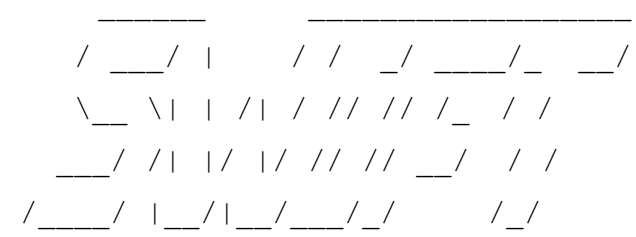
SWIFT exercises and experiments



SPH With Inter-dependent Fine-grained Tasking

Webpage : www.swiftsim.com

This document describes two sets of exercises with the Swift code:

1/ A set of test problems, allowing you to compare SPH flavours in Swift with the Gizmo implementation in Swift

2/ A set of strong scaling tests

We suggest you try the first tests before coffee, and the scaling tests after coffee. We begin by describing how to download the code, compile it, and generate the initial conditions.

When trying this at home, you need at least:

- A c-compiler (GCC, Intel or LLVM/clang)
- The hdf5 library (v 1.8.x)
- python with the h5py module (to create ICs and analyse output)

and optionally:

- An MPI library that supports MPI_THREAD_MULTIPLE
- The Metis graph decomposition library.
- Paraview to visualise the outputs.

Read the README and INSTALL.swift files in the main directory for more details.

Code download and compilation on JUDGE

Download a fresh copy of the code:

```
git clone https://gitlab.cosma.dur.ac.uk/swift/swiftsim.git
```

or copy it from the shared directory on JUDGE:

```
cp -R /work/hpclab/train001/csam-software/swift/swiftsim .
```

Swift modules to use:

Load the following modules:

```
module purge
module load intel/14.0.3 hdf5/1.8.9 parastation/intel12-mt-5.0.27
```

use:

```
./autogen.sh
```

and then

```
./configure CC=icc MPIRUN=mpirun_openib --disable-doxygen-doc
CFLAGS=-g --with-metis=/work/hpclab/train001/csam-software/metis/
```

to convince the autotools to pick-up the right compiler, MPI command and metis library. Then compile with

```
make
```

Python modules to use:

To run the python scripts (generating ICs and analysing the output), the following modules are required:

```
module purge
module load intel/13.1.3 hdf5/1.8.13-mpi3 python/2.7.6-intel
module load parastation/intel13-mt-5.1.0
```

To get vTune, load the module

```
module load vtune
```

Running SWIFT

The main executables are in the ./examples sub-directory. There are four versions

- `test_fixdt` - All particles use a single time-step given as an input parameter
- `test_fixdt_mpi` - All particles use a single time-step given as an input parameter, MPI version
- `test_mindt` - All particles use a single time-step defined as the minimum of all particles' CFL condition.
- `test_mindt_mpi` - All particles use a single time-step defined as the minimum of all particles' CFL condition, MPI version

The code requires a few parameters to run:

- `-d xxx` Initial time-step for the simulation (or fixed time-step when running fixdt).
- `-f xxx` Initial conditions file.
- `-t xxx` Number of threads (per MPI rank) to use.
- `-r xxx` Number of time-steps to run.
- `-w xxx` Minimal number of particles in a task (5000 is a typical value to use).
- `-m xxx` Maximal smoothing length to consider.
- `-c xxx` Final time of the simulation if `-r` is not given

Physics simulation exercises

Several test cases are set-up in the examples directory:

- “Sedov blast wave”: this is the evolution of a point-explosion in 3D, and has a similarity solution
- Sod shock”: the 1D evolution of a hydrodynamical shock, with similarity solution
- “Kelvin-Helmholtz instability”: the instability that arises when two flows shear across each other

You can run all three using the SPH implementation in Swift, as well as with the Gizmo implementation. Swift has two SPH implementations:

- the default is the one implemented in Gadget-2 (version 2.0.7) but using internal energy instead of entropy as thermodynamic variable.
- comment out line 68 in src/const.h to use a version which implements a conduction term and an improved viscosity term

Examine the (minimal) differences in the loop that implements the force calculation:

files runner_iact.h versus runner_iact_legacy.h. The force calculation is function

runner_iact_force. You may want to run these two different SPH flavours in different example directories.

