Converting HyPar to OP2 eDSL

5665943
Charlene Ng Andrew

Introduction

- **Scientific computing** is the collection of tools, techniques and theories used to solve mathematical models on a computer in the field of science and engineering.
- They tend to be too large (to be run) for our everyday computers, hence the need for **high performance computing**.
- Mini applications are used to simulate a simplified version of a scientific computing application.
- In my dissertation project, I aim to implement OP2, an embedded Domain Specific Language (eDSL) library into HyPar, a Partial Differential Equations (PDE) Solver mini application.

[1] https://scicomp.rptu.de/about/scientific-computing/

OP2

- OP2 (Oxford Parallel library for unstructured mesh solvers) is a high-level eDSL to write unstructured mesh algorithms.
- An open-source version of OPlus, which was built for Rolls Royce 10+ years ago.
- OP2 abstracts and automates parallel execution and data movement in unstructured mesh applications.
    - Optimize memory layout
    - Fuse loops
    - Reorder data access for cache and SIMD
    - Auto-generate CUDA/OpenMP/MPI code
- Supports generating code targeting multi-core CPUs with SIMD vectorisation and OpenMP threading, many-core GPUs with CUDA or OpenMP offloading, and distributed memory cluster variants of these using MPI.

OP2

- OP2 Key Features:
  - Static mesh structures (sets, maps, and data are fixed after setup)
  - Deterministic behavior (order of operation must not affect results, except for floating-point precision)
  - Example of OP2 usage is as follows:

```c
void double_values(float *val1, float *val2) {
        *val1 = 2.0f * (*val1);
        *val2 = 2.0f * (*val2);
}

int main(int argc, char **argv) {

        op_init(argc, argv, 1);
        op_set points = op_decl_set(5, "points");
        op_set cells = op_decl_set(2, "cells");

        // Declare mapping: each cell maps to 2 points
        int cell_to_point[2 * 2] = {
                0, 1,  // Cell 0 connects to points 0 and 1
                3, 4   // Cell 1 connects to points 3 and 4
        };
        op_map pmap = op_decl_map(cells, points, 2, cell_to_point, "pmap");

        // Declare data on points
        float data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
        op_dat point_data = op_decl_dat(points, 1, "float", data,
        "point_data");

        op_par_loop(double_values, "double_values", cells,
                op_arg_dat(point_data, 0, pmap, 1, "float", OP_RW),
                op_arg_dat(point_data, 1, pmap, 1, "float", OP_RW)
        );

        op_fetch_data(point_data, data);

        printf("Doubled values:\n");
        for (int i = 0; i < 5; i++) {
                printf("%f\n", data[i]);
        }

        op_exit();
}
```

```
Doubled values:
2.000000
4.000000
3.000000
8.000000
10.000000
```

HyPar

- **HyPar**, short for **Hyp**erbolic-**Par**abolic PDE Solver, is a parallelised, high-performance solver for systems of conservation laws, such as shallow water equations, fluid dynamics, and so on.
- Written in C/C++, designed to solve PDE on structured grids using finite-difference methods.
- A mini-application that handles problems where the equations are:
    - Purely hyperbolic (like inviscid fluid flow, shocks, waves)
    - Hyperbolic-parabolic (like viscous flows, heat diffusion)

HyPar (Numerical Method)

- Define PDE : ∂u/∂t = F_hyp(u) + F_par(u) + F_sou(u)
- Spatial Discretization
  - Discretize Hyperbolic Term (Upwind schemes, finite differences)
  - Discretize Parabolic Term (Diffusion schemes)
  - Evaluate Source Term (Direct function evaluation)
- Now we have : du/dt = F(u)
  - Becomes an ODE (Ordinarily Differential Equation)
- Choose Time Integration Method
  - Native Explicit Integrators
    - Forward Euler
    - Runge-Kutta (e.g., RK4)
  - PETSc Time Integrators
    - Implicit methods (Backward Euler, Crank-Nicholson)
    - IMEX (split hyperbolic explicit, parabolic implicit)
- March Forward in Time
- Update Solution u(t,x)
- Check : Done yet?
  - No → Go back to Time Marching
  - Yes → Finish

