus_gwaves.x -	dgemm_n_n2	0.03	Innermost	98.63	98.8
13793	cottbus_gwaves.x -	dgemm_t_n23	0.11	Innermost	98.47	98.57
23628	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_dcopy_right24_ea	0.01	Innermost	98.25	56.36
23664	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_dcopy_down24_ea	0.01	Innermost	96	96.5
23607	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_dcopy_right8_ea	0.01	Innermost	94.92	54.87
13792	cottbus_gwaves.x -	dgemm_t_n23	0.09	Innermost	83.58	74.53
32365	cottbus_gwaves.x -	mkl_blas_avx512_xdaxpy	0.02	Single	80	82.5
949	cottbus_gwaves.x -	m_discr_fourier_mp_t_discr_fourier_eval_	0.44	Innermost	77.78	22.22
32361	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_0	0.03	Innermost	76.6	79.52
31731	cottbus_gwaves.x -	mkl_blas_avx512_dtrmm_kernel_rl_0	0.01	Innermost	76.6	79.52
31179	cottbus_gwaves.x -	mkl_blas_avx512_dtrsm_kernel_ru_0	0.01	Innermost	76.6	79.52
31841	cottbus_gwaves.x -	mkl_blas_avx512_dtrmm_kernel_lu_0	0.01	Innermost	76.6	79.52
23564	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	2.02	Innermost	74.23	77.45
23563	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	2.15	Innermost	73.97	77.23
23244	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b1	0.03	Innermost	73.97	77.23
23510	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	0.01	Innermost	73.97	77.23
23557	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	0.5	Innermost	73.47	76.79
23553	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	0.01	Innermost	73.28	76.62
22607	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0	1.06	Innermost	72.73	76.14
23556	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	0.08	Innermost	72.73	76.14
22606	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0	1.1	Innermost	72.48	75.92
22287	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1	0.03	Innermost	72.48	75.92
22234	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1	0.01	Innermost	72.48	75.92
22286	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1	0.09	Innermost	71.05	74.67
22605	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0	0.03	Innermost	71.05	74.67
23562	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	0.06	Innermost	69.23	73.08
22600	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0	0.07	Innermost	68.57	72.5
22599	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0	0.01	Innermost	67.92	71.93
21642	cottbus_gwaves.x -	mkl_blas_avx512_dgemm_kernel_nocopy_TN_b0	0.02	Innermost	67.61	71.65

Figure 7: Loop profile using MAQAO – Vectorization metrics before fixing vectorization efficiency

We have to tell the compiler to force full width vectorization using the following flags (ICC in out case).

GCC: -march=skylake-avx512 -mprefer-vector-width=512
ICC : -xCORE-AVX512 -qopt-zmm-usage=high

We can see in figure 8 that vectorization is harnessing the full available vector width in most of the cases now.