You can also run the same test problems with Gizmo. Do so in a separate directory.

Adapt the python script that plots numerical and similarity solutions to over plot the 3 different implementations.

Detailed instructions:

SEDOV blast wave

Obtain and compile SWIFT (or SWIFT GIZMO):

```
git clone https://gitlab.cosma.dur.ac.uk/swift/swiftsim.git
cd swiftsim
```

For GIZMO then:

```
git checkout gizmo
```

Both versions are compiled as described above.

```
cd examples/SedovBlast
module purge
module load intel/13.1.3
module load python/2.7.6-intel
module load hdf5/1.8.13-mpi3
module load parastation/intel13-mt-5.1.0
python makeIC.py
```

this will generate the initial conditions file, sedov.hdf5. You can use h5ls to examine its contents. To run Swift, you need to load different modules. The simulation runs on two threads (-t 2) with a constant time-step. This low resolution run only takes a minute or so to run - so you can run interactively. The -t flag below tells Swift to use 2 threads.

```
module purge
module load intel/14.0.3
module load hdf5/1.8.9
module load parastation/intel12-mt-5.0.27
../test_mindt -f sedov.hdf5 -t 2 -d 1.0 -c 1.0 -m 0.1
module purge
module load intel/13.1.3
module load python/2.7.6-intel
module load hdf5/1.8.13-mpi3
module load parastation/intel13-mt-5.1.0
python profile.py
```

Obtain and compile Gadget2

```
cp /work/hpclab/train001/csam-software/swift/gadget-2.0.7.tar.gz .
tar -xvf gadget-2.0.7.tar.gz
cd Gadget-2.0.7/
```

```
cp /work/hpclab/train001/csam-software/swift/gfiles.tar.gz .
tar -xvf gfiles.tar.gz
source gadget_modules.csh
cd Gadget2
cp ../Makefile.sedov Makefile
make
```

Running the Sedov blastwave with Gadget2

```
cp ../sedov.param parameterfiles/
mkdir ICs
```

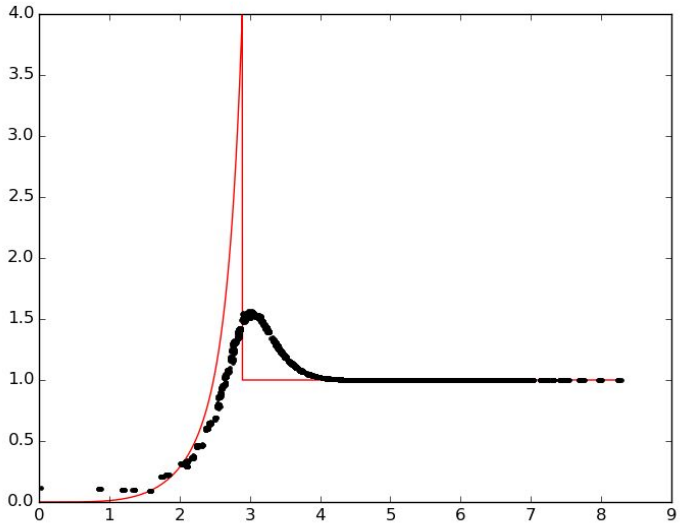
< copy the SWIFT initial condition sedov.hdf5 to the newly create ICs folder >

```
mkdir sedov
./Gadget2 parameterfiles/sedov.param
cd sedov
```

< copy the profile.py and sedov.py from the SWIFT folder to the sedov/ folder >

```
python profile.py
```

A comparison of the solution obtained using Gizmo (for the default low resolution test of 21^3 particles) is shown below, with the similarity solution in red, and the Gizmo solution in



black. Plotted is density as function of radius for this spherical blast. Up the resolution to check for numerical convergence.

