Project Report

236606

Shared-Memory Parallelism: CPUs, GPUs and in-between

Name: Hussein Abu Jabal

ID: 318585577

Email: Hussein.ab@campus.technion.ac.il

This document and the relevant code can be found in the GitHub repository:
https://github.com/AbuJabal-Hussein/ScalSALE

Table of Contents

Compile and Run:

Follow these steps to compile and run the project code:

Requirements:

We require using the following to compile and run the project code:

1- Environment: Intel DevCloud.
2- Compiler: Intel ***ifort*** compiler.
3- Processor: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

Compilation:

1- Login to the Intel DevCloud and open a Terminal.
2- Download or clone the project code from the GitHub repository https://github.com/AbuJabal-Hussein/ScalSALE
3- Setup the environment parameters by running the following commands in the command line prompt (cmd):
   a. ***export I_MPI_F90=ifort***
   b. ***source /opt/intel/inteloneapi/setvars.sh***
4- Download and compile the json-fortran library using the ***ifort*** intel compiler following the instructions in the original project paper.
5- Update the path of the json-fortran build in the ***src/CMakeLists.txt*** file in the ***Goal1*** directory.
6- Compile the project code in ***Goal1*** directory using ***ifort*** intel compiler by running the following commands from the ***Goal1*** directory:
   a. ***cd src/Scripts***
   b. ***./clean.sh***

Running:

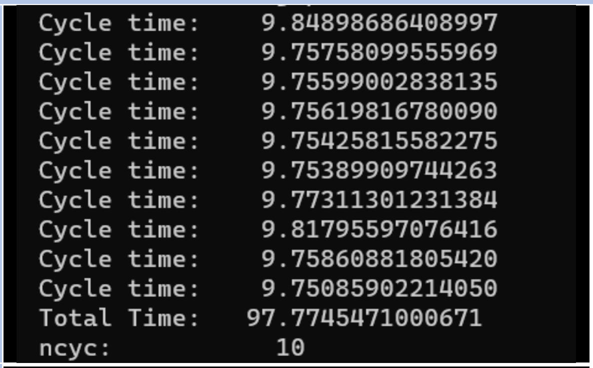
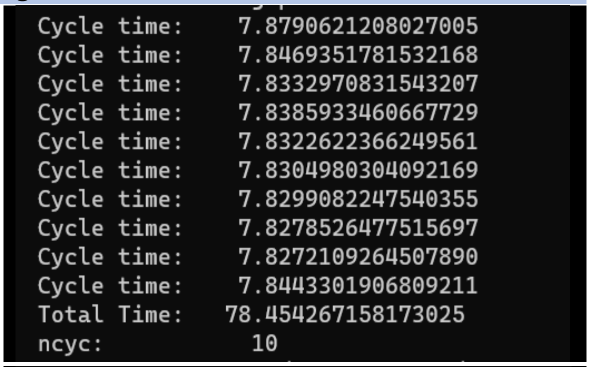
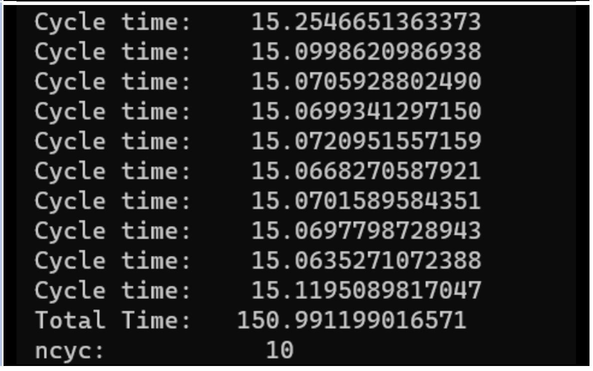
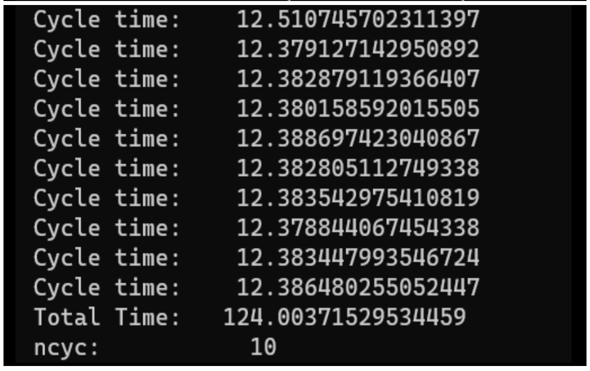
1- Reserve an "11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz" processor in the Intel DevCloud using the following command:
qsub -l nodes=1:gpu:ppn=2 -I

