Parallelization of iSALE Using OpenMP

By

Xianzheng Li

Supervisor: Gareth Collins

Department of Earth Science and Engineering

Imperial College London

Project Plan Prepared for the ACSE-9 Final Project

Rationale

iSALE (impact-SALE) is a well-established shock physics code (written in Fortran 95) that has been developed and used for over two decades to simulate impact cratering (e.g. the collision of an asteroid with a planetary surface and related processes). The code has been benchmarked against other 3D impact simulation codes and validated by comparison of simulation results with laboratory-scale impact experiments. Two and three-dimensional versions of the software exist, and the 3D version has included parallel optimization, achieved using simplistic domain decomposition and MPI.

Since the iSale has been used for more than twenty years and used in different fields, improve the performance and the efficiency of the software has become the priority. Currently, for the most challenging or higher spatial resolution simulations, the running time of the serial code can be up to a month or more, which is a significant time cost. With the rapid development of computer architectures in recent years, the processing power of the computer is getting stronger and stronger and can provide more creative ways to improve the performance of the code. Parallelization is one of the most efficient ways that comes with the development of computer architectures. As mentioned above, currently, this software has two and three-dimensional versions, but only the 3D version includes parallel optimization.

Thus, this project has two main objectives. First, the project aims to test, extend and complete iSALE2D parallelization using OpenMP, and provide an initial template which will be used in further development. Since the iSALE solution algorithm involves a series of big and complex loops, adding parallel programming to the code has the potential to reduce the run time substantially. Second, the project will explore the three different parallel scheduling methods which are dynamic, static and guided, and combine with other parameters such as chunk size to find the best combination for a specific test case, then get the optimal performance of the paralleled iSALE.

Background

In this project, I will use the iSALE shock physics code (Wunnemann et al., 2006), which is an extension of the SALE hydrocode (Amsden et al., 1980). To simulate hypervelocity impact processes in solid material, SALE was modified to include an elasto-plastic constitutive model, fragmentation models, various equations of state (EoS), and multiple materials (Melosh et al., 1992; Ivanov et al., 1997). More recent improvements include a modified strength model (Collins et al., 2004), a porosity compaction model (Wunnemann et al., 2006; Collins et al., 2011) and a dilatancy model (Collins et al., 2014).

The partial differential equations solved by the basic SALE program are the compressible Navier-Stokes equations and the mass and internal energy equations. The iSALE solution algorithm is an Arbitrary Lagrangian Eulerian finite difference method on a structured rectangular mesh. Scalar and tensor fields are piecewise constant, centred within each computational cell; vector fields (velocity and position) are centred at the four nodes that define each cell's corners. Each timestep involves an explicit Lagrangian step that updates the nodal velocities. Subsequently, the nodal positions can be updated to deform the mesh, if a Lagrangian solution is desired; alternatively, an Eulerian 'advection' step can be invoked to transfer momentum and all cell quantities between cells on the fixed mesh, based on the overlap volumes between the deformed Lagrangian mesh and the original fixed mesh. A hybrid approach is also possible, but rarely used in practice. Please find out the flow diagram in Figure 1 below.