Kelvin-Helmholtz test

You can compare the two SPH simulations, Gizmo and Gadget for this test.

running the Kelvin-Helmholtz test with Gadget2

cd .. (Gadget2/)

```
cp ../Makefile.kh Makefile
make
cp ../makeIC_sph.py ICs/
cd ICs/
python makeIC_sph.py
cd ..
mkdir kh
cp ../kh.param parameterfiles/
./Gadget2 parameterfiles/kh.param
```

cd kh

< copy the plot_density.py script from swiftsim/examples/KelvinHelmholtz >

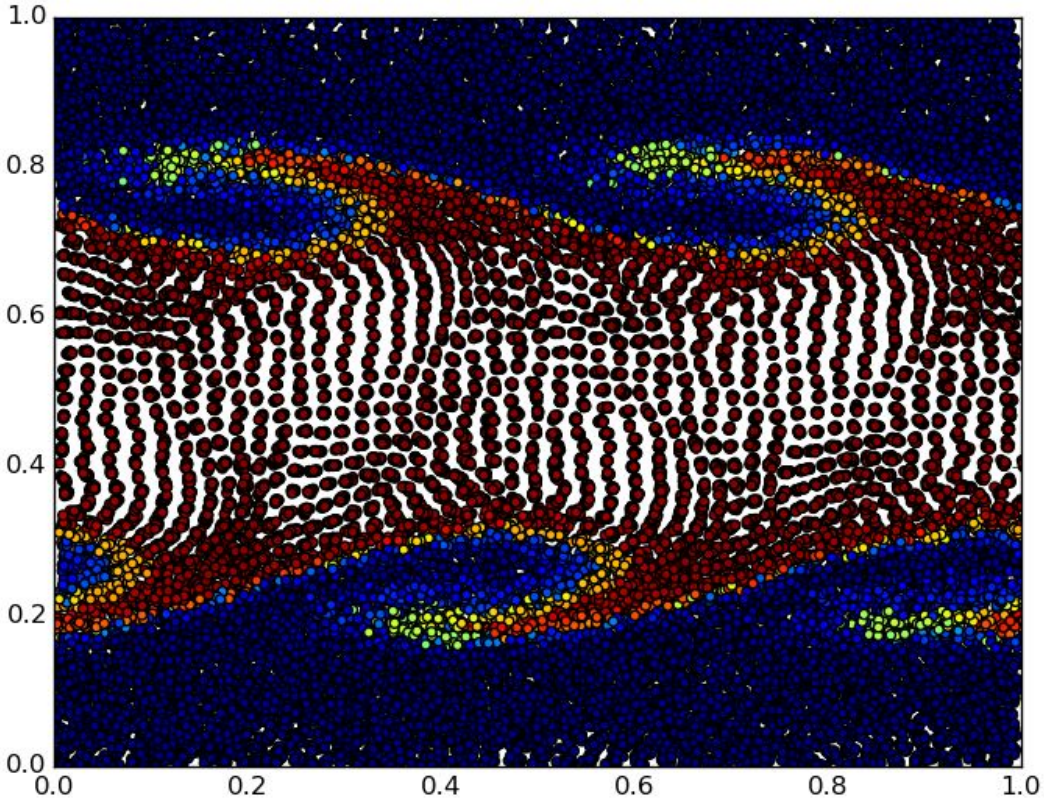
```
python plot_density.py
```

Run the K-H instability with SWIFT GIZMO

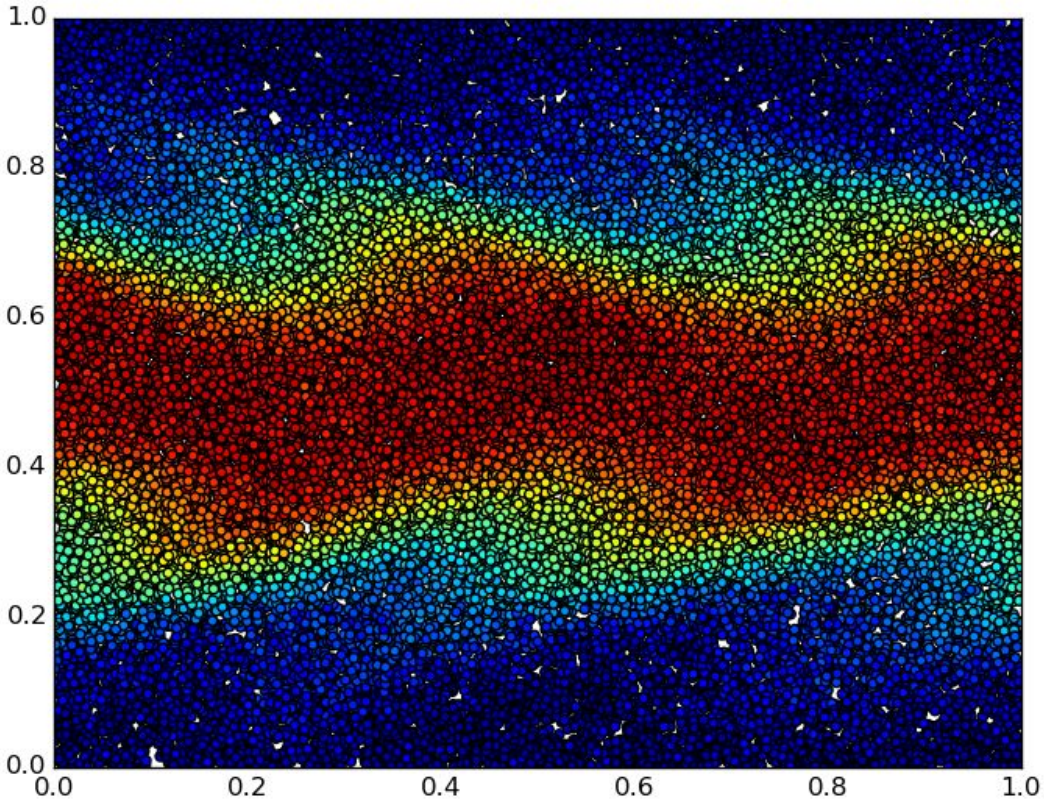
```
cd swiftsim/examples/KelvinHelmholtz
python makeIC.py