$$\mathbf{F}_{\text{hyp}}(\mathbf{u}) = -\sum_{d=0}^{D-1} \frac{\partial \mathbf{f}_d(\mathbf{u})}{\partial x_d} \approx -\sum_{d=0}^{D-1} \frac{1}{\Delta x_d}\left(\hat{\mathbf{f}}_{d,j+1/2} - \hat{\mathbf{f}}_{d,j-1/2}\right)$$

$$\mathbf{F}_{\text{par}}(\mathbf{u}) = \sum_{d=0}^{D-1} \frac{\partial^2 \mathbf{g}_d(\mathbf{u})}{\partial x_d^2}$$

$$\mathbf{F}_{\text{par}}(\mathbf{u}) = \sum_{d1=0}^{D-1}\sum_{d2=0}^{D-1} \frac{\partial^2 h_{d1,d2}(\mathbf{u})}{\partial x_{d1}\partial x_{d2}}$$

HyPar vanilla code structure

1) **START**
2) Initialize MPI
3) Detect simulation type
   i) Single / Ensemble / Sparse Grids
4) Create Simulation object
5) Simulation Setup
   i) Define simulation
   ii) Read Inputs
   iii) Initialize arrays, grid, initial solution
   iv) Setup boundaries and physics

- Single
  - One grid, one set of initial conditions, one set of parameters
  - One set of u on one grid, evolved over time
- Ensemble
  - Multiple problems with
    - Different initial conditions, parameters, etc
    - Each ensemble member would have its own u and own time evolution.
- Sparse Grids
  - Instead of a grid, solve on reduced, smartly-chosen points
  - Adaptively select points where u is interesting or changing rapidly and avoid wasting effort on smooth areas