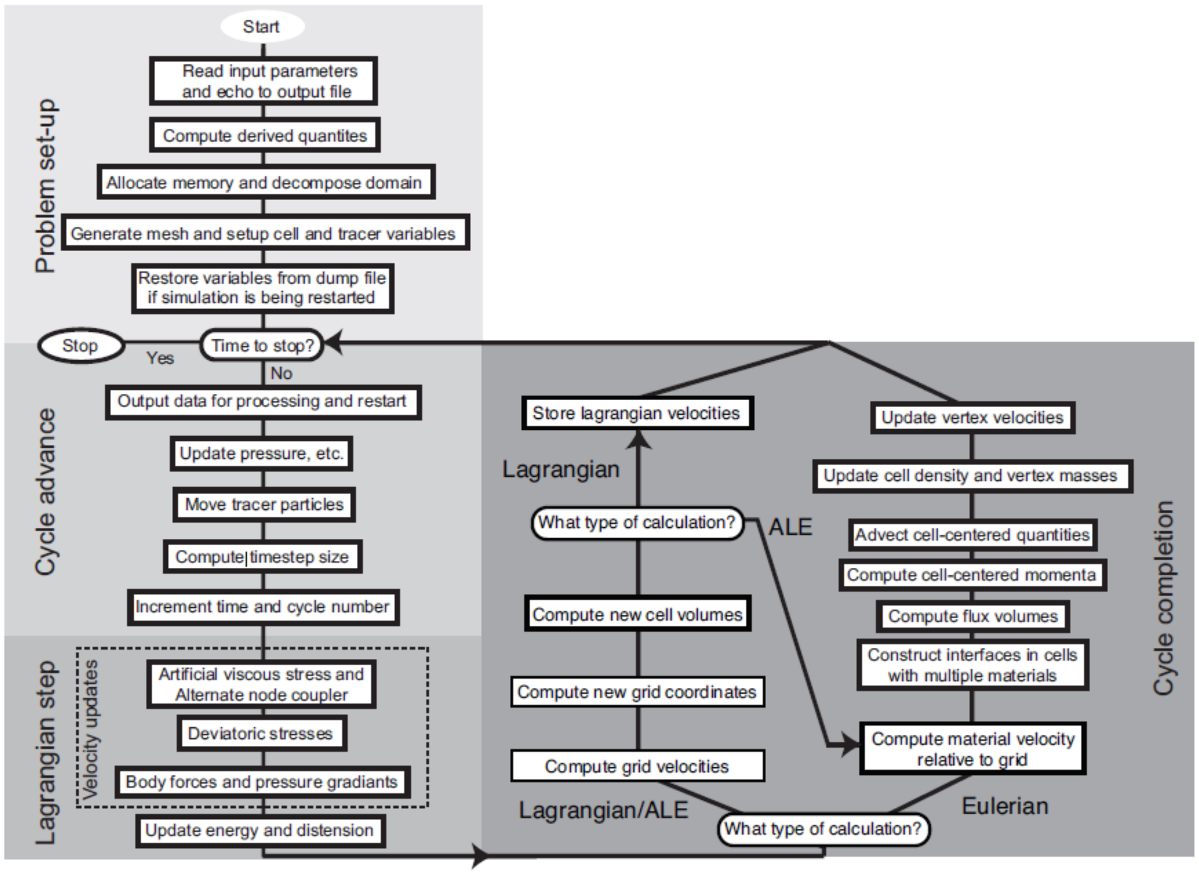

Start
Read input parameters and echo to output file
Compute derived quantites
Allocate memory and decompose domain
Generate mesh and setup cell and tracer variables
Restore variables from dump file if simulation is being restarted
Problem set-up
Stop
Yes
Time to stop?
No
Output data for processing and restart
Update pressure, etc.
Move tracer particles
Compute timestep size
Increment time and cycle number
Cycle advance
Artificial viscous stress and Alternate node coupler
Deviatoric stresses
Body forces and pressure gradiants
Velocity updates
Update energy and distension
Lagrangian step
Store lagrangian velocities
Lagrangian
What type of calculation?
ALE
Compute new cell volumes
Compute new grid coordinates
Compute grid velocities
Lagrangian/ALE
What type of calculation?
Eulerian
Update vertex velocities
Update cell density and vertex masses
Advect cell-centered quantities
Compute cell-centered momenta
Compute flux volumes
Construct interfaces in cells with multiple materials
Compute material velocity relative to grid
Cycle completion

Figure 1. Flow Diagram of iSALE2d

With developments in recent years, the iSALE program includes an array of material models which are rheologic models, porosity models, strength models, thermal softening model, damage model and dilatancy model (Collins et al., 2016), used to solve more specific situations, such as porous compaction of impact crater formation (Wunnemamm, Collins, G., and Melosh, H., 2006). The explicit time step and limitations make the program more straightforward. Furthermore, there is no external library used in the iSALE program, which reduces the software dependence on external resources.

This project will be updated on a new brunch of GitHub iSALE repository, which will be easy to check any progress and helpful for further development,

Work Plan

The work plan of this project can be separated into six main parts:

- VTune Profiling
- Code Refactoring
- First Parallel Evaluation
- Parallelization for Other Subroutines
- Evaluation of Parallel Speed Up
- Final Report

The timeline and expected deliverable for each part are listed in the Gantt chart below.

