ACSE-9 Independent Research Project

Parallelization of iSALE using OpenMP

Xianzheng Li

GitHub: x1ng4me
Email: xianzheng.li18@ic.ac.uk

Supervisor: Prof. Gareth Collins
Department of Geoscience and Engineering
Imperial College London
August 30, 2019

Abstract

iSALE (impact-SALE) is a well-established shock physics code (written in Fortran 95) that has been developed and used for more than two decades to simulate impact cratering. The two-dimensional version of the code module is serial, which will affect the efficiency of the software. This paper looks at improving the performance of the iSALE-2D code by adding OpenMP parallelization. By examining various sample cases, testing different scheduling methods and re-factoring part of the codes, the following paper clearly demonstrates the improvement of code efficiency.

Keywords: iSALE-2D, Parallelization, OpenMP

Introduction

The iSALE code has been benchmarked against other 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-2D has been used for more than 20 years and used in different fields, improves the performance and the efficiency of the software has become a development 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 growing stronger and stronger and can provide more creative ways of improving the performance of the code. Parallelization is one of the most efficient methods to result from the development of computer architectures.

Thus, this project has two main objectives. First, the project aims to parallel the highest-priority subroutines with iSALE, verify the approach and establish a template for further development. Since the iSALE solution algorithm involves a series of large and complex loops, adding parallel programming to the code has the potential to reduce the run time substantially. Second, the project will assess the performance of the parallelization and determine the optimum scheduling method for typical iSALE simulations - dynamic, static and guided - and combine with other parameters such as chunk size to find the best combination for a specific test case, then gain the optimal performance of the paralleled iSALE.

This report includes a brief overview of iSALE code; profiling analysis to determine the priority subroutines for parallel efficiency; summary of principal subroutine to parallelise; obstacles to parallelization and strategy to overcome these;

test used to verify correctness; analysis of parallel performance, and scheduling, sensitivity to problem type and problem size.

Background

Introduction to iSALE

This project involved the parallelization of 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 is rarely used in practice (Figure 1).

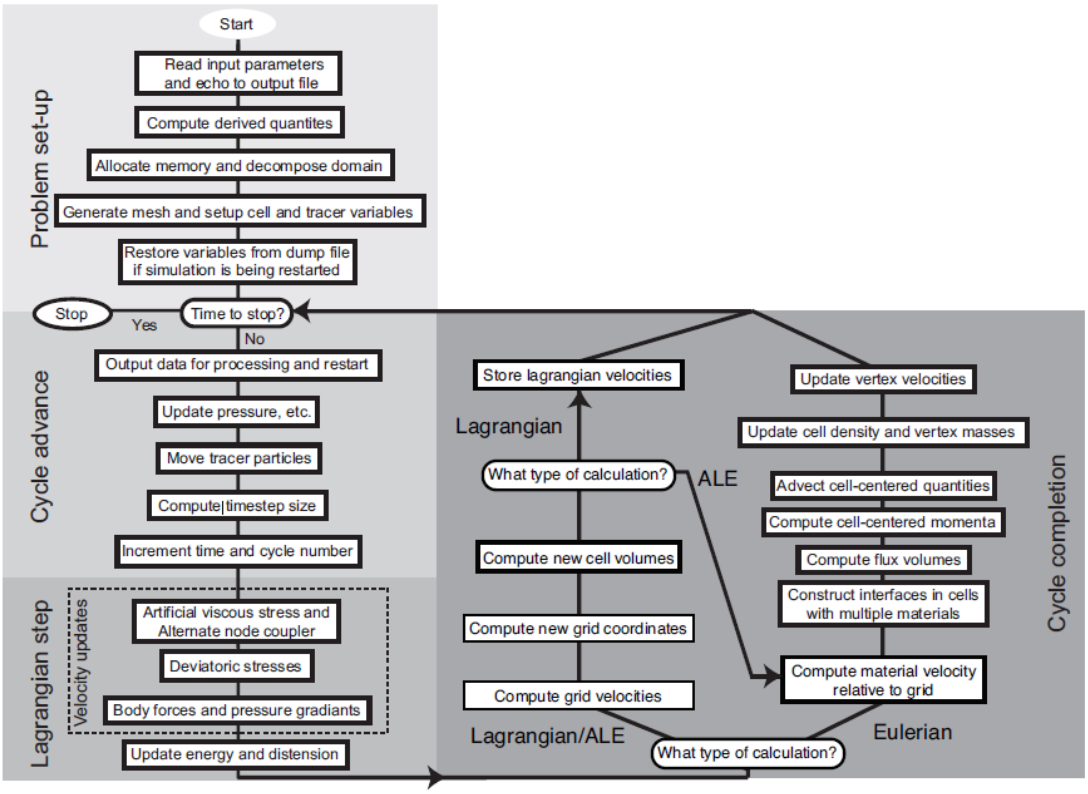


Figure 1: Flow Diagram of iSALE-2D

With developments in recent years, the iSALE program includes an array of material models - rheologic models, porosity models, strength models, a thermal softening model, a damage model and a 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 stepping scheme results in a relatively straightforward algorithm involving a series of loops over the two-dimensional domain. Furthermore, no external library is used in the iSALE program, which reduces the software dependence on external resources.

Parallelization Strategy

Open Multi-Processing (OpenMP) and Message Passing Interface (MPI) are two main parallel methods in parallel programming. OpenMP is a way to program on shared memory devices. This means that the parallelism occurs where every parallel thread has access to all of the data. MPI is a way to program on distributed memory devices. This means that the parallelism occurs where every parallel process is working in its own memory space in isolation from the others. There are two main reasons that the OpenMP was preferred to parallelise iSALE2D in this project. First, compared with MPI, the set-up of OpenMP is more straightforward. Code structure would be changed significantly to allow addition of MPI. Second, since typical iSALE-2D simulations do not require a large

memory footprint, it does not need to allocate a huge memory and can be finished in a single hyper-performance computer. OpenMP is more suitable for this situation.

OpenMP Scheduling

To obtain the higher efficiency of the code, parallel the code with the optimal scheduling method is required. The least complicated way of doing this is by giving the same number of iterations to each thread. However, this is not always the best choice, since the computational cost of the iterations may not be equal for all of them. Therefore, various ways of distributing the iterations exist, which is the reason to use the *OMP_SCHEDULING(type,chunk)*. The *type* specifies the way in which the work is distributed over the threads. The *chunk* is an optional parameter specifying the size of the work given to each thread. Defining the chunk size is the most complicated facet of the scheduling part. With different type, the principle of defining the chunk size is different (Hermanns, 2002).