HyPar vanilla code structure

1) **START**
2) Initialize MPI
3) Detect simulation type
   i) Single / Ensemble / Sparse Grids
4) Create Simulation object
5) Simulation Setup
   i) Define simulation
   ii) Read Inputs
   iii) Initialize arrays, grid, initial solution
   iv) Setup boundaries and physics
6) Wrap up initialization
7) Start Timer
8) Solve Simulation
9) Stop Timer
10) Calculate and Write Errors / Runtimes
11) Cleanup: delete simulation, free/exit MPI
12) **END**

```
HyPar - Parallel (MPI) version with 1 processes
Allocated simulation object(s).
Reading solver inputs from file "solver.inp".
  No. of dimensions                          : 1
  No. of variables                           : 2
  Domain size                                : 500
  Processes along each dimension             : 1
  Exact solution domain size                 : 500
  No. of ghosts pts                          : 3
  No. of iter.                               : 600
  Restart iteration                          : 0
  Time integration scheme                    : rk (44)
  Spatial discretization scheme (hyperbolic) : weno5
  Split hyperbolic flux term?                : no
  Interpolation type for hyperbolic term     : characteristic
  Spatial discretization type   (parabolic ) : nonconservative-1stage
  Spatial discretization scheme (parabolic ) : 2
  Time Step                                  : 1.000000E-01
  Check for conservation                     : no
  Screen output iterations                   : 10
  File output iterations                     : 150
  Initial solution file type                 : ascii
  Initial solution read mode                 : serial
  Solution file write mode                   : serial
  Solution file format                       : text
  Overwrite solution file                    : no
  Physical model                             : shallow-water-1d
Partitioning domain and allocating data arrays.
Reading array from ASCII file initial.inp (Serial mode).
Volume integral of the initial solution:
 0:  2.3265000000000000E+04
 1:  0.0000000000000000E+00
Reading boundary conditions from boundary.inp.
  Boundary                  extrapolate:  Along dimension  0 and face +1
  Boundary                  extrapolate:  Along dimension  0 and face -1
2 boundary condition(s) read.
Initializing solvers.
Reading WENO parameters from weno.inp.
Initializing physics. Model = "shallow-water-1d"
Reading physical model inputs from file "physics.inp".
Reading array from ASCII file topography.inp (Serial mode).
Setting up time integration.
Solving in time (from 0 to 600 iterations)
Writing solution file topography_00000.dat.
Writing solution file op_00000.dat.
iter=     10  t=1.000E+00  CFL=4.670E-01  norm=2.8767E-01  wctime: 9.6E-04 (s)
iter=     20  t=2.000E+00  CFL=4.670E-01  norm=2.6943E-01  wctime: 1.0E-03 (s)
iter=     30  t=3.000E+00  CFL=4.670E-01  norm=2.5614E-01  wctime: 1.1E-03 (s)
iter=     40  t=4.000E+00  CFL=4.670E-01  norm=2.5099E-01  wctime: 1.2E-03 (s)
iter=     50  t=5.000E+00  CFL=4.670E-01  norm=2.4637E-01  wctime: 1.2E-03 (s)
iter=     60  t=6.000E+00  CFL=4.670E-01  norm=2.4496E-01  wctime: 1.4E-03 (s)
iter=     70  t=7.000E+00  CFL=4.670E-01  norm=2.4145E-01  wctime: 1.3E-03 (s)
iter=     80  t=8.000E+00  CFL=4.670E-01  norm=2.3927E-01  wctime: 1.2E-03 (s)
```

```
iter=    180  t=1.800E+01  CFL=4.697E-01  norm=2.8565E-01  wctime: 1.4E-03 (s)
iter=    190  t=1.900E+01  CFL=4.912E-01  norm=3.4897E-01  wctime: 1.1E-03 (s)
iter=    200  t=2.000E+01  CFL=4.877E-01  norm=3.5270E-01  wctime: 1.0E-03 (s)
iter=    210  t=2.100E+01  CFL=4.869E-01  norm=3.3387E-01  wctime: 1.1E-03 (s)
iter=    220  t=2.200E+01  CFL=4.865E-01  norm=3.2480E-01  wctime: 1.1E-03 (s)
iter=    230  t=2.300E+01  CFL=4.864E-01  norm=3.1428E-01  wctime: 1.2E-03 (s)
iter=    240  t=2.400E+01  CFL=4.864E-01  norm=3.1298E-01  wctime: 9.9E-04 (s)
iter=    250  t=2.500E+01  CFL=4.880E-01  norm=3.0976E-01  wctime: 9.7E-04 (s)
iter=    260  t=2.600E+01  CFL=4.896E-01  norm=3.0888E-01  wctime: 9.7E-04 (s)
iter=    270  t=2.700E+01  CFL=4.910E-01  norm=3.1105E-01  wctime: 1.0E-03 (s)
iter=    280  t=2.800E+01  CFL=4.920E-01  norm=3.1016E-01  wctime: 1.1E-03 (s)
iter=    290  t=2.900E+01  CFL=4.924E-01  norm=3.1487E-01  wctime: 1.2E-03 (s)
iter=    300  t=3.000E+01  CFL=4.925E-01  norm=3.1195E-01  wctime: 9.7E-04 (s)
Writing solution file topography_00002.dat.
Writing solution file op_00002.dat.
iter=    310  t=3.100E+01  CFL=4.925E-01  norm=3.1660E-01  wctime: 9.7E-04 (s)
iter=    320  t=3.200E+01  CFL=4.925E-01  norm=3.1377E-01  wctime: 1.3E-03 (s)
iter=    330  t=3.300E+01  CFL=4.925E-01  norm=3.1739E-01  wctime: 1.1E-03 (s)
iter=    340  t=3.400E+01  CFL=4.925E-01  norm=3.1420E-01  wctime: 9.7E-04 (s)
iter=    350  t=3.500E+01  CFL=4.925E-01  norm=3.1569E-01  wctime: 1.1E-03 (s)
iter=    360  t=3.600E+01  CFL=4.925E-01  norm=3.1452E-01  wctime: 9.7E-04 (s)
iter=    370  t=3.700E+01  CFL=4.925E-01  norm=3.1501E-01  wctime: 9.8E-04 (s)
iter=    380  t=3.800E+01  CFL=4.925E-01  norm=3.1525E-01  wctime: 9.7E-04 (s)
iter=    390  t=3.900E+01  CFL=4.925E-01  norm=3.1294E-01  wctime: 1.0E-03 (s)
iter=    400  t=4.000E+01  CFL=4.925E-01  norm=3.1506E-01  wctime: 9.7E-04 (s)
iter=    410  t=4.100E+01  CFL=4.925E-01  norm=3.1230E-01  wctime: 9.7E-04 (s)
iter=    420  t=4.200E+01  CFL=4.925E-01  norm=3.1563E-01  wctime: 9.9E-04 (s)
iter=    430  t=4.300E+01  CFL=4.925E-01  norm=3.1105E-01  wctime: 1.1E-03 (s)
iter=    440  t=4.400E+01  CFL=4.925E-01  norm=3.1483E-01  wctime: 9.7E-04 (s)
iter=    450  t=4.500E+01  CFL=4.925E-01  norm=3.1141E-01  wctime: 9.7E-04 (s)
Writing solution file topography_00003.dat.
Writing solution file op_00003.dat.
iter=    460  t=4.600E+01  CFL=4.925E-01  norm=3.1506E-01  wctime: 9.8E-04 (s)
iter=    470  t=4.700E+01  CFL=4.925E-01  norm=3.1180E-01  wctime: 1.1E-03 (s)
iter=    480  t=4.800E+01  CFL=4.925E-01  norm=3.1361E-01  wctime: 1.1E-03 (s)
iter=    490  t=4.900E+01  CFL=4.925E-01  norm=3.1265E-01  wctime: 9.6E-04 (s)
iter=    500  t=5.000E+01  CFL=4.925E-01  norm=3.1337E-01  wctime: 9.7E-04 (s)
iter=    510  t=5.100E+01  CFL=4.925E-01  norm=3.1364E-01  wctime: 9.6E-04 (s)
iter=    520  t=5.200E+01  CFL=4.926E-01  norm=3.1146E-01  wctime: 9.8E-04 (s)
iter=    530  t=5.300E+01  CFL=4.936E-01  norm=3.1376E-01  wctime: 9.7E-04 (s)
iter=    540  t=5.400E+01  CFL=4.945E-01  norm=3.1108E-01  wctime: 1.1E-03 (s)
iter=    550  t=5.500E+01  CFL=4.953E-01  norm=3.1451E-01  wctime: 9.8E-04 (s)
iter=    560  t=5.600E+01  CFL=4.959E-01  norm=3.0995E-01  wctime: 1.2E-03 (s)
iter=    570  t=5.700E+01  CFL=4.963E-01  norm=3.1380E-01  wctime: 1.1E-03 (s)
iter=    580  t=5.800E+01  CFL=4.965E-01  norm=3.1011E-01  wctime: 1.2E-03 (s)
iter=    590  t=5.900E+01  CFL=4.965E-01  norm=3.1317E-01  wctime: 1.0E-03 (s)
iter=    600  t=6.000E+01  CFL=4.965E-01  norm=1.8102E-01  wctime: 9.7E-04 (s)
Completed time integration (Final time: 60.000000), total wctime: 0.644403 (seconds).
Writing solution file topography_00004.dat.
Writing solution file op_00004.dat.
Solver runtime (in seconds): 6.4894099999999999E-01
Total  runtime (in seconds): 6.4944700000000000E-01
Deallocating arrays.
Finished.
```