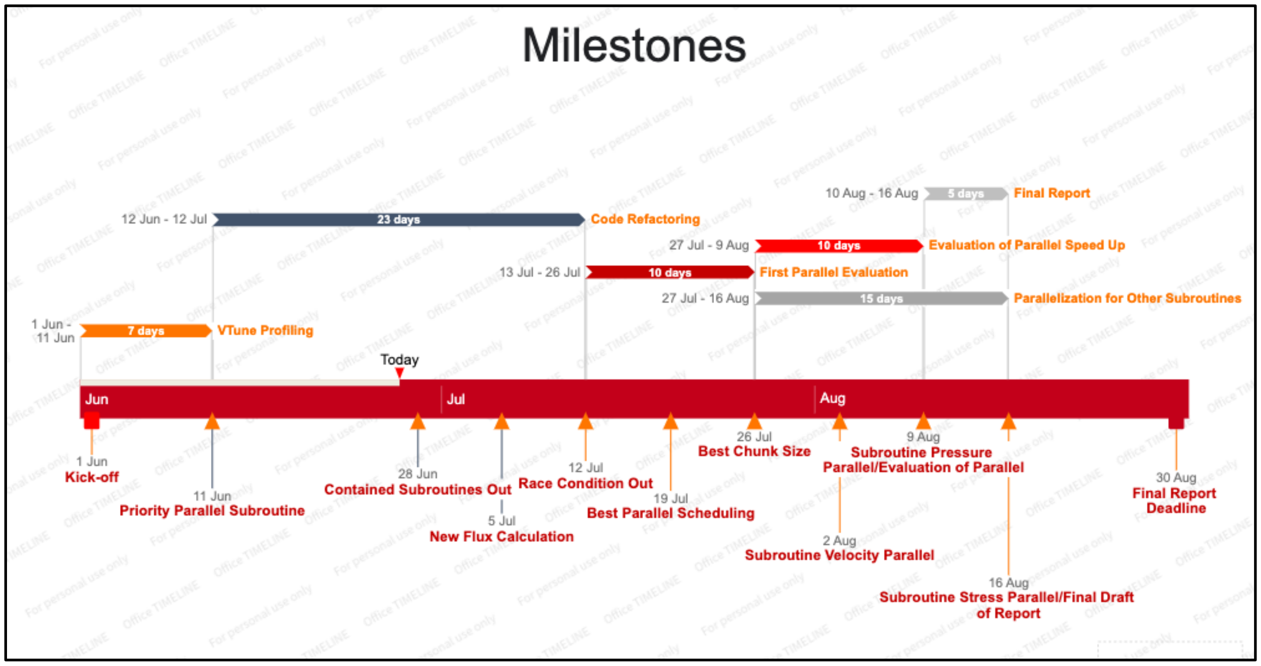


Figure 2. Gantt Chart for The Project with Expected Finish Time and Deliverable

VTune Profiling

Code profiling is the first and significant part for the project and can be used to point out a clear direction of the project. Although the main objective is to add parallelization to the code, parallel all loops or all subroutines is not an ideal way to improve the efficiency of the code. Because running processors also cost time, if a loop is not that complicated, adding parallel will have a negative effect. The VTune amplifier is a commercial application for software performance analysis. It both supports the Mac OS and Linux system and will be used to run the whole code, timing each subroutine and generate a configuration analysis report. It will provide me with a list of time cost on each subroutine, which helps me figure out which subroutine I should parallel first, and which one is the end target of parallelization.

Code Refactoring

Since I get the subroutines which cost most run time from the last section, I can consider adding parallelization to these subroutines. However, the code refactoring is required because the original structure of the code has some problems with parallel programming. The structure problems can be divided in:

1. Contained Subroutines

In the current code, there are some contained subroutines involved in a main subroutine, which will affect the efficiency of the code and is hard to add parallel. What I am going to do is to separate the subroutines outside the large one, then I can add parallel to both small and large subroutine.

2. Race Condition

Race condition will exist on the node after adding parallelization. In the current code structure, data from different cells will pass to the neighboring node, and the race condition will cause data overwriting on the node. I am going to use orders in OpenMP or rewrite the code to avoid the race condition.

3. Data Transportation in Specific Direction

The data in some loops need to be transported in a specific direction, for example, from left to right. Adding parallelization will mess up the specific orders of the data transportation. What I am going to do is to refactor the loops contain this situation to avoid any conflicts.