../test_mindt -f kelvinHelmholtz.hdf5 -t 2 -d 2.0 -c 2.0 -m 0.1
python plot_density.py
```

OR USE THE kelvinHelmholtz.hdf5 file generated by makeIC_sph.py for a higher resolution (but slower) version

The Gizmo solution using kelvinHelmholtz.hdf5 as ICs is shown below, with particles colour coded according to density.


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

The corresponding SPH solution is



Sod shock test

Running the Sod shock test with SWIFT GIZMO:

```
cd examples/SodShock
python makeIC.py
../test_mindt -f sodShock.hdf5 -t 12 -d 0.12 -c 0.12 -m 0.1
```

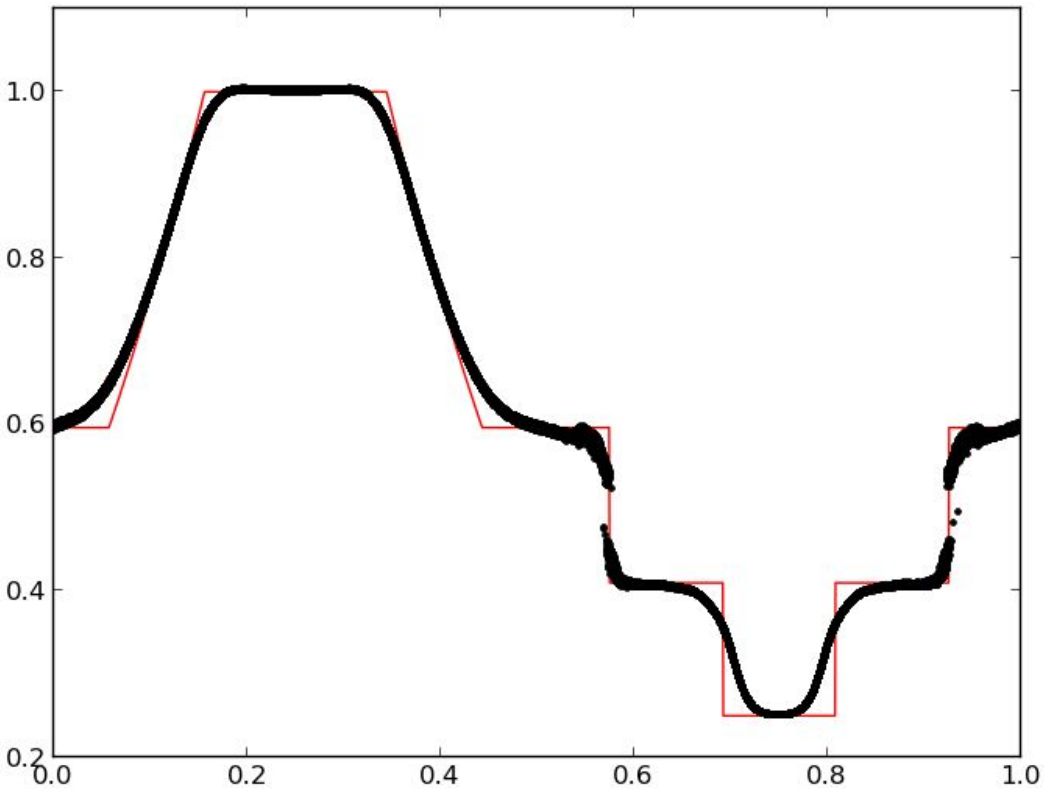
To plot the density profile of the snapshots:

```
python profile.py
```