Snippet of the output of running HyPar for the first time on the 1D Shallow Water example

Motivation

- Why OP2?
    - Abstracts parallelism
    - Handles data movement
    - Portable across CPUs & GPUs
    - Modern + scalable: Both MPI and OpenMP needs to be manually declared to determine how to parallelise each lines of code, but OP2 will automatically assign/declare them.
- HyPar is written entirely in C/C++ and uses the MPICH library. **OpenMP and CUDA are a work in progress**.
    - Implementing OP2 instead of the OpenMP and MPI = aids in modernising the codebase and abstracting away low-level parallelism (OpenMP, MPI)
    - Which means it will be easier to maintain and be highly portable to different architectures - including CUDA and OpenMP.

```
int main(int argc, char** argv) {
        MPI_Init(&argc, &argv);  // Initialize MPI

        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get process rank
        MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get number of processes

        int number = rank + 1; // Each process has a number (rank + 1)

        // --- SEND and RECV ---
        if (rank == 0) {
          int received_number;
          MPI_Recv(&received_number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
          printf("Process 0 received %d from process 1\n", received_number);
        }
        else if (rank == 1) {
          MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
          printf("Process 1 sent %d to process 0\n", number);
        }

        // --- REDUCE ---
        int sum;
        MPI_Reduce(&number, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        if (rank == 0) {
          printf("Sum of all numbers = %d\n", sum);
        }

        MPI_Finalize(); // Finalize MPI
        return 0;
}
```

Related Works

- OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures
    - Paper written by G. R. Mudalige, M.B. Giles, I. Reguly, C. Bertolli and PH.J Kelly in May 2012.
    - The paper talks about the implementation of OP2
        - Allows developers to write application code that can be automatically transformed into parallel implementations for various back-end platforms.
    - Tests its implementation of OP2 onto a mini-application called Airfoil.
    - Results showed significant performance improvements on GPU clusters for larger meshes and highlighted the influence of data layout and partition sizes on performance