First Parallel Evaluation

1. Parallel Scheduling – Static, Dynamic, Guided
2. Chunk size

In this part, I will test different parameters in parallelization with different sample tests to determine the best combination for each sample test. Since there are three main parallel scheduling methods I can use in the !*$omp parallel schedule()* order, I will test each of them with different chunk size. The best scheduling method and chunk size will be varied with different sample tests. All combinations will be recorded and used in the final test.

Parallelization for Other Subroutines

After dealing with the prior subroutine, I will follow the same strategy and use the same parameters to parallel other subroutines. The main subroutines include the subroutine for advection, velocity, pressure and shear stress. One of them will become the priority of parallelization after profiling, and others will be dealt with in this part.

Evaluation of Parallel Speed Up

I will evaluate the speed up of parallelization in this part. To achieve the purpose, I will test the code by using an increasing number of threads and see if the performance is getting better. Meanwhile, since there are two different nodes available, which are CX nodes and AMD nodes, I will use both nodes to run the tests as well and record the performance of the code. Furthermore, I will record the run time of a sample test with different resolutions. I believe the parallelization will increase the efficiency of the code.

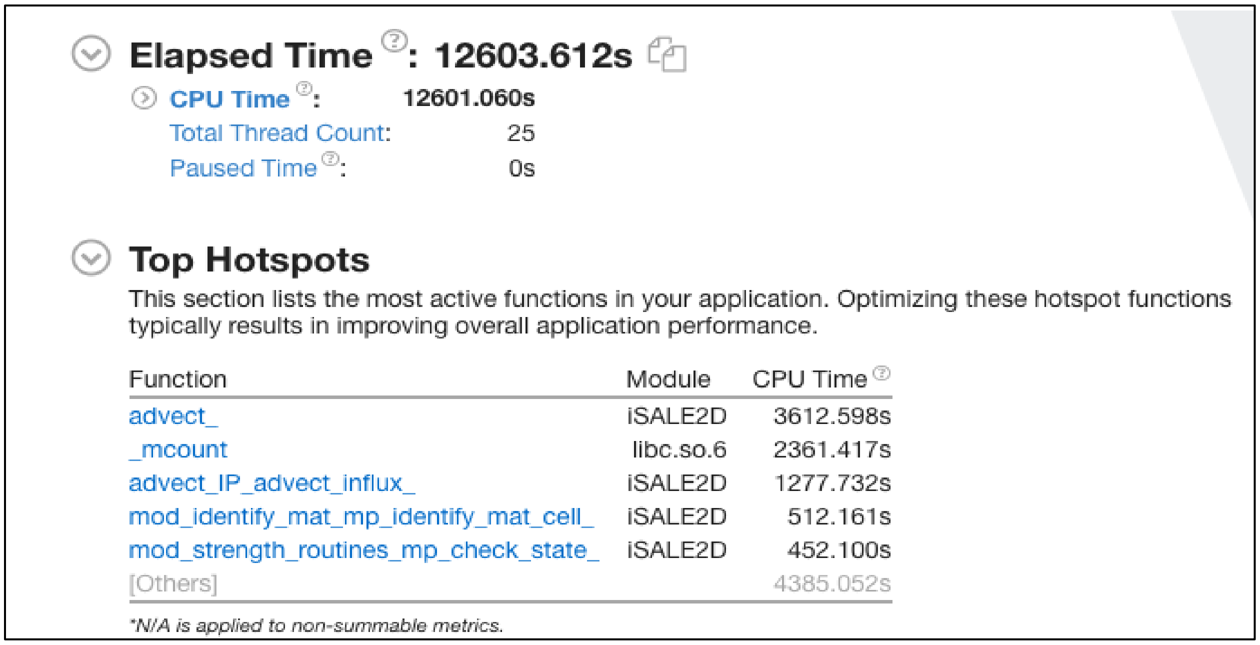
Final Report

After finishing implementing all parts above, I will work on the final report. The report will contain efforts I will make in this project, results analysis and conclusion.