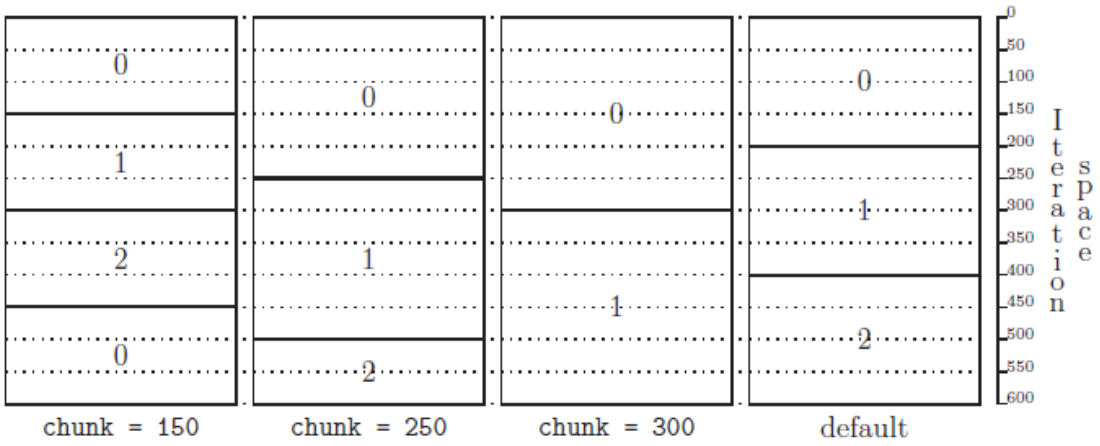


Figure 2: Graphical representation of the example explaining the working principle of the *schedule(static,chunk)*

For static scheduling, assume there is a loop with 600 iterations and distributed across four threads by changing the value of chunk. The results are illustrated graphically in Figure 2. By default, each thread should be allocated 200 iterations. However, if the chunk size changes to 150, thread 0 will have two job orders. When the thread 0 is running, other threads are waiting in an idle state, which is not efficient. When the chunk size is 250, the number of blocks equals the number of threads, which is a positive result. However, this time the sizes of the blocks are not equal since the total number of iterations is not an exact multiple of chunk. When the chunk size equals 300, the 600 iterations will only be divided into only two work sections and the thread 2 will remain idle. The resulting efficiency is equal to the chunk size 150 (Miguel Hermanns, 2002).

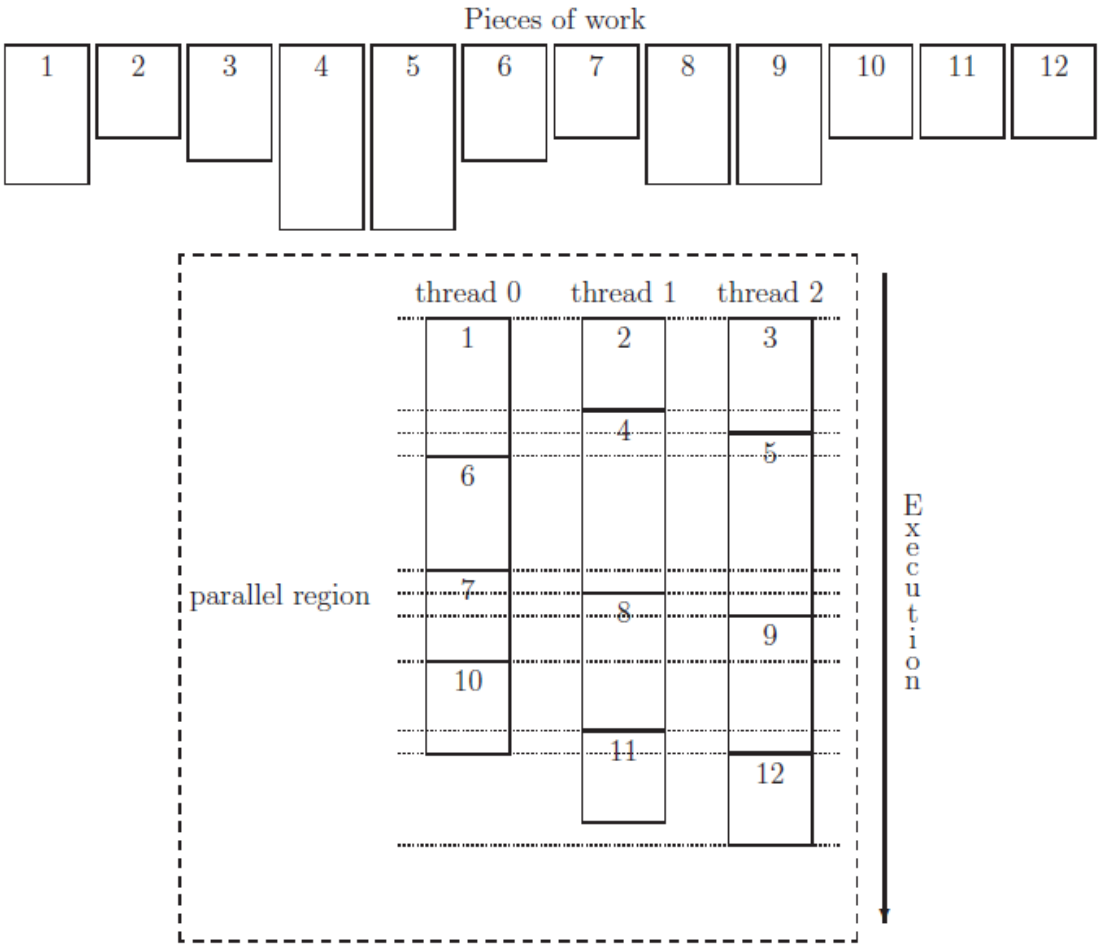


Figure 3: Graphical representation of the example explaining the working principle of the *schedule(dynamic,chunk)*

For dynamic scheduling, Figure 3 indicates that the total work will be separated into 12 different sizes of chunk, and the first three job sections will be allocated to the three threads first. Then, once any of the three threads finishes its work, a new work section will be allocated to the idle thread immediately. Therefore, when determining the chunk size, the chunk size should not be too small nor too large. On the one hand, if the chunk is too large, once the workload is unbalanced, the resulting efficiency will decrease. On the other hand, if the chunk is too small, the thread starting time will increase and be even longer than the run time, which will decrease the efficiency as well.

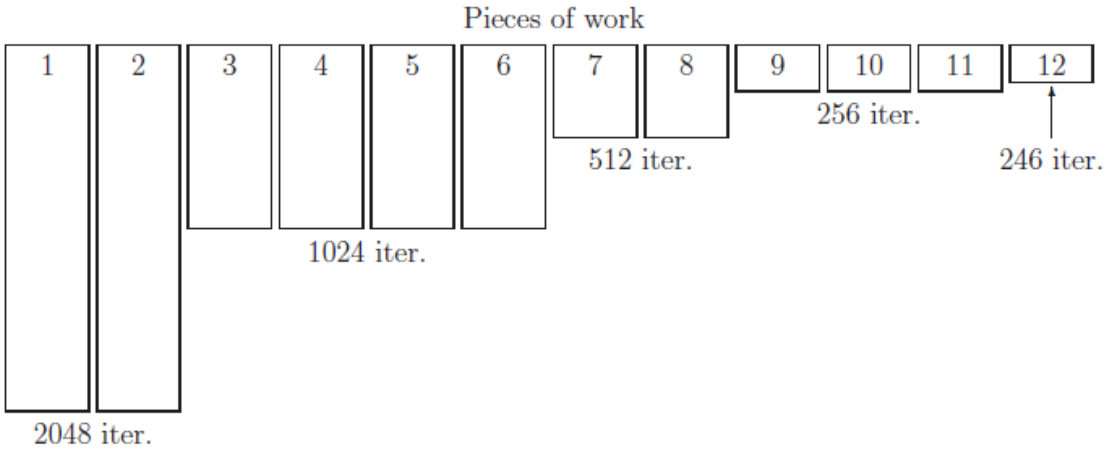


Figure 4: Graphical representation of the example explaining the working principle of the *schedule(guided,chunk)*

For guided scheduling, this method is similar to dynamic, but will reduce the potential overhead time in computational cost. The last piece of work may have a smaller number of iterations than specified in chunk to cover completely the iteration space (Hermanns, 2002). Therefore, the chunk size should not be too small or the threads have to start many times to run a righly small work section. Consequently, the threads' starting time will be much longer than the run time, and the other threads' overhead cannot be ignored.