2- Navigate to the directory ***Goal1/src/Scripts***
3- Run the script ***run.sh*** as the following:
./run.sh 1

Explanation and Analysis:

Serial Code:

The following figures compares the serial execution time of the code without any compiler optimizations using different compilers and processors:

Processor\Compiler	Ifort	gfortran
11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz	 Cycle time: 9.84898686408997 Cycle time: 9.75758099555969 Cycle time: 9.75599002838135 Cycle time: 9.75619816780090 Cycle time: 9.75425815582275 Cycle time: 9.75389909744263 Cycle time: 9.77311301231384 Cycle time: 9.81795597076416 Cycle time: 9.75860881805420 Cycle time: 9.75085902214050 Total Time: 97.7745471000671 ncyc: 10	 Cycle time: 7.8790621208027005 Cycle time: 7.8469351781532168 Cycle time: 7.8332970831543207 Cycle time: 7.8385933460667729 Cycle time: 7.8322622366249561 Cycle time: 7.8304980304092169 Cycle time: 7.8299082247540355 Cycle time: 7.8278526477515697 Cycle time: 7.8272109264507890 Cycle time: 7.8443301906809211 Total Time: 78.454267158173025 ncyc: 10
Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz	 Cycle time: 15.2546651363373 Cycle time: 15.0998620986938 Cycle time: 15.0705928802490 Cycle time: 15.0699341297150 Cycle time: 15.0720951557159 Cycle time: 15.0668270587921 Cycle time: 15.0701589584351 Cycle time: 15.0697798728943 Cycle time: 15.0635271072388 Cycle time: 15.1195089817047 Total Time: 150.991199016571 ncyc: 10	 Cycle time: 12.510745702311397 Cycle time: 12.379127142950892 Cycle time: 12.382879119366407 Cycle time: 12.380158592015505 Cycle time: 12.388697423040867 Cycle time: 12.382805112749338 Cycle time: 12.383542975410819 Cycle time: 12.378844067454338 Cycle time: 12.383447993546724 Cycle time: 12.386480255052447 Total Time: 124.00371529534459 ncyc: 10

As we can see in the table above, the best choice for serial execution is using the ***11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz*** processor with ***gfortran*** compiler.

We also notice that the ***Intel i9*** provides a better running time for the serial execution for both compilers.

Goal #1:

For this part, I run the code on the ***11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz*** processor in DevCloud, compiled with ***ifort*** compiler, and run using 8 threads.

Profiling:

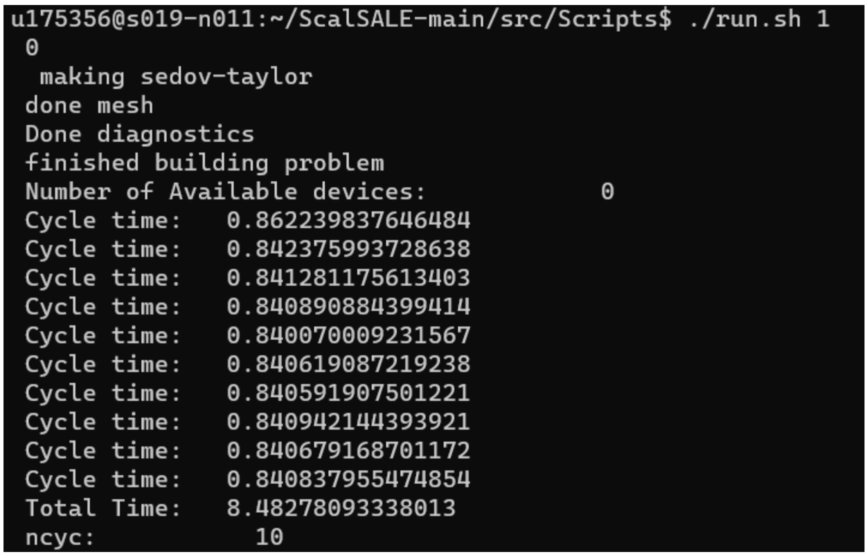
Through profiling using VTune, I learned about the hotspots of the code, and explored the heavy for-loops that take the most time to execute.