Progress So Far

- Profiling

As mentioned above, the purpose of profiling is to run the test on the Intel VTune Amplifier, get the list of run time, find which subroutine costs most processing time and parallel this subroutine first. After running the most challenging sample test, I got the report below:



Elapsed Time: 12603.612s

CPU Time:	12601.060s
Total Thread Count:	25
Paused Time:	0s

Top Hotspots

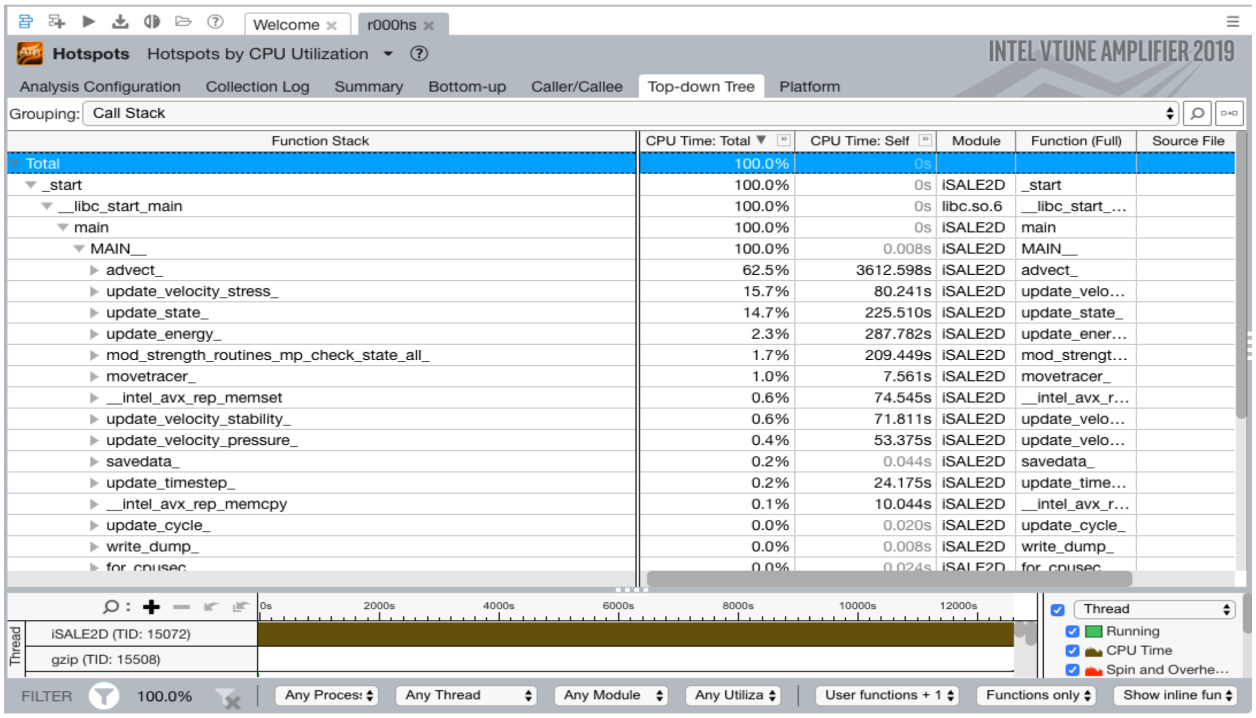
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
advect_	iSALE2D	3612.598s
_mcount	libc.so.6	2361.417s
advect_IP_advect_influx_	iSALE2D	1277.732s
mod_identify_mat_mp_identify_mat_cell_	iSALE2D	512.161s
mod_strength_routines_mp_check_state_	iSALE2D	452.100s
[Others]		4385.052s

*N/A is applied to non-summable metrics.

Figure 3. Bottlenecks and Hotspots Check

Figure 3 and 4 show the top hotspots in the software. The subroutine *advect* costs 62.5% of the total processing time. Thus, the subroutine *advect* will be the priority for parallel programming. Meanwhile, the subroutine *update_velocity_stress* and *update_state* costs 15.7% and 14.7%, total 30.4% of the run time. These three subroutines cost more than 90% of the run time. The next one is the subroutine *update_energy*, and it only cost 2.3%. Thus, I will start from subroutine *advect*, and end at the *update_state*.