For non uniformed workload, the running time is varied with three scheduling methods. The dynamic and static method should give better results than static only if there is an unbalanced workload. Because the running time of static will be based on the section with the heaviest workload. At the meantime, all other sections will be finished and threads will stay idle, which is not efficient. However, the dynamic and guided will allocate threads dynamically to make sure that most threads have jobs to do which incurs an additional overhead in processing time. Therefore, determine the optimal scheduling type, and chunk size is the significant issue for improving the efficiency of the code.

Profiling

Code profiling was the first part of the project and was used to point out a clear direction. Although the main objective was to add parallelization to the code, parallelising all loops or all subroutines is not an ideal way to improve the code's efficiency. Because scheduling multiple threads also costs time, once thet start, the threads will remain active which is the overhead. If a loop is not that complicated, adding parallelization will have a negative effect. The VTune amplifier is a commercial application for software performance analysis. It supports both the Mac OS and Linux system and is used to run the entire code, timing each subroutine and generating a configuration analysis report. It provided a list of time cost for each subroutine, which helped users determine which subroutine

should be paralleled first, and which one is the end target of parallelization.

Profiling is essential after each change of the code. For instance, after adding parallelization to one loop, profile the running time and do hot-spot analysis to determine whether the paralleled code sees a significant improvement in speed is required. VTune amplifier can also indicate the running time of each parallel section explicitly. Meanwhile, in the summary from Vtune, it would show the usage and running time of each thread. If the usage of any thread did not make sense, the Vtune could provide a clue to help the user figure out the location of the problem.
In this project, four sample test problems from the set of example problems distributed with iSALE were profiled, which were *Demo2D*,
Aluminum_1100_2D, *Collision2D* and *Chicxulub*. Four cases had different mesh sizes which would cause obvious run time difference. The reason for using four cases in profiling is the problems sample a range of different, yet typical code paths and their reports should ensure a representative result of hot spots.

Cases / Subroutines	Demo2D	Aluminum	Collision2D	Chicxulub
advect	44.7%	66.5%	67.3%	62.5%
velocity_stress	14.3%	15.7%	15.2%	15.7%
state	13.5%	6.7%	5.5%	14.7%
energy	2.2%	2.9%	2.1%	2.3%
movetracer	0.0%	3.8%	1.6%	1.0%
mod_strength_routines	0.5%	1.7%	1.7%	1.7%
I/O	21.9%	0.3%	0.3%	0.2%

Table 1: Run Time Proportion for Top Subroutines

Table 1 shows the top subroutines and their proportions of the run time after profiling. The case *Demo2D* has the smallest mesh size and shortest run time which is around 33s. The I/O time has a high proportion of the total run time which is 21.9% in this case, and the time cost is around 6s. For other cases with larger mesh size and longer run time, the proportion of I/O is around 0.3%, and the time cost for I/O is still around 7s.
Therefore, although I/O section has a high proportion of the run time in case *Demo2D*, it should not be considered as a significant part of parallelization as this example is not representative of a typical simulation. For other subroutines, the subroutine *advect* always had the highest proportion of the run time for all cases. Meanwhile, subroutine *velocity_stress*, *state* and *energy* followed the *advect* in the second, third and fourth position for all cases.
From the results, the subroutine *advect* was identified as the priority for parallel programming. Meanwhile, the subroutine *velocity_stress*, *state*, *energy*, *movetracer* and *mod_strength_routines* which total cost around 30% of the run time should be considered for parallelization if time permitted.

Parallelization Methods

Introduction to subroutine advect

The function of the "advect" subroutine is to transfer field variables from the deformed Lagrangian mesh to the original (fixed) Eulerian mesh. This is achieved by several loops over the two-dimensional array of cells or nodes in the mesh, as outlined in Table 2. In the original version of the subroutine, after some vector operations to initialize temporarily allocated arrays to store intermediate values of advected quantities, the first loop calculates the volume of each material present in each cell that must be fluxed across each cell boundary. To avoid unnecessary storage of duplicated fluxes, the loop is constructed to be order-specific where, for example, the flux across the right face of the cell indexed i is retained for the subsequent iteration, where it is taken to be the flux across the left face of *i+1th* cell. The next loop accounts for the change in field quantities in each cell based on the volume fluxes through each face: the temporary arrays are used to store the flux quantities, which are typically the field of interest scaled by the volume or mass of material in the cell (e.g., the density field is advected as mass, the specific internal energy field is advected as internal energy). An additional loop over cells is then required to divide the advected quantities by the new material volumes/-masses and update the field arrays for use in the next time step. Further loops are required to compute some derived quantities, such as nodal quantities (e.g., mass) that are calculated based on the average of surrounding cell quantities, and to handle the advection of the nodal velocity field, which is done separately based on advected cell-centred momenta. As is common with many loops in iSALE, several of the advect loops update nodal quantities (e.g., velocity) via a loop over cells. This structure requires no special cases for the boundaries and thus results in loops that are efficiently vectorised; however, it implies that the same nodal quantity is updated by visits to multiple cells, which is unsafe to parallelise as it introduces a so-called "race condition."

Since the the subroutine *advect* was identified as the single most important subroutine to parallel, analyze the code structure and decide the position to add parallelization was the first priority. Table 2 shows the main loops inside the subroutine *advect*.

Nr.	Loop over nodes or cells	Description	Safe or Not	Obstacle to Parallelization
0.	/	Allocate and initialize memory for temporary arrays	Yes	N/A
1.	Cell	Calculate volume transfer from cell to cell; construct material boundaries in mixed cells and calculate individual volume fluxes for each material based on material boundary position	No	Order-Specific
2.	Cell	Calculate advection: account for outflux and influx of field quantities across cell boundaries; update material densities and volume fractions	Yes	N/A
3.	Cell	Calculate nodal volume fractions from cell volume fractions	No	Race Condition
4.	Cell	Advect finalise: update global fields from temporary fields; calculate bulk fields for material specific quantities; calculates nodal mass from cell mass	No	Race Condition
5.	Cell	Identify materials in cell based on volume fractions	Yes	N/A
6.	Node	Compute reciprocal nodal mass, based on nodal mass. Prepare velocity for update	Yes	N/A
7.	Cell	Update nodal velocities based on cell-entered momenta fluxes	No	Race Condition
8.	Node	Cap nodal velocity to eliminate spurious velocities	Yes	N/A

Table 2: Summary of original version of *advect* routine. "Safe or Not" refers to whether the loop can be paralleled without changes

Table 2 shows that the subroutine *advect* had 8 loops originally, some of them looped over nodes, the others looped over cells. Meanwhile, race condition and order-specific are two significant problems for adding parallelization. These two problems should be solved by code re-factoring before parallelization.

Parallelization Strategy

- **Contained Subroutines -> External Subroutines**
  Except the two big problems mentioned earlier, some small "contained" subroutines were involved in a main subroutine, which affected the efficiency of

the code and made it difficult to add parallel. To solve this, small subroutines outside the large one were separated, all used parameters at the beginning of each subroutine would be defined, and put all input parameters into the brackets in the small subroutine line. Then both small and large subroutine were able to be paralleled.