You can run the problem with the SPH version of the code as well.

A sample density profile is shown in the picture below. The results obtained using SWIFT GIZMO are plotted as black dots, the analytical solution is shown as a full red line.

Try increasing the resolution to obtain a better match.



Scaling experiments

One node performance analysis

Run the small cosmological volume through vTune and analyse the bottlenecks.

1) Get the ICs:
   `cp /work/hpclab/train001/swift_data/cosmoVolume.hdf5 CosmoVolume/`
2) Start an interactive session on JUDGE:
   `msub -I -X -l nodes=1:ppn=12,walltime=00:30:00`
3) When the session has started, move to the code directory, load the modules and compile the SWIFT code as described above.
4) Start vTune:
   `amplxe-gui &`
5) Create a new project by clicking on “New Project...”, give it a name and a location for the results (can be anywhere).
6) Set the executable to
   `swiftsim/examples/test_fixdt`
7) Set the runtime parameters to
   `-r 10 -t 8 -m 0.6 -w 5000 -d 1e-8 -f CosmoVolume/cosmoVolume.hdf5`
   with the number after “-t” giving the number of threads to use.
8) Click on “Basic Hotspots Analysis”
9) This will run the code for a few time steps and collect information
10) After the run, a general feedback window opens showing a summary of the CPU usage and efficiency.
11) The “Top-down Tree” panel lists the different functions in the code and displays the time spent in them as well as the synchronisation time.
12) Clicking on one function opens a new panel with the source code and assembly code. The time spent in the different instructions is displayed. Caveat: This does not include time to read/write from/to memory.
13) Clicking on “Tasks and Frames” displays the activity of each thread.

vTune manual: https://software.intel.com/en-us/amplifier_2015_help_lin

Alternatively, vTune can be run non-interactively using the submission script

`cp /work/hpclab/train001/swift_data/vtune_sub.sh .`

and editing the relevant parameters.

The analysis of the results is then done using amplxe-gui xxxx/yyyy.amplxe & where xxxx is the directory created by the tool and yyyy the unique tag generated by the tool.

Multi-node performance analysis

Goals:

- Run the small and big cosmological volumes with the MPI version of the code.
- Run the code with 12 MPI ranks per node vs. 1 MPI rank and 12 threads.
- Try running with more than 12 threads per node to see the effect of Hyper-threading
- Analyse the thread activity (non MPI case).

1. Get the ICs:
   ```
   cp /work/hpclab/train001/swift_data/cosmoVolume.hdf5
   CosmoVolume/
   cp /work/hpclab/train001/swift_data/bigCosmoVolume.hdf5
   BigCosmoVolume/
   ```
2. Get the submission script:
   ```
   cp /work/hpclab/train001/swift_data/cosmo_sub.sh .
   ```
3. Edit the script and submit the job:
   ```
   msub cosmo_sub.sh
   ```
   See 1st column of https://goo.gl/3aR3RC for basic MOAB batch commands.
4. Analyse the performance of the code. The last column in the output
   ```
   # Step  Time  time-step  CPU Wall-clock time [ms]
   0 1.000000e-08 1.000e-08 211214.184
   1 2.000000e-08 1.000e-08 109588.760
   ...
   ```
   gives the time (in ms) to complete a time-step.
5. Run with different combinations of MPI ranks and thread numbers to analyse the scaling of the code.
6. Plot a task graph using the python script provided.