Grouping: Call Stack

Function Stack	CPU Time: Total	CPU Time: Self	Module	Function (Full)	Source File
Total	100.0%	0s			
_start	100.0%	0s	iSALE2D	_start	
__libc_start_main	100.0%	0s	libc.so.6	__libc_start_...	
main	100.0%	0s	iSALE2D	main	
MAIN__	100.0%	0.008s	iSALE2D	MAIN__	
advect_	62.5%	3612.598s	iSALE2D	advect_	
update_velocity_stress_	15.7%	80.241s	iSALE2D	update_velo...	
update_state_	14.7%	225.510s	iSALE2D	update_state_	
update_energy_	2.3%	287.782s	iSALE2D	update_ener...	
mod_strength_routines_mp_check_state_all_	1.7%	209.449s	iSALE2D	mod_strengt...	
movetracer_	1.0%	7.561s	iSALE2D	movetracer_	
__intel_avx_rep_memset	0.6%	74.545s	iSALE2D	__intel_avx_r...	
update_velocity_stability_	0.6%	71.811s	iSALE2D	update_velo...	
update_velocity_pressure_	0.4%	53.375s	iSALE2D	update_velo...	
savedata_	0.2%	0.044s	iSALE2D	savedata_	
update_timestep_	0.2%	24.175s	iSALE2D	update_time...	
__intel_avx_rep_memcpy	0.1%	10.044s	iSALE2D	__intel_avx_r...	
update_cycle_	0.0%	0.020s	iSALE2D	update_cycle_	
write_dump_	0.0%	0.008s	iSALE2D	write_dump_	
for_cpusec	0.0%	0.024s	iSALE2D	for_cpusec	

Figure 4. Detailed Performance and CPU Time of Each Subroutine

- Code Refactoring – Remove Contained Subroutines

As mentioned above, there are some contained subroutines in the main subroutine, which will be hard to add parallel programming on it. I moved them out, defined all used parameters at the beginning of the subroutine and put all input parameters into the brackets in the subroutine line.

The original iSALE already has some existing framework tests for testing the correctness of the code. Thus, I used these tests to check my work. With any small changes in the code, I run the test to check if the software with the updated code could compile successfully and gave me all passed check results for each parameter. The difficulty of this procedure was the failure test did not show any error report to help me correct the code. Thus, I had to explore and run the test with even a really small change every time, which cost some time to finish this part. The part has been finished; the deliverable is the code without contained subroutines.

- Code Refactoring – Race Condition

The plot below shows that in the current code, each node will receive data from its four neighbor cells. If the code is serial, the data will be passed to the node by following the orders of increasing i and j. However, after adding parallelization, data in the cells will be processed on multiple processors simultaneously, and send to the node at the same time, which means the information from the cell (i, j+1) will overwrite the information from the cell(i, j).