- **Order-Specific Problem**
  In the subroutine *advect*, several temporary arrays would be used in the function. In some arrays, the data must be filled in with a specific direction, for example, from left to right. Adding parallelization will alter the specific orders of the data transportation. The problem can be explained by using the psudo-code below. The original data is in a 2D frame but to explain this simply, 1D frame would be used here.

```
do i=1,nxp   !nxp is the number of nodes
   if (i.eq.1) fr = 0
   VolFlux_l(i) = fr
   VolFlux_r(i) = calculate_right_flux()
   fr = VolFlux_r(i)
end do
```

The pseudo-code indicates that in each iteration, the volume flux on the left side equals to the number of fr, then the volume flux on the right would be updated with the number from the function
`calculate_right_flux`, and finally, update the fr equalling to the volume flux on the right side. The structure indicates that the data passed from the left to the right. However, once adding the parallelization to this, each processor will process one iteration, and the data can not be guaranteed to pass in the specific order because the processing time of each processor will be varied.
To solve this problem, updated the fr value before updating the volume flux on both the left and right sides is necessary. The method could be explained by using the following pseudo-code:

```
do i=1,nxp
   fr(i) = calculate_right_flux(i)
end do
do i=1,nxp
   if (i.eq.1) then
      VolFlux_l(i) = 0
      VolFlux_r(i) = fr(i)
   end if
   VolFlux_l(i) = fr(i-1)
   VolFlux_r(i) = fr(i)
end do
```

The new code can be paralleled easily. it is easy to 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.

- **Race Condition**
  Race condition would exist on the node after adding parallelization to some loops over cells. The problem can be explained by the following code example:

```
! Loop over cells
! ie, je are the number of cells in i and j directions
do j = js,je
   do i = is,ie
      qmomv_b(X:Y) = 0.25D0*vmomp(X:Y,1,i,j)
      !Update nodal velocities at corners of cell,
      !based on cell momenta
      V(X:Y,i+1,j)   = V(X:Y,i+1,j)   + qmomv_b(X:Y)*rmv(i+1,j)
      V(X:Y,i+1,j+1) = V(X:Y,i+1,j+1) + qmomv_b(X:Y)*rmv(i+1,j+1)
      V(X:Y,i,j+1)   = V(X:Y,i,j+1)   + qmomv_b(X:Y)*rmv(i,j+1)
      V(X:Y,i,j)     = V(X:Y,i,j)     + qmomv_b(X:Y)*rmv(i,j)
   end do
end do
```

In the previous serial code structure shown above, the loop was over cells, each cell was visited one at a time, but it was a nodal quantity (velocity) that is being updated. The nodal velocities of each of the four corners of the cell were updated. However, after adding parallelization to this code, various cells would be visited synchronously, and the information on nodes shared by those cells could be updated at the same time. As a result, the race condition would cause data overwriting on the cells. To avoid this race condition, the method is re-factoring the loop. The re-factored code is like the following code example:

```
! now a loop over nodes;
! iep, jep are the number of nodes
do j = js,jep
   do i = is,iep
      ! Update nodal velocities, based on the momenta
      ! in surrounding cells
      ! Consider boundary condition first
      if (j.eq.1) then
```

```
         if (i.eq.1) then
            V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i,j)*rmv(i,j)
         else if (i.eq.iep) then
            V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i-1,j)*rmv(i,j)
         else
             V(X:Y,i,j) = V(X:Y,i,j)
                          + 0.25D0*rmv(i,j)*(vmomp(X:Y,1,i-1,j)
                             + vmomp(X:Y,1,i,j))
         end if
      ...
      else
         ! Calculate all nodes not on the boundary
         V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i,j)*rmv(i,j)
         V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i,j-1)*rmv(i,j)
         V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i-1,j)*rmv(i,j)
         V(X:Y,i,j) = V(X:Y,i,j) + 0.25D0*vmomp(X:Y,1,i-1,j-1)*rmv(i,j)
      end if
   end do
end do
```

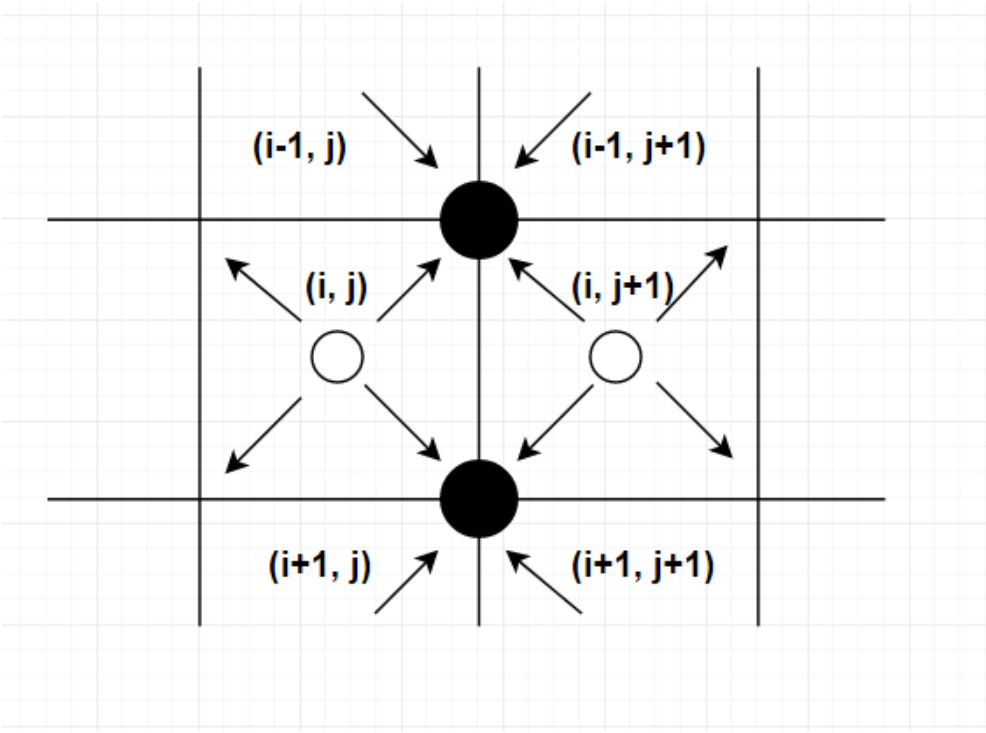


Figure 5: Sample Example of Mesh Grid and Condition after Re-factoring

Figure 5 indicates that the information updated over it is still the nodal quantity that is being updated, it is just that the loop is over nodes, so each node is updated in one go, not in multiple steps by visits to different cells.

For example, the upper node would read information from four neighbour cells, but the information in cells would not be changed, and there was no communication between two neighbour nodes. Therefore, this method avoided the race condition, and all information would be updated safely. A limitation of this re-factoring was that it introduced the need for a number of if-statements inside the loop to handle special conditions on the boundaries, This led to a modest increase in run time.

- **Loop Merging**
  In the previous code, several loops were called in the subroutine *advect* (see Table 1). A benefit of re-factoring loops that updated nodal quantities from cell-loops, which previously had to be done separately could be merged in to one large loop. As a result, the large loop can be paralleled easily, and the run time for the new code is shorter than the previous one.
  The loops merging was not possible before finishing code re-factoring, because in the original version of the code, some loops are over nodes, the others are over cells, these loops could not be merged in one large loop before re-factoring all of them loop over cells.

Then, all potential problems had been resolved. The new code structure for subroutine *advect* are listed in the following table.

