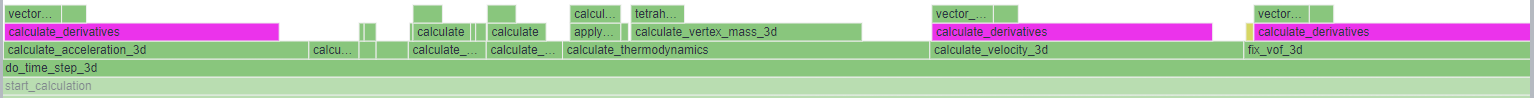
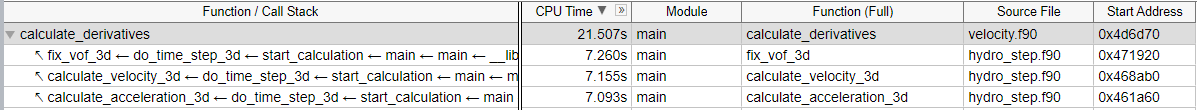
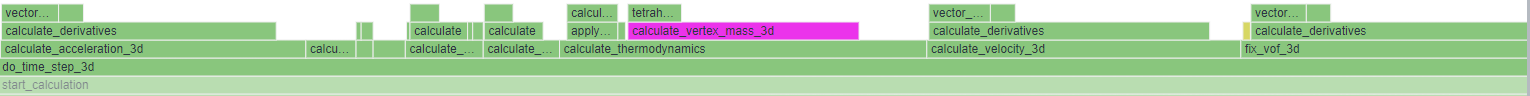
# cs236606 Final Project Report – Adding Open MP to ScalSale

1. Serial execution profiling to detect the hotspots:
   1. Since the code base is vast, I used Intel VTune to detect the parts of the code where the serial execution spends most of the time.  
      Compilation with ifort compiler (equivalent version with gnu compiler exists too), using the default compiler knobs – emphasize on -O2 knob – (provided in CMakeLists.txt), yields total execution time of 50.50[s] on Intel Xeon Gold 6128 with 24 logical cores: Text

      Description automatically generated  
      Using GNU compiler with the default knobs from CMakeLists.txt (-O0), not-so-surprisingly I get worse single-thread performance: Text

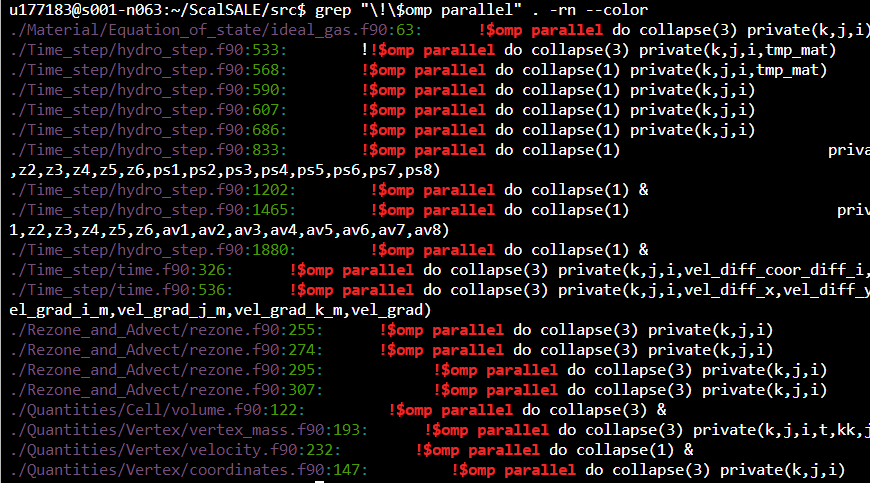
      Description automatically generated  
      I think this is because we work on Intel HW and an intel compiler is best suited for performance on that HW. Also of course that ifort with -O2 will be better than any other compiler with -O0.
   2. As expected, the serial code run on a single core and therefore causes very low core utilization:  
      Text

      Description automatically generated with low confidence
   3. This analysis revealed where we should focus our efforts. The answer is the function *calculate\_derivatives* inside the file *velocity.f90*. Graphical user interface, table

      Description automatically generated  
      roughly 35% of the execution time spent inside this function. It is being called 3   
      times during execution, as showed here: Second in its contribution to the total execution time is the function *calculate\_vertex\_mass\_3d* in the file *vertex\_mass.f90*: 
2. A little deeper look reveals some interesting phenomena which I haven’t seen before. Not sure what can I do with it in the context of this project, but it is worth mentioning (feel free to skip😊). Using the HPC Performance VTune report, we can see the effects of ‘first touch’ an array: We can see that the 1st access takes an order of magnitude longer than the rest. I think it is because the need to do TLB lookup. Given large enough pages, it happens once per loop iteration per array.  
   Then, each consecutive access has the shortest time, since the data is probably in the L0$ already. for example, line 236.  
   But when we access another array item farther than the current cache line, it takes longer. For example, accessing the next y iteration (jp, at line 237). I speculate that the next y iteration is not in L0$, but does exist already in L1$ maybe, because it is still shorter than accessing the next k iteration, like in line 239.
3. A few words about the methodology:
   1. I worked iteratively to parallelize all ‘low hanging fruits’, aka, serial long and ugly loops.
   2. For that I used VTune HPC-Performance report, especially its section: 
   3. I was done when all top serial hotspots were ‘system-originated things’ (such as page faults, Open MP forks, etc.)
   4. For example (after I squeezed every milligram of parallelism out of it):  
      Graphical user interface