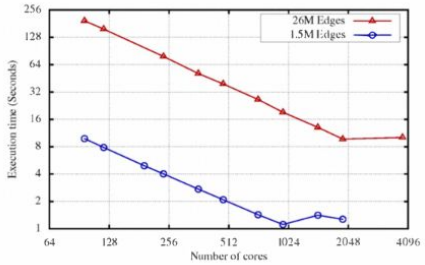


Figure 7: Airfoil strong scaling on HECToR (1.5M and 26M edges)

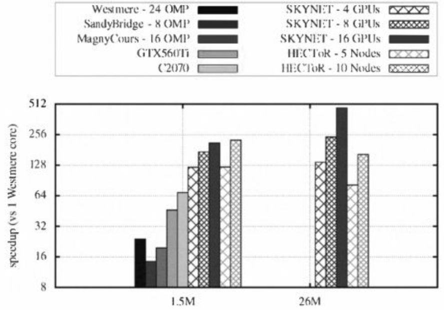


Figure 8: Airfoil speedups summary

Related Works

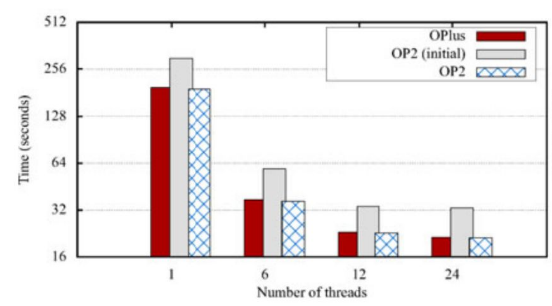
- Airfoil
  - A mini-application written by Mike Giles in 2010-2011, based on FORTRAN code by Devendra Ghate and Mike Giles in 2005
  - A finite volume application that solves the 2D Euler equations using a scalar numerical dissipation written in C++.
  - Downloadable with the OP2 Github repo to demonstrate OP2's usage.

```
 Initializing OP2
reading in grid
initialising flow field

  OP diagnostic output
  --------------------

       set       size
  -------------------
     nodes     721801
     edges    1438600
    bedges       2800
     cells     720000

       map        dim       from         to
  -----------------------------------------
     pedge          2      edges      nodes
    pecell          2      edges      cells
    pbedge          2     bedges      nodes
   pbecell          1     bedges      cells
     pcell          4      cells      nodes

       dat        dim        set
  ------------------------------
   p_bound          1     bedges
       p_x          2      nodes
       p_q          4      cells
    p_qold          4      cells
     p_adt          1      cells
     p_res          4      cells

 100  5.02186e-04
 200  3.41746e-04
 300  2.63430e-04
 400  2.16288e-04
 500  1.84659e-04
 600  1.60866e-04
 700  1.42253e-04
 800  1.27627e-04
 900  1.15810e-04
 1000  1.06011e-04


Test problem with 720000 cells is within 7.389644451905042E-12 % of the expected solution
This test is considered PASSED
Max total runtime = 81.340277
```

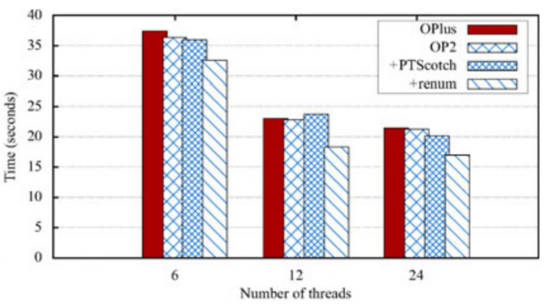
Output of airfoil_seq

Related Works

- Acceleration of a Full-Scale Industrial CFD Application with OP2
    - Paper written by Istvan Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H.J. Kelly, and David Radford in May 2016
    - Hydra, a major Rolls-Royce CFD application, was ported to OP2 for efficient multi-core and many-core parallelism.
    - Result: OP2 achieved near-optimal performance, boosting speed on both conventional CPUs and many-core systems.
        - ➔ Running Hydra in a hybrid CPU-GPU setup gave up to 15% more speedup compared to GPU-only execution.