Nr.	Loop over nodes or cells	Description	Safe or Not	Contain Old Loops
0.	Cell	Initialise memory for temporary arrays; calculate and store volume transfer between cells	Yes	Part of 1, 0.
1.	Cell	Construct material boundaries in mixed cells and calculate individual volume fluxes for each material based on material boundary position	Yes	Part of 1.
2.	Cell	Calculate advection: account for outflow and influx of fields across cell boundaries; update material densities and volume fractions	Yes	Part of 2.
3.	Cell	Advect finalise: update global fields from temporary fields; calculate bulk fields for material specific quantities; identify materials in cell based on volume fractions	Yes	4 & 5
4.	Node	Calculate nodal volume fractions from cell volume fractions (REFACTORED); compute nodal mass (REFACTORED) and reciprocal mass, based on cell mass. Prepare velocity for update; Update nodal velocities based on cell-entered momenta fluxes (REFACTORED); Cap nodal velocity to eliminate spurious velocities	Yes	Part of 2 & 4,3,6,7,8

Table 3: Summary of re-factored version of *advect* routine. "Safe or Not" refers to whether the loop can be paralleled without changes. The total number of loops has reduced from 8 to 5.

Parallelization with OpenMP Parallel Do

- **Implementation**
  After all problems resolved, the initial parallelization orders were added to all new loops by using **omp parallel do**. This order makes the immediately a do-loop be paralleled. For example:

```
!$omp parallel private(j)
!$omp do
do j = 1, 1000
   ...
end do
!$omp end do
!$omp end parallel
```

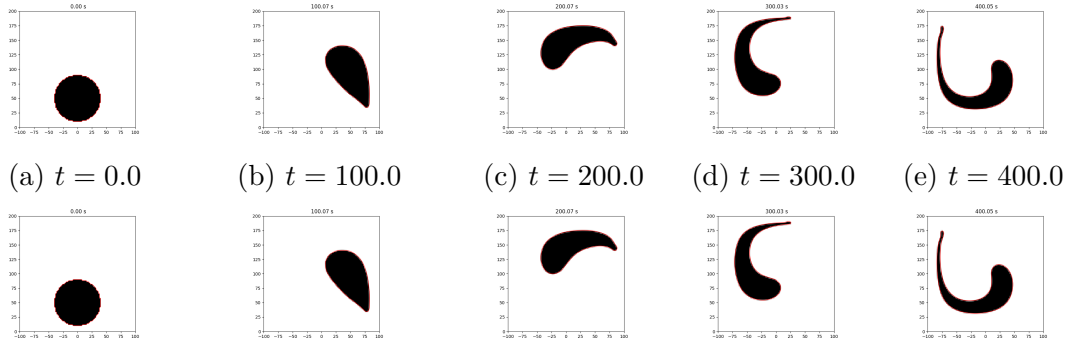
This order distributes the do-loop over the different threads, each thread computes part of the iterations. For example, if 10 threads are in use, then in general each thread computes 100 iterations of the do-loop: thread 0 computes from 1 to 100, thread 1 from 101 to 200 and so on (Hermanns, 2002). Meanwhile, because iSALE problems typically involve square domain, parallelise the outer, j loop is the approximate option of parallelization.

- **Pin Threads with omp_pinning**
  The order **omp do** has the same functionality with **omp do schedule(static)**. Before testing with other scheduling methods, some tests were finished for this under current scheduling method. The results indicated that the run time decreased, but when 12 or more threads were used, the run time would go much longer which was a puzzling result.
  This problem was caused by the **omp_affinity**. This problem is important on multi-socket nodes. When the computer has more than one socket, communication between two threads will cost more time, since the two threads may be located on different sockets. Thus, binding all threads in one socket will reduce the communication time and thereby speed up the code. A safe default setting is using **export omp_proc_bind=true** in the **run-script.bash** file, which pins all threads on one socket, and prevents the operating system from migrating a thread. This prevents many scaling problems. After several tests with thread pinning, the run time decreased significantly. Additionally, the run time variance of several same tests was small, which indicated that the communication between threads was stable.

- **OpenMP Scheduling Methods Investigation**
  Since the positive results after using **omp do** and binding threads on a single socket were achieved, the next step was considering changing the scheduling type and chunk size. As mentioned earlier, determine the optimal scheduling type, and chunk size is the significant issue for improving the efficiency of the code.
  There are three different scheduling types, which are **(static, chunk)**, **(dynamic, chunk)** and **(guided, chunk)**. Except the type of **static**, the other two types were also investigated with different chunk sizes during the project. Due to the time limitation, the investigation of chunk size was not such complicated. The used chunk size for **dynamic** was half of the chunk size of the **static**. Since under the option **dynamic**, each thread would

be allocated a new job section after finishing current one, half the chunk size meant double the amount of job sections, which should be a reasonable number for **dynamic**. The used chunk size for **guided** is the same as the static, which is the approximately chunk size from Intel's engineer.

Verification



(a) $t = 0.0$ (b) $t = 100.0$ (c) $t = 200.0$ (d) $t = 300.0$ (e) $t = 400.0$

(f) $t = 0.0$ (g) $t = 100.0$ (h) $t = 200.0$ (i) $t = 300.0$ (j) $t = 400.0$

Figure 6: Verification of subroutine *advect* after Re-factoring

After all code re-factoring and loop merging, the first priority was to test the functionality of the subroutine *advect*, to ensure the changed code could produce same results. A simple test of the advection routine, included as part of iSALE's standard test suite, was used to verify the re-factoring of subroutine *advect*. This test imposes a fixed rotational (vortex) velocity field that advects a circle of material around. Figure 6 shows the rotating tests before and after code re-factoring. Five time steps were picked out, the top row is the test results after code re-factoring, the second row is the original results. There was no difference between these two versions, which meant the re-factored code did not change the correct function of *advect*. This step ensured all following tests were based on a reliable code version.

As mentioned earlier, the test file for iSALE-2D already existed. The purpose of the test file is to compare the results of a simple impact simulation after 100 time steps with those of a previously calculated reference simulation. The report of the test would reveal whether the simulation results using the new code give identical results to the previous code version. This test file is helpful to guide users towards the source of a problem and is also a significant step before making any further changes. However, the particular test had some issues during the code re-factoring and parallelization. After removing the race condition and data transportation, the results could not pass the test continuously.

To determine the source of a problem, the result of each time step were printed out and compared with the previous correct answer. The differences between the two versions were very small, approximately 1×10^{-17} (i.e. numerical precision). Compared the initial threshold 1×10^{-8}, the numerical error for each single time step was able to pass the test. However, the cumulative numerical error after 100

time steps became larger than the threshold. To double check the test results, the visualization output before and after code re-factoring are listed below.

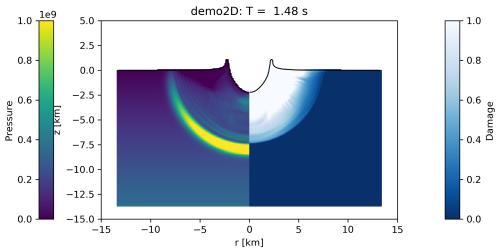


Figure 7: Before Code Re-factoring.

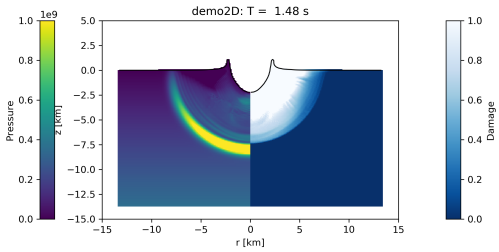


Figure 8: After Code Re-factoring.