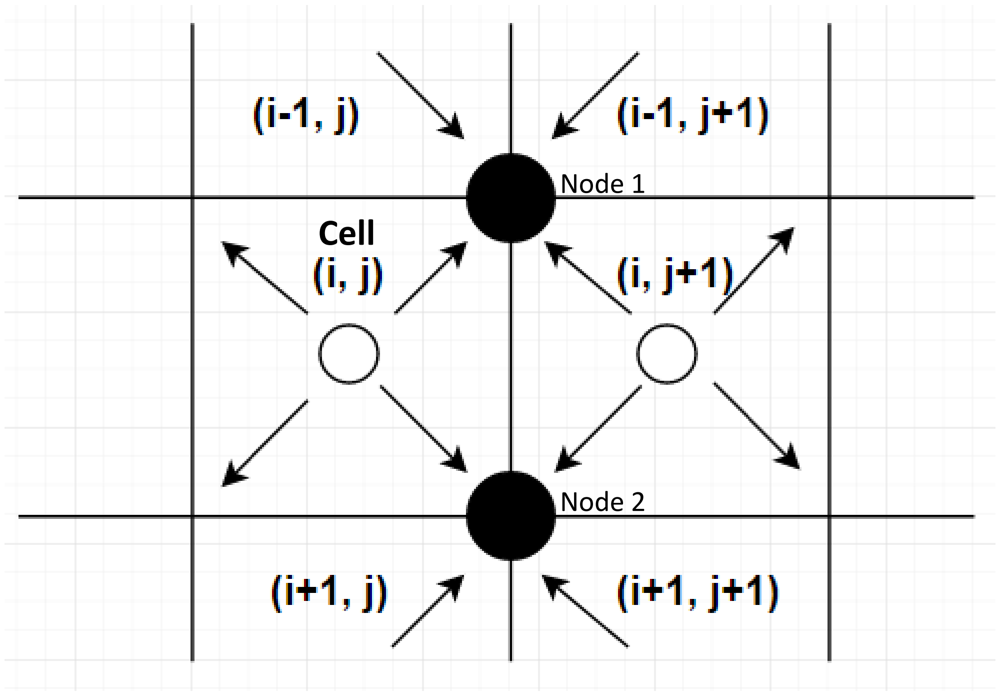


Figure 5. Sample Example of Mesh Grid and Race Condition

I will add the !*$omp parallel reduction* order in the loop to solve this problem. This order allows multiple processors send and receive data sequentially, each processor saves a copy of all updated parameters, and at the end of the order, use a specific symbol to connect the original parameters and the updated parameters, and finally update the shared parameters to remove this potential conflict. Another way is to rewrite the loop. Each cell will be updated by receiving information from its four neighbor nodes. I will try the first method to evaluate the performance of the parallelization. If the results are not correct or the efficiency does not increase apparently, I will try to use the second method. Since the method has been found, this part will be finished shortly.

- Code Refactoring – Data Transportation in Specific Direction

The original data is in a 2D frame, to simply explain the problem, I will use a 1D frame here. The problem can be explained by using the following pseudo-code:

```
do i=1, nx
     if(1.eq.1)
         fr = 0
     volflux_l(i) = fr
     volflux_r(i) = calculate_right_flux()
     fr = volflux_r(i)
end do
```

The pseudo-code shows that in each iteration, the volume flux on the left side equals to the number of *fr*, then the volume flux on the right will be updated with the number from the function *calculate_right_flux*, and finally, update the *fr* equaling to the volume flux on the right side. The structure shows the data was passed from the left to the right. However, if I add the parallelization to this, each processor will process one iteration, and I cannot promise the data can be passed in the specific order because the processing time of each processor will be varied.

To solve this problem, I will update the *fr* value before update the volume flux on both the left and right side. The method I will use is explained by using the following pseudo-code:

```
do i=1, nx
    fr(i) = calculate_right_flux(i)
end do
do i=1, nx
    volflux_l(i) = fr(i-1)
    volflux_r(i) = fr(i)
end do
```

The new code can be paralleled easily. I can add a parallel section in the first loop without changing anything in the second loop. Since the function *calculate_right_flux* is the most expensive part, this method will reduce the run time significantly.

Reference

[1] Stephen J. Chapman. 2003. Fortran 90/95 for Scientists and Engineers (2 ed.). McGraw-Hill, Inc., New York, NY, USA.

[2] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. 5, 1 (January 1998), 46-55. DOI: https://doi.org/10.1109/99.660313

[3] Amsden, A., Ruppel, H., and Hirt, C. (1980). SALE: A simplified ALE computer program for fluid
flow at all speeds. Los Alamos National Laboratories Report, LA-8095:101p. Los Alamos, New Mexico: LANL.

[4] Wünnemann, K., Collins, G., and Melosh, H. (2006). A strain-based porosity model for use in
hydrocode simulations of impacts and implications for transient crater growth in porous targets. Icarus, 180:514–527.

[5] Melosh, H. J., Ryan, E. V., and Asphaug, E. (1992). Dynamic fragmentation in impacts: Hydrocode
simulation of laboratory impacts. J. Geophys. Res., 97(E9):14735–14759.

[6] Collins, G., Melosh, H. J., and Wünnemann, K. (2011). Improvements to the epsilon-alpha compaction
model for simulating impacts into high-porosity solar system objects. International Journal
of Impact Engineering, 38(6):434–439.

[7] Collins, G. S. (2014). Numerical simulations of impact crater formation with dilatancy. Journal of
Geophysical Research - Planets, 119:2600–2619.

[8] Collins, G. S., Elbeshausen, D., Wünnemann, K., Davison, T. M., Ivanov, B., and Melosh, H. J.
(2016). iSALE-Dellen manual: A multi-material, multi-rheology shock physics code for simulating
impact phenomena in two and three dimensions. dx.doi.org/10.6084/m9.figshare.3473690.

[9] Collins, G. S., Melosh, H. J., and Ivanov, B. A. (2004). Modeling damage and deformation in impact simulations. Meteoritics and Planetary Science, 39:217–231.