(a) OPlus vs OP2 (MPI only)



(b) OPlus vs OP2 (with PTSotch and renumbering)

Related Works

- Improving CUDA performance of an unstructured high-order CFD application under OP2 framework
  - Paper published by Kangjin Huang, Yonggang Che, Chuanfu Xu, Zhe Dai, Jian Zhang in October 2023.
  - Using OP2 to improve the CUDA performance of an application called HOUR2D using optimisation methods (shown on the bottom image)
  - HOUR2D is a high-order, unstructured CFD application - solving steady/unsteady Navier-Stokes and Euler equations.
  - Result: The optimized OP2-CUDA version outperforms the manual CUDA version by a factor of 2.4 times.

Table 5 Costs of kernel functions and API calls of manual CUDA version and OP2-CUDA version for CASE1 (s)

Type	Function	Manual	OP2-Unoptimized	OP2-optimized
GPU activities	rhscal_auxvar()	8.12	66.69	2.14
GPU activities	rhscal()	3.56	6.04	1.75
GPU activities	rhscal_calsideflux()	1.75	1.85	1.54
GPU activities	CUDA memcpy DtoH	6.08	21.10	0.95
API calls	cudaDeviceSynchronize	14.98	76.92	6.62
API calls	cudaMemcpy	6.17	21.38	1.02

Performance test output

Table 4 Runtime for 1000 time-steps of different OP2-based versions on the CPU and GPU (s)

	CASE1			CASE2		
Code versions	Serial	OpenMP	CUDA	Serial	OpenMP	CUDA
OP2-unoptimized	357.6	52.1	118.6	1475.7	206.6	534.7
Data race avoiding strategies	357.6	52.1	55.2	1475.7	206.6	443.9
Data transfer optimization	338.9	24.6	31.9	1414.0	106.5	328.9
Using local arrays	345.5	25.3	11.5	1435.4	102.0	58.0
Other optimizations	345.5	25.3	9.0	1435.4	102.0	43.4

Optimisation methods used in the paper

Current Progress

1) Setting HyPar and OP2 on my local machine and VSCode.
2) Understanding and reading HyPar and OP2's documentation.
3) Test run the Airfoil application.
4) Run the vanilla HyPar mini-application without OP2 implementation.
5) At the initial stage of implementing OP2 into HyPar