One time step during the tests was picked out, figure 7 and 8 show the visual results before and after code re-factoring and parallelization. The two figures looks have no difference, or some tiny differences which will not affect the results. Therefore, the error occurred in the test function was recognized as numerical precision error created during the calculation. The best way to reduce the error is to re-define the test file. Thus, after full consideration, the benchmark file was replaced with the new test results. Then, after adding any OpenMP orders, the test file was used to made sure every change of the code pass the test.

Performance Results

Description of Three Sample Cases

There were three sample cases I used to test the parallelization performance, which are *Aluminum_1100_2D*, *Collision2D* and *Chicxulub*. These cases were chosen as being representative of a range of typical iSALE simulation problems with different geometries, number of materials and material model options. The key model differences between each case are listed in the table below.

Case Name	Mesh Size (H×V)	Material #	Non-planar Target Geometry	Strength	Porosity
Aluminum_1100_2D	250×290	1	✓	Rock	✓
Collision2D	280×360	3	✗	Metal	✗
Chicxulub	310×280	3	✗	Rock	✗

Table 4: Description of Three Test Cases

The *Collision2D* has the largest mesh size with three material. It also involves a non-planar problem geometry with the most mixed cells in any calculation and a porous material model. The *Chicxulub* and *Aluminum_1100_2D* both involve planar targets. The difference is the *Chicxulub* involves three materials and a complex rock material model. *Aluminum_1100_2D* involves only a single material and a simple metal strength model.

Serial Code Performance

The following chart indicates the serial run time of the three code versions of sample case *Collision2D*. The results in this section were generated with full time steps.

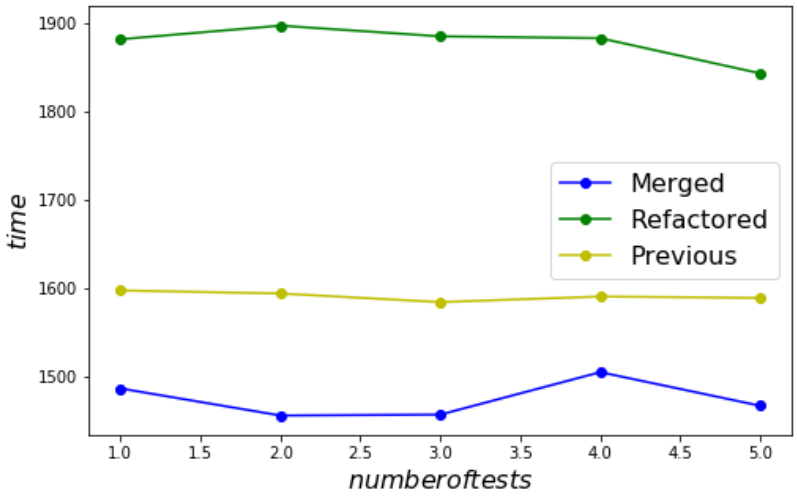


Figure 9: Run Time of Three Different Code Versions for Collision2D

To ensure the accuracy of the run time, each version of the code ran five times and the running time were recorded. The figure 9 shows that after re-factoring the loops, the run time increased for all three cases, which was reasonable because the code re-factoring introduced more conditional cases inside loops to deal with boundary cases. After merging 8 loops into 5, the run time of the code was shorter than the original code, which indicated that reducing the number of loops will improve the efficiency of the code. Therefore an unexpected benefit of preparing the code for parallel processing was a speedup up the serial code.

Case Name	Serial Run Time(s)	Re-factored Serial Run Time(s)	Loops Merged Serial Run Time(s)
Aluminum_1100_2D	2220.161	2465.135	2098.648
Collision2D	1597.856	1882.035	1286.998
Chicxulub	12603.612	14482.718	10600.417

Table 5: Serial Run Time of Three Test Cases with Different Code Versions

Table 5 shows the run time results with three test cases. The trend of the run time changes in case *Chicxulub* and *Aluminum_1100_2D* were consistent with *Collision2D*.

Multi-Threads Performance

Before conducting parallel performance tests, a predicted optimum performance enhancement which could be used to compare with test results were necessary. The optimal results are based on Amdahl's law, a formula that gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. There were three equations which would be used to calculate parameters in further results.

$$T_{opt} = (1-p)T_{1-thread} + \frac{p}{s}T_{1-thread} \tag{1}$$

$$T_{relative\_1} = \frac{T(s)}{T_{1-thread}} \tag{2}$$

$$T_{relative\_opt} = \frac{T(s)}{T_{opt}} \tag{3}$$

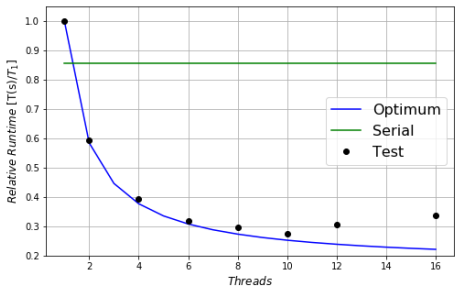
T_{opt} is the theoretical optimum run time if the subroutine *advect*'s parallel efficiency is perfect. The $T_{1-thread}$ is the run time for one thread with OpenMP enabled, which is longer than the serial run time owing to the overhead of OpenMP scheduling on a single thread. It includes the execution time of the part that would not benefit from the improvement of the resources and the execution time of the one that would benefit from it. The fraction of the execution time of the task that would benefit from the improvement of the subroutine *advect* is denoted by *p*. The one concerning the part that would not benefit from it is therefore *1-p* (McCool, 2012), and *s* is the number of threads. The *T(s)* is the run time with different threads. $T_{relative\_1}$ is the relative run time which divided the *T(s)* by run time with one thread. $T_{relative\_opt}$ is the relative run time which divided the *T(s)* by the optimum run time. The tests ran with 1, 2, 4, 6, 8, 10, 12, and 16 threads.

	Aluminum_1100_2D		*Collision2D*		*Chicxulub*	
Threads	Run Time	Relative Time	Run Time	Relative Time	Run Time	Relative Time
1	453.603	1.000	1318.768	1.000	1683.688	1.000
2	295.413	1.009	783.106	1.014	1074.623	1.019
4	215.425	1.020	520.989	1.044	758.913	1.021
6	191.706	1.047	429.191	1.032	661.157	1.032
8	184.315	1.080	393.482	1.089	625.814	1.066
10	173.581	1.076	361.408	1.086	588.057	1.061
12	184.618	1.178	403.417	1.281	638.504	1.196
16	210.768	1.401	444.699	1.519	710.959	1.394

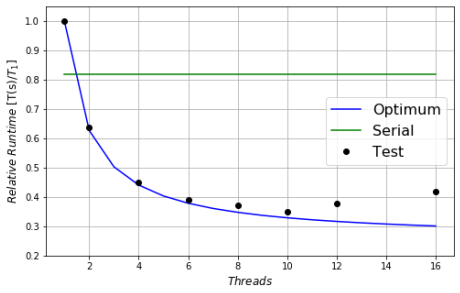
Table 6: Multi-threads Run time and Reletive Run Time of Three Test Cases

Due to the time limitation, the *Aluminum_1100_2D* and *Chicxulub* used eight time steps for performance tests due to time limitation, and the *Collision2D* used full time steps. Each run time was taking average of running same test three times. The table 6 shows the run time and relative run time with varied thread number

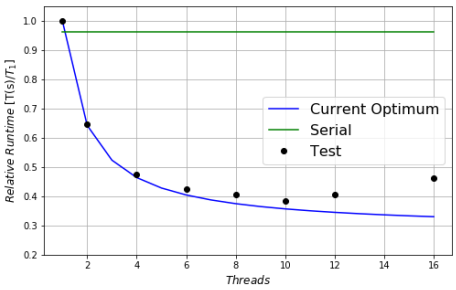
of three test cases. With more threads, the run time decreased significantly. The relative run time shows how close the test results to the theoretical optimum run time.