These loops are found in the following paths:

1- ***src/Quantities/Cell/volume.f90***, in subroutine "***Calculate***", line 119
2- ***src/Quantities/Vertex/velocity.f90***, in subroutine "***Calculate_derivatives***", line 226
3- ***src/Quantities/Vertex/vertex_mass.f90***, in subroutine "***Calculate_vertex_mass_3d***", line 118

Running time:

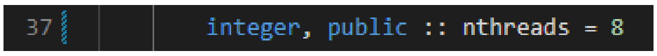
The best total time I could achieve was ***8.482 seconds*** with an average cycle time of ***0.84 seconds***, as seen in the following output print-screen:



```
u175356@s019-n011:~/ScalSALE-main/src/Scripts$ ./run.sh 1
 0
  making sedov-taylor
 done mesh
 Done diagnostics
 finished building problem
 Number of Available devices:           0
 Cycle time:   0.862239837646484
 Cycle time:   0.842375993728638
 Cycle time:   0.841281175613403
 Cycle time:   0.840890884399414
 Cycle time:   0.840070009231567
 Cycle time:   0.840619087219238
 Cycle time:   0.840591907501221
 Cycle time:   0.840942144393921
 Cycle time:   0.840679168701172
 Cycle time:   0.840837955474854
 Total Time:   8.48278093338013
 ncyc:           10
```

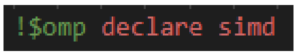
OpenMP pragmas and code changes:

1- In file ***src/Parallel/parallel_parameters.f90***, **line 37**, added the following line, which specifies the number of threads we use:



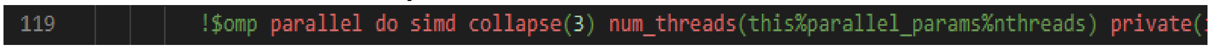
```
37          integer, public :: nthreads = 8
```

2- In file ***src/General/geometry.f90***, we specify that the subroutines in these the following line can be used as a vectorized operation: **lines 971**, **988**, **1000**, **1044**, **1096**. Using the command:



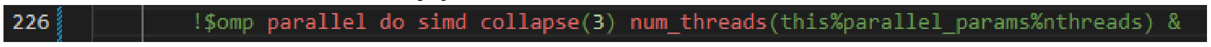
```
!$omp declare simd
```

3- In file ***src/Quantities/Cell/volume.f90***, we parallelize the loop in **line 119**



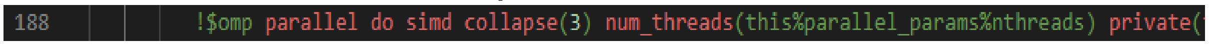
```
119          !$omp parallel do simd collapse(3) num_threads(this%parallel_params%nthreads) private(
```

4- In file ***src/Quantities/Vertex/velocity.f90***, we parallelize the loop in **line 226**



```
226          !$omp parallel do simd collapse(3) num_threads(this%parallel_params%nthreads) &
```

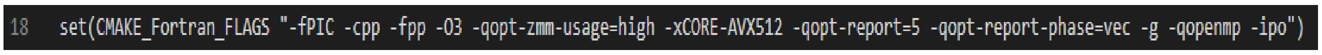
5- In file ***src/Quantities/Vertex/vertex_mass.f90***, we parallelize the loop in **line 188**



```
188          !$omp parallel do simd collapse(3) num_threads(this%parallel_params%nthreads) private(
```

Compiler and Optimizations:

After conducting some experiments, I chose to use the Intel ***ifort*** compiler (see insight #3 bellow) with the following optimizations:



```
set(CMAKE_Fortran_FLAGS "-fPIC -cpp -fpp -O3 -qopt-zmm-usage=high -xCORE-AVX512 -qopt-report=5 -qopt-report-phase=vec -g -qopenmp -ipo")
```

Insights:

1- I tried running the code on some different processors, and it turns out that using the ***11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz*** processor provides the best performance.

2- In our case, using 8 threads for the aforementioned processor, which is the number of its physical cores, provides the best performance. If you want to use the ***Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz*** processor instead, you should change the number of threads to 12.

3- I noticed that after optimizing the code with compiler optimizations and openMP pragmas, I got almost the same running time when I built the code using both compilers (for the same source code and the same machine). This could mean that both compilers performed pretty much the same optimizations on the code, which resulted in similar running time.