output.txt **VSCode**

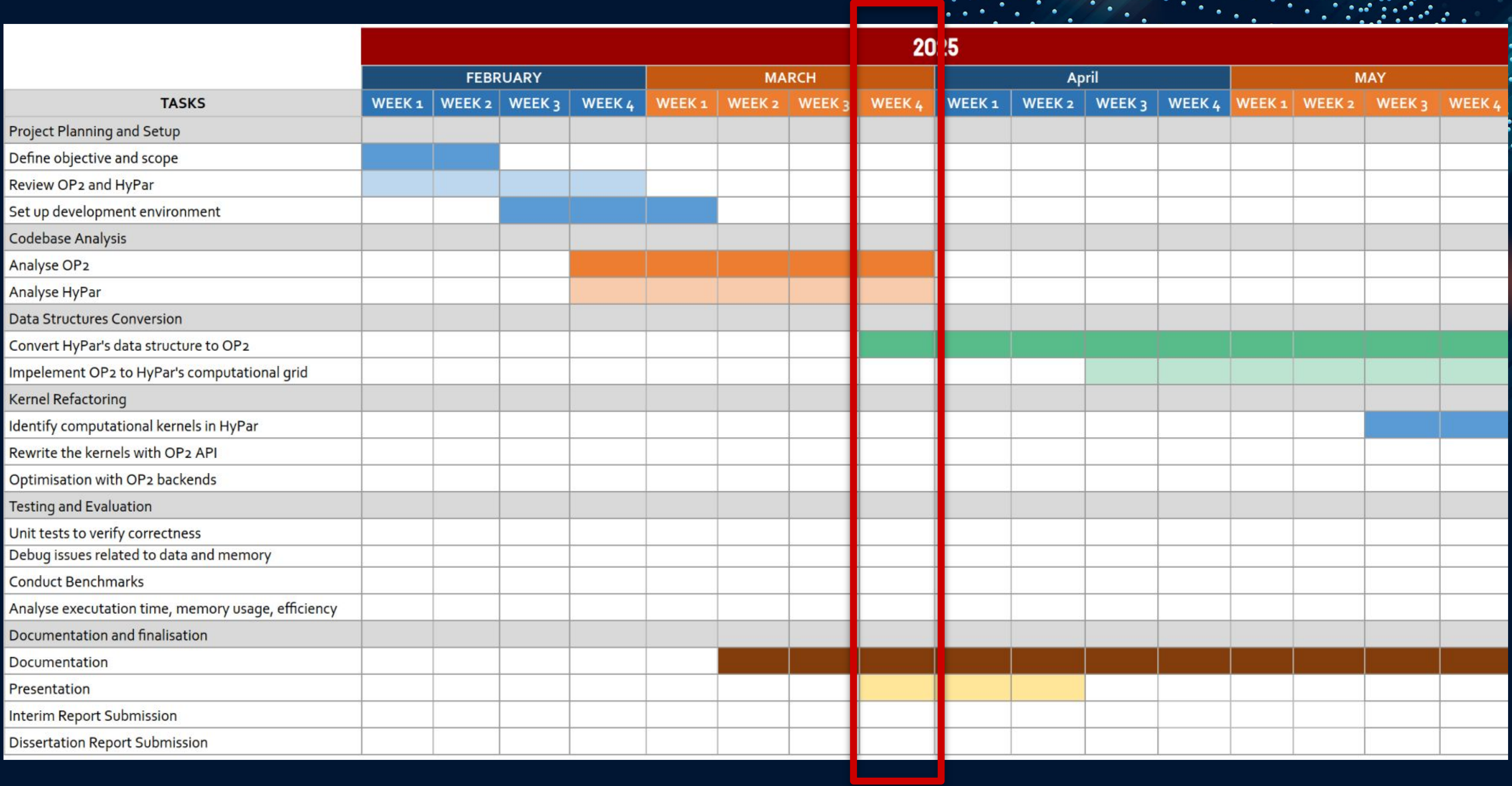
OP2's command printing out as expected

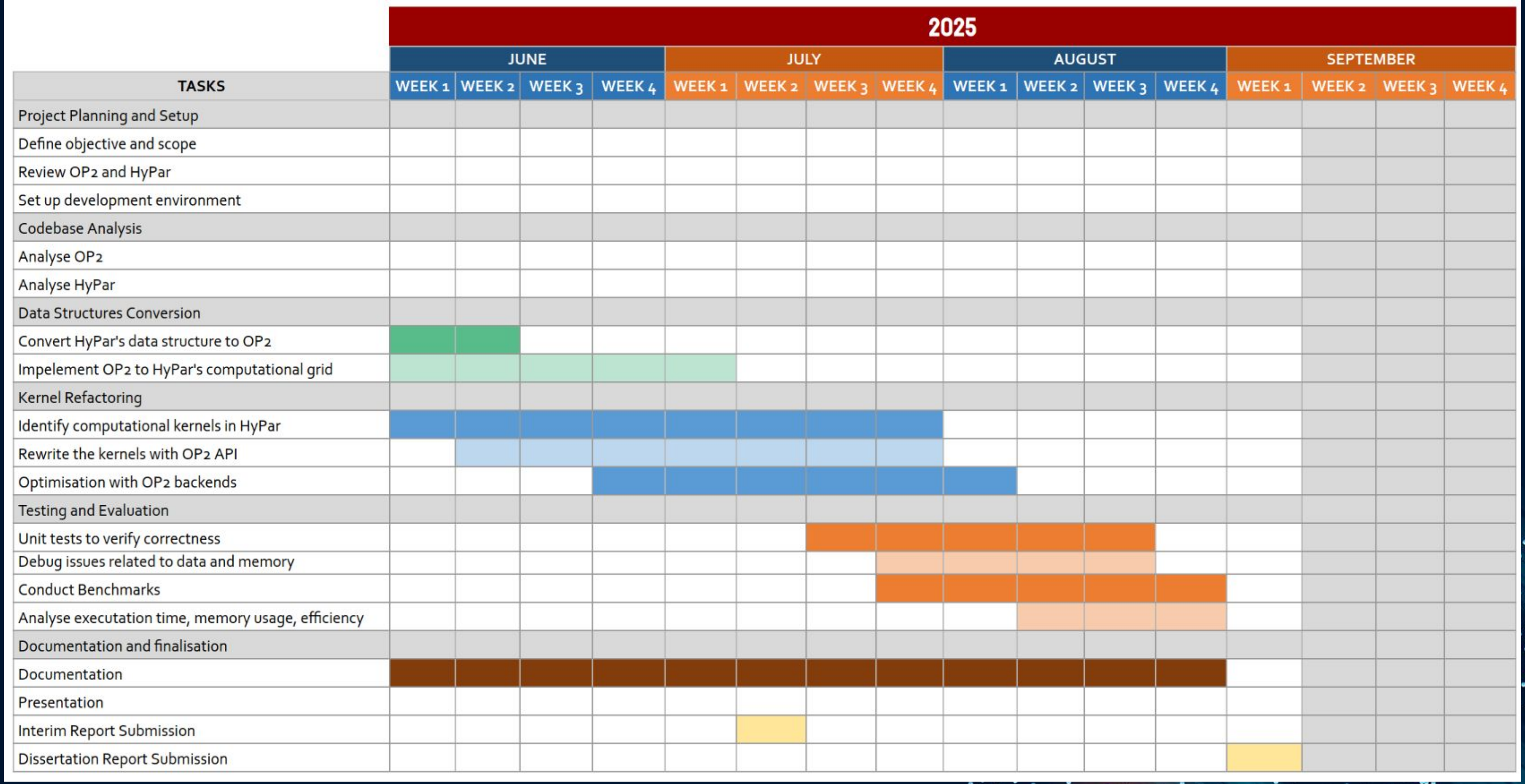
Next steps

- Go through one loop at a time
    - `InitializeSolvers(), InitializePhysics()`
- Pick up on the data structure of HyPar
    - The arrangement and calls of the arrays, grids and loops
- Convert each loop to OP2 API
- Check output against the original
- Fully converted sequential version
- Initiate parallelisation
- Test and benchmark
    - Each parallelised loops VS the vanilla HyPar

Timeline

A Gantt Chart of the timeline I aim to follow is in the following slides →


2025
FEBRUARY
MARCH
April
MAY
TASKS
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
Project Planning and Setup
Define objective and scope
Review OP2 and HyPar
Set up development environment
Codebase Analysis
Analyse OP2
Analyse HyPar
Data Structures Conversion
Convert HyPar's data structure to OP2
Impelement OP2 to HyPar's computational grid
Kernel Refactoring
Identify computational kernels in HyPar
Rewrite the kernels with OP2 API
Optimisation with OP2 backends
Testing and Evaluation
Unit tests to verify correctness
Debug issues related to data and memory
Conduct Benchmarks
Analyse executation time, memory usage, efficiency
Documentation and finalisation
Documentation
Presentation
Interim Report Submission
Dissertation Report Submission


2025
JUNE
JULY
AUGUST
SEPTEMBER
TASKS
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
WEEK 1
WEEK 2
WEEK 3
WEEK 4