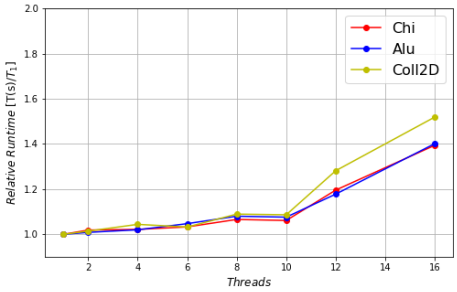
(a) Collision 2D Performance, $p = 0.83$



(b) Chicxulub Performance, $p = 0.745$



(c) Aluminum Performance, $p = 0.714$



(d) Normalized Performance

Figure 10: Normalized Multi-threads Performance of Three Difference Cases

Since the values of p were different which would cause the differences for the theoretical optimum run time, the parallelization performances for three cases were plotted into three separate figures above (Figure10 (a), (b) and (c)). The green line in the three figures are original serial relative run time, the blue line is the theoretical optimum run time, and the dots are test results.

The sub-figure (a), (b) and (c) in figure 10 illustrate $T_{relative\_1}$ of sample case *Collision2D*, *Chicxulub* and *Aluminum_1100_2D*. Although the theoretical optimum run time varied, the performances of the code with different cases were similar, and the trends of relative run time were similar as well. The real run time was closed to the theoretical optimum run time when the used threads were less than 12 (which was the number of physical cores on the CPU used in this work), which means the stability and functionality of the code is reliable.

The sub-figure (d) shows the $T_{relative\_opt}$ of three cases. Since the relative run time is used to show how close the test results to the theoretical optimum run time. The ideal result for each case should be a flat line at $T_{relative\_opt} = 1$, which indicates with each number of threads, the relative run time is consistent with the theoretical optimum run time. In figure (d), the part of the line before 12 threads

were flat and closed to 1, which meant the performance of the code with less than 12 threads is great. With 12 and 16 threads, the relative run time went higher.

The reason for the longer run time and poor performance with 12 and 16 threads is the hardware issue. The workstation for this project used Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, with 12 cores, and 24 threads because of hyper-threading. Therefore, when the allocated threads reached or beyond the physical limitation of the hardware, the communication would be messed up, and the efficiency of the parallelization would fall down and become unstable. Although the the *omp_affinity* issue was recognized and fixed, the limitation of the hardware was still the point for longer run time with more than 12 threads. Another potential reason is that the current mesh size was too small and 16 threads were too many; the threads' scheduling time was more than the running time. With a higher resolution, which means larger mesh size, the performance of tests with 12 and 16 threads would improve further.

Performance for Loops Merging

As introduced earlier, eight loops had been merged into five and the OpenMP parallelization had been added to these five large loops. The performances before and after loops merging are indicated in the following chart.

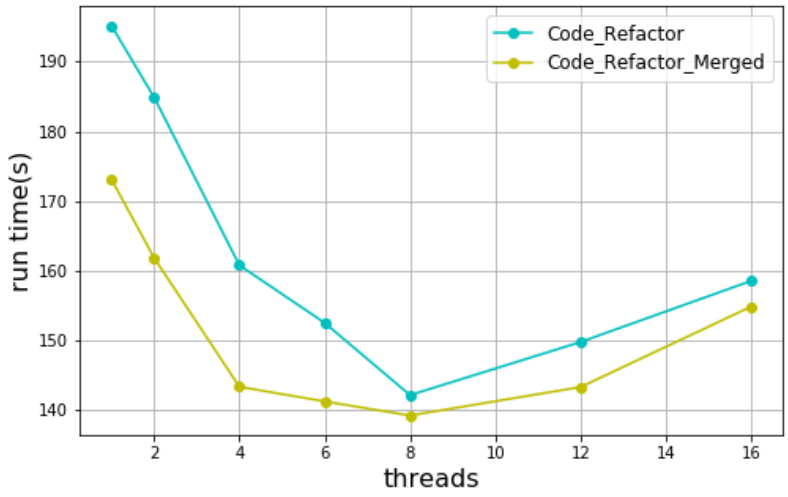


Figure 11: Comparison Before and After Loops Merging

Figure 11 illustrates that after re-factoring and merging the loops, the running time had a significant decrease compared with the code before re-factoring. Meanwhile, the trend of run time change is consistent with previous results. The run time went up from using 12 or more threads in the tests because of the hardware issue mentioned above.

Meanwhile, the run time decreased rapidly from using one to four threads. Then, the decrease rate went down at six and eight threads. The potential reason for this situation is, with more threads used, the workload on each thread was smaller, each thread would use shorter time to finish the job. However, the time cost on starting threads and communication increased, which reduced the efficiency of the code.

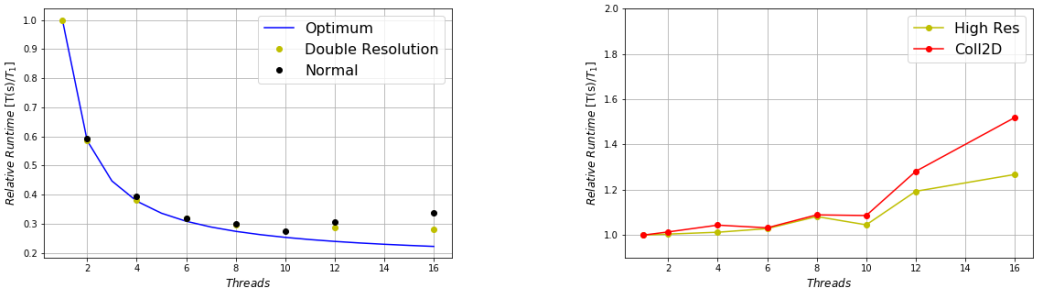
Performance for Higher Resolution

As introduced earlier, one of the potential reasons for poor performance with 12 and 16 threads is, the workload were too small on each thread, then the OpenMP starting time would be longer than calculation time. Thus, double the resolution of the cases which meant double the mesh size to increase the workload should expect a better performance for 12 and 16 threads.

	Collision2D		*Collision2D_High_Resolution*	
Threads	Run Time	Relative Time	Run Time	Relative Time
2	783.106	1.014	2092.426	1.005
6	429.191	1.032	1129.431	1.029
10	361.408	1.086	1005.745	1.116
12	403.417	1.281	1016.384	1.193
16	444.699	1.519	1001.058	1.267

Table 7: Multi-threads Run time and Relative Run Time with Different Resolutions

The table 7 shows the run time and relative run time for *Collision2D* original and double resolution. The results in the relative time were more straightforward to show improvement of the code performance with higher resolution example, especially for 12 and 16 threads.



(a) Relative Run Time (b) Normalized Performance

Figure 12: Multi-threads Performance with Higher Resolution of *Collision2D*

The reuslts in table 7 and figure 12 sort of proved that the workload from original *Collision2D* was not enough, and usage of the threads were not efficient. The better results could be expected with a triple or higher resolution case.

OpenMP Scheduling Methods Investigation

The scheduling type in previous tests was *(*static). Since some positive results had been achieved, consider changing the scheduling type and chunk size to determine if the efficiency could improve even further is the next step.