Loop id	Source Location	Source Function	Coverage (%)		▼Vectorization Ratio (%)	Vectorization Efficiency (%)
56257	hocsBaroCavity.x -	__intel_skx_avx512_memcpy	4.39	Single	100	100
26083	hocsBaroCavity.x -	mkl_bl[illegible]_intrinsics	0.55	Innermost	100	100
42758	hocsBaroCavity.x -	mkl_lapack_ps_avx512_drot3	0.45	Single	100	100
2769	hocsBaroCavity.x - m_sol			ost	100	33.33
2754	hocsBaroCavity.x - m_sol			ost	100	100
2775	hocsBaroCavity.x - m_sol			ost	100	33.33
1095	hocsBaroCavity.x - m_ope			ost	100	75
1623	hocsBaroCavity.x - m_ope			ost	100	87.5
138	hocsBaroCavity.x - hocsB			ost	100	100
96	hocsBaroCavity.x - hocsB			ost	100	100
126	hocsBaroCavity.x - hocsB			ost	100	100
120	hocsBaroCavity.x - hocsB			ost	100	100
144	hocsBaroCavity.x - hocsB			ost	100	100
5720	hocsBaroCavity.x - m_ns_			ost	100	100
108	hocsBaroCavity.x - hocsB			ost	100	100
2760	hocsBaroCavity.x - m_sol			ost	100	33.33
5270	hocsBaroCavity.x - m_ns_			ost	100	100
1081	hocsBaroCavity.x - m_ope			ost	100	75
132	hocsBaroCavity.x - hocsB			ost	100	100
5687	hocsBaroCavity.x - m_ns_			ost	100	100
182	hocsBaroCavity.x - hocsB			ost	100	100
5287	hocsBaroCavity.x - m_ns_			ost	100	100
114	hocsBaroCavity.x - hocsB			ost	100	100
2765	hocsBaroCavity.x - m_sol			ost	100	33.33
1921	hocsBaroCavity.x - m_ope			ost	100	100
102	hocsBaroCavity.x - hocsB			ost	100	100
84	hocsBaroCavity.x - hocsB			ost	100	100
1939	hocsBaroCavity.x - m_ope			ost	100	100
5302	hocsBaroCavity.x - m_ns_			ost	100	100
1049	hocsBaroCavity.x - m_ope			ost	100	75
60	hocsBaroCavity.x - hocsB			ost	100	100
66	hocsBaroCavity.x - hocsB			ost	100	100
5317	hocsBaroCavity.x - m_ns_			ost	100	100
78	hocsBaroCavity.x - hocsB			ost	100	100
54	hocsBaroCavity.x - hocsB			ost	100	100
5741	hocsBaroCavity.x - m_ns_			ost	100	100
5455	hocsBaroCavity.x - m_ns_			ost	100	100
5703	hocsBaroCavity.x - m_ns_			ost	100	100
90	hocsBaroCavity.x - hocsB			ost	100	100
49	hocsBaroCavity.x - hocsB			ost	100	100
5709	hocsBaroCavity.x - m_ns_			ost	100	100
5672	hocsBaroCavity.x - m_ns_			ost	100	100
5515	hocsBaroCavity.x - m_ns_			ost	100	100
5761	hocsBaroCavity.x - m_ns_			ost	100	100
5440	hocsBaroCavity.x - m_ns_			ost	100	100
5560	hocsBaroCavity.x - m_ns_			ost	100	100
5292	hocsBaroCavity.x - m_ns_			ost	100	100
188	hocsBaroCavity.x - hocsB			ost	100	100
5545	hocsBaroCavity.x - m_ns_			ost	100	100
5652	hocsBaroCavity.x - m_ns_			ost	100	100

Figure 8: Loop profile using MAQAO – Vectorization metrics after fixing vectorization width (512 bits)

5. Results

After having fixed the memory allocation issue it is now possible to launch much bigger case.

The problem we found not only forbade using cases bigger than 256x256x512 but also the number of nodes that could be used. Indeed, the memory footprint could not be spread among nodes.

The user was able to launch the target 512x512x4096 case.

The solution to fix the memory allocation issue also provided a significant performance gain of 2X on the 128x128x512 case we used on 1 node.

In a nutshell, not only the memory footprint can be spread among processes but computations within this phase also.

We also succeeded in enabling the highest vectorization efficiency but it did not provide significant performance enhancement. Most of the time spent in computation is relate to the MKL library which does automatically detect the processor’s capabilities. The time spent in the user’s loops is actually negligible when compared to the MKL. So even if the user’s loops vectorization efficiency was enhanced there is no significant performance gain overall.

6. Conclusions

We found a problem with respect to memory allocation and fixed it. The user is now able to launch the target test case he needed. This is a major breakthrough for the user.

Fixing the memory allocation issue also leveraged a 2X speedup on the 128x128x512 case we used on 1 node.

Enhancing vectorization efficiency did not provide significant performance gain because the MKL, which is intensively used, automatically detects the architecture capabilities and use them (already efficient).