Project Planning and Setup
Define objective and scope
Review OP2 and HyPar
Set up development environment
Codebase Analysis
Analyse OP2
Analyse HyPar
Data Structures Conversion
Convert HyPar's data structure to OP2
Impelement OP2 to HyPar's computational grid
Kernel Refactoring
Identify computational kernels in HyPar
Rewrite the kernels with OP2 API
Optimisation with OP2 backends
Testing and Evaluation
Unit tests to verify correctness
Debug issues related to data and memory
Conduct Benchmarks
Analyse executation time, memory usage, efficiency
Documentation and finalisation
Documentation
Presentation
Interim Report Submission
Dissertation Report Submission

Conclusions

- OP2 (Oxford Parallel library for unstructured mesh solvers) is a high-level eDSL to write unstructured mesh algorithms.
- **HyPar**, short for **Hyp**erbolic-**Par**abolic PDE Solver, is a parallelised, high-performance solver for systems of conservation laws, such as shallow water equations, fluid dynamics, and so on.
- Motivation: Implementing OP2 into HyPar aids in modernising the codebase and abstracting away low-level parallelism (OpenMP, MPI).
- Related works: A paper on OP2, OP2 on a large-scale application, and a very recent published paper on using OP2 to improve CUDA performance.
- Current progress: Implementation progress, deeper understanding of OP2 and HyPar.
- Next steps: Implementation of OP2 in the loops, get a working & accurate sequential version of HyPar up, initiate parallelisation, benchmarking.