      Description automatically generated with medium confidence
   5. For every serial hotspot, I tried one vanilla OpenMP work-sharing construct, and multiple more complex pragma-set, depend on the scenario.  
      For example, different *collapse* values, re-ordering the loops for better memory access pattern, merging consecutive loops into a single loop to save thread-spawning overhead, SIMD, scheduler clauses, etc.  
      In a few cases where max-reduction presented (for example: time.f90, *Calculate\_vel\_grad\_dt\_3d* subroutine), I added critical section pragmas to ensure memory coherency.  
      I didn’t use reduction pragma nor atomic construct because as part of the reduction there are others global variables which had to be updated, so I decided to encapsulate them all together using a single critical construct.
   6. I also ‘shmooed’ for different thread counts. There is a separated branch for each version.  
      Non-surprisingly, I have found that the optimal number of threads for Xeon Gold 6128 with 2 sockets, 12 physical cores is 12. With the default thread affinity in the system, it utilizes all physical cores, while allowing all threads to avoid races on each core resources.
4. SIMD:   
   I tried different combinations of SIMD compiler knobs (***-xHost***, ***-xCORE-AVX512***), with/without ***-qopt-zmm-usage=high*** and ***-O3***/***-O2***.  
   I split the evaluation to two sections, without SIMD (next bullet), and with SIMD. Each set of experiments is in a different branch, to make things ordered and in place.
5. OpenMP without SIMD:  
   the implementation is in branch omp.  
   I use compilation with ***-O3*** and ***-ipo*** (inter-procedural optimization).  
   you can find the files in the directory: ***ScalSALE/src/Scripts/ifort\_parallel/o3/no\_simd/march18***

I get total execution time of 10.25[s], that is ~1.01[s/iteration].  
Given the serial time of ~50.5[s] (see section 1), it leads to 4.9x speedup.  
I wrote a script to compare the results and make sure they are valid. Its path is: ***/home/u177183/ScalSALE/src/Scripts/validate\_result.ipynb*** and you can use it yourself to make sure the results are the same as for the serial version. Just make sure to change to paths inside the Jupyter Notebook.  
Text

Description automatically generated  
These are all the places I added parallelism:  


1. OpenMP with SIMD:  
   implementation in omp-simd branch.

When I enabled all SIMD OpenMP pragmas (wherever it makes sense – in loops with too much branches which I didn’t manage to eliminate, I left the regular thread-parallelism as is), I get I found that there is no significant better speedup compared to OpenMP parallelism:  
Text

Description automatically generated  
Although the vectorization did happen pretty well:  
Table

Description automatically generated with medium confidence  
I suspect that it is because of a lot of scattered memory accesses I didn’t manage to eliminate by changing the memory layout of the arrays. By That I mean for example to the read of x(i,j,k), x(i+1,j,k), x(i,j+1,k), x(i+1,j+1,k), x(i,j,k+1), etc. All are literals of the vectorized subroutine. It makes the memory access very inefficient and leads to bad cache behavior in the low levels.

1. OpenMP GPU offloading:
   1. First thing to notice is that the program is build from one main loop section which is sequential in terms of each iteration is dependent on the previous (the loop in problem.f90). therefore, this loop can’t be offloaded easily to GPU.
   2. Second, I looked for one function inside the main loop to offload to GPU. From the same reasons as for the CPU case, I decided to offload the longest (in terms of execution time) function that is Calculate\_derivatives (in velocity.f90).
   3. The problem arises is that Calculate\_derivatives produces data structures used by other function which I didn’t offloaded to GPU (for the sake of simplicity). And equivalently, uses data structures that is being updated in each iteration by other function in CPU memory.
   4. One obvious solution for that can be to offload the entire program to GPU. But it will take too much time, and we can’t even run it in this exercise.
   5. So, there are two options:
      1. One, to map the relevant data structures on each iteration from CPU memory to GPU memory and vice versa. This option is implemented in branch omp-gpu
      2. And two, to map the relevant data structures once initially before the main loop, and then only update the relevant data structures on each iteration. Read-only arrays from CPU to GPU **before** Calculate\_derivatives, and written arrays from GPU to CPU **after**. This option is implemented in branch omp-gpu-v2.
   6. Option 2 can be efficient only if the system can do lazy-copy (or copy-on-dirty), and, if the allocation of memory takes some time regardless the fulfillment itself. In such case, it will be worth something to allocate the buffers once before the main loop, and only update parts of it on each iteration.