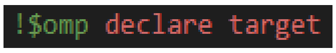
Goal #2:

In this part, we should use a machine that has a strong GPU. Therefore, it is recommended that we use a machine with an Nvidia GPU (like the NegevHPC server).

If you want to run the code on a machine that only has Intel GPU, you should add this flag to the compilation command: -fopenmp-targets=spir64

The changes to the code that I think will make the code execution more efficient are located in the directory "***Goal2/src***" (only the files that changed from Gaol1 are found there):

1- In file ***src/General/geometry.f90***, we specify that the subroutines in these the following line can be run on the target device (GPU): **lines 989**, **1002**, **1047**, **1100**. Using the command:

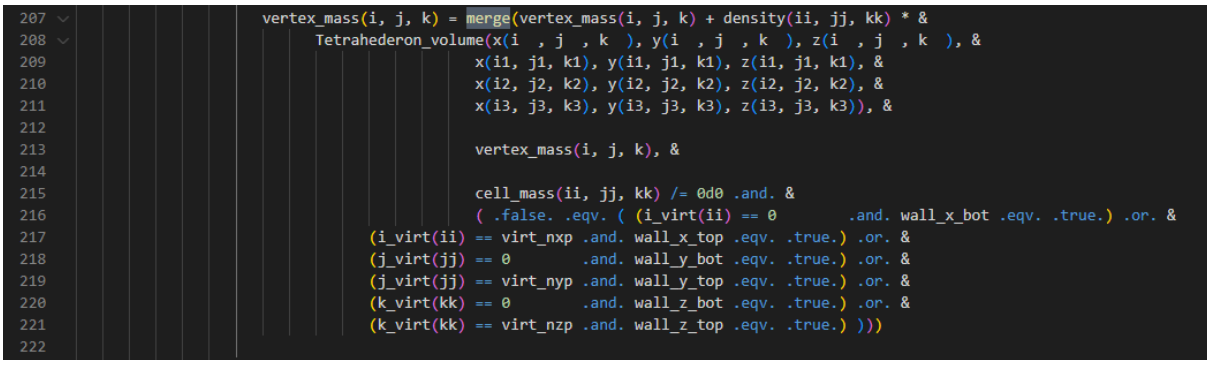


```
!$omp declare target
```

2- In file ***src/Quantities/Cell/volume.f90***, we parallelize the loop in **line 119** on the GPU by distributing the work over teams. Also, we perform the necessary data offloading before and after the loop.

3- In file ***src/Quantities/Vertex/velocity.f90***, we parallelize the loop in **line 226** on the GPU as bullet 2.

4- In file ***src/Quantities/Vertex/vertex_mass.f90***, we parallelize the loop in **line 188** on the GPU as bullet 2.
In addition, we replace the If-condition in the inner loop with a "merge" operation, which is equivalent to a Ternary conditional operation in the "C language". This change can improve the performance in this loop significantly, since the executing the If-condition on the GPU is very expensive. The change is as shown in the following print-screen:



```fortran
vertex_mass(i, j, k) = merge(vertex_mass(i, j, k) + density(ii, jj, kk) * &
      Tetrahederon_volume(x(i  , j  , k  ), y(i  , j  , k  ), z(i  , j  , k  ), &
                        x(i1, j1, k1), y(i1, j1, k1), z(i1, j1, k1), &
                        x(i2, j2, k2), y(i2, j2, k2), z(i2, j2, k2), &
                        x(i3, j3, k3), y(i3, j3, k3), z(i3, j3, k3)), &

                        vertex_mass(i, j, k), &

                        cell_mass(ii, jj, kk) /= 0d0 .and. &
                        ( .false. .eqv. ( (i_virt(ii) == 0        .and. wall_x_bot .eqv. .true.) .or. &
            (i_virt(ii) == virt_nxp .and. wall_x_top .eqv. .true.) .or. &
            (j_virt(jj) == 0        .and. wall_y_bot .eqv. .true.) .or. &
            (j_virt(jj) == virt_nyp .and. wall_y_top .eqv. .true.) .or. &
            (k_virt(kk) == 0        .and. wall_z_bot .eqv. .true.) .or. &
            (k_virt(kk) == virt_nzp .and. wall_z_top .eqv. .true.) )))
```

5- In file ***src/Main/problem.f90***, we surround the main loop in **line 750** with data offloading by mapping the necessary data to and from the GPU, using “enter data” and “exit data” directives.