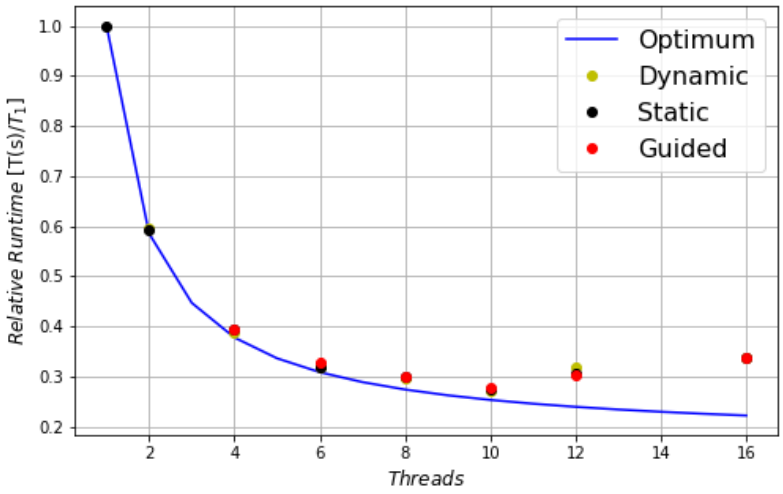


Figure 13: Scheduling Methods Investigation of *Collision2D* with Original Resolution

The initial mesh size on the vertical side of *Collision2D* is 360. During the investigation, the type was changed to dynamic, and chunk size was set as the half of the static. With different threads, the chunk size would be varied. After several tests, the type was changed to guided, and set the chunk size the same as the static, which was the suggested chunk size from Intel's engineer.

The figure 13 shows that the differences between the three scheduling types are small, offering no valuable information for a decision on which type should be used for final tests. From the principle of scheduling types, there should be some differences with using different types.

There are two potential reasons for the tiny differences in this project.

- Balanced Workload
  As introduced earlier, each thread had equal work sections when the *static* type applied. One disadvantage of the *static* type is when the workload were not balanced for different sections, some threads would finish the work quicker than the others, and cause significant idle time. The *dynamic* and *guided* type would allocate the threads which finished the work section with a new one. The idle time would be reduced, then improve the efficiency of parallelization. The results in figure 13 show the relative run time of *static*, *dynamic* and *guided* are similar, which potentially indicated that the workload are balanced. Then, no matter which type was used, the run time would be stable and similar.

- Chunk Size
  The second potential reason is, the chunk size should be re-considered. Due to the time limitation, the tests for different chunk size were not substantial. Only two different chunk sizes were tested for *dynamic*, and one for *guided*. Find the optimal chunk size may cause a significant improvement of the parallelization performance.

Currently, since the difference can be ignored, the most straightforward method which is the *static* was used in this project.

Discussion and Conclusion

After code re-factoring, adding or removing OpenMP to different subroutines, and conducting a multitude of tests, current code version has a good performance of parallelization, which is close to the optimum for up to 12 threads (the physical number of cores on the machine used in this work). The performance of the code is consistent with different cases, and the efficiency of the parallelization is close to theoretical optimum run time. Meanwhile, the running time has reduced to 60% compared with the serial code when paralleled with two threads, and can reach 30% with 10 threads.

Reflecting on this independent research project, there were two most difficult tasks to resolve in completing the implementation.

- The first is to re-factor loops from looping over nodes to cells. Since the existing of the race condition and the order specific problem, re-factoring of the structure was necessary. However, the data used in this project are more than two-dimensional, and fully understand the structure of the code was required before making any changes.
- The second is identifying the OMP_PINNING issue. We struggled with the performance of 12 and 16 threads for quite a long time, and we always thought the weak performance was caused by parallelization rather than the limitation of the hardware.

There are four main strengths of the final solution of this project. First, this solution is based on adding OpenMP to the serial code. OpenMP is a more straightforward tool to parallel codes, and the code will be easier to read and helpful for further development. Consequently, re-factoring the code and reducing the number of large loops make the structure clearer, which will reduce the running time and enhance the efficiency of the code. Furthermore, the current solution and results are significantly close to the ideal solution, especially given that only part of the code were paralleled. Finally, the current solution and results are based on a multitude of tests with many different parameters, and the solution is reliable.

However, there remain a few limitations. First, since the workstation used in this project has 12 cores and 24 threads because of hyperthreading, it is difficult to

test the code with more threads. Second, because of the time limitation, running more tests with higher resolution is not possible but is necessary because higher resolution is more closely aligned with the real situation.

Thus, for the next steps, tests with triple or higher resolution and more threads are necessary to assess the performance of the code. Meanwhile, due to the time constraint, chunk size tests are conducted in limited numbers. Find the optimal chunk size is still one the of the most important task for improving the efficiency of parallelization. Lastly, as mentioned earlier, there were 5 more subroutines in the profiling results which could be paralleled as well. Add parallelization to those subroutines would extend the proportion of the run time that should benefit from parallel efficiency to more than 90% ($p > 0.9$).

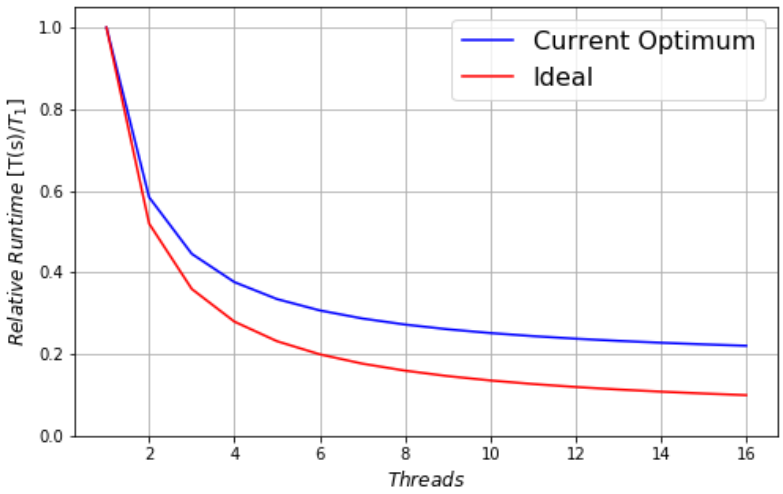


Figure 14: Ideal Relative Run Time of Collision 2D with Further Parallelization

The red line in figure 14 is the new theoretical run time based on 95% of the run time being benefited from parallelization. If the code performance after further parallelization can still close to the new theoretical run time, it will reduce 90% of the serial run time, which will be a huge improvement of iSALE.

Bibliography

[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.

[10] Miguel Hermanns (2002). Parallel Programming in Fortran 95 using OpenMP.

[11] Michael McCool; James Reinders; Arch Robison (2013). Structured Parallel Programming: Patterns for Efficient Computation. Elsevier. p. 61.

Appendix A

Link to Git-Hub Repository:
https://github.com/msc-acse/acse-9-independent-research-project-x1ng4me

Nr.	Modified Subroutine	Location
1.	*advect*	advect.F90
2.	*advect_finalize*	advect.F90
3.	*advect_restore_fluxes*	advect.F90
4.	*advect_momentum*	advect.F90
5.	*advect_momentum_his*	advect.F90
6.	*advect_outflux*	advect.F90
7.	*advect_influx*	advect.F90
8.	*cal_cmv*	advect.F90
7.	*advect_influx*	advect.F90
9.	*advect_influx*	advect.F90
10.	*update_velocities*	advect_routines.F90
11.	*crop_velocity*	advect_routines.F90
13.	*setup_initialize*	setup_initialize.F90

Table 8: Modified Subroutines and the Location of Each Modified Subroutine