# High-level Abstractions for Performance, Portability and Continuity of Hydrocodes on Future Computing Systems

**G.R. Mudalige , I. Z. Reguly , M.B. Giles.**

Oxford e-Research Centre, University of Oxford

# OUTLINE

- Motivation
  - Future proofing parallel HPC applications
  - Previous work
- OPS : A High-level abstraction for Structured mesh applications
  - The OPS API
  - Porting AWE's Cloverleaf to the OPS abstraction
- Code generation
  - Single threaded developer version
  - Cloverleaf code generation with optimizations
    - Optimized single threaded (with AVX vectorization)
    - OpenMP, CUDA and MPI
    - Hybrid – MPI+OpenMP
  - Lessons learnt
- Cloverleaf Performance - Sequential, OpenMP, CUDA, MPI and Hybrid
- Future work and Conclusions

# FUTURE PROOFING PARALLEL HPC APPLICATIONS

- High performance computing (HPC) is currently in a period of enormous change
- Emerging Many-core architectures roadmap
  - Mainstream CPUs with larger vector units vector units key to gaining higher performance
  - NVIDIA GPUs and AMD GPUs
  - Intel MIC - Integrated x86 multi-processor with SIMD units (Xeon Phi)
  - Energy efficient designs from ARM
    E.g. Mont-Blanc Project - ARM processors integrated on to a GPU
  - Other Heterogeneous processors – FPGAs, DSP type accelerators

# FUTURE PROOFING PARALLEL HPC APPLICATIONS

- Hardware is rapidly changing with ambitions to overcome exascale challenges
  - Massive parallelism due to single thread performance limitations – Billion-way threads ?
  - Limited memory bandwidth – Memory wall
  - Power efficiency – communications-centric architectures to reduce energy consumption ?
  - Resilience / Fault-tolerance
- There is considerable uncertainty about which platform to target
  - Not clear which architectural approach is likely to ``win'' in the long-term,
  - Not even clear in the short-term which platform is best for each application.

# FUTURE PROOFING PARALLEL HPC APPLICATIONS

- Increasingly complex programming skills set needed to extract best performance for your workload on the newest architectures.
  - Need a lot of platform specific knowledge
  - Not clear which architectural approach is likely to ``win'' in the long-term,
  - Cannot be re-coding applications for each “new” type of architecture or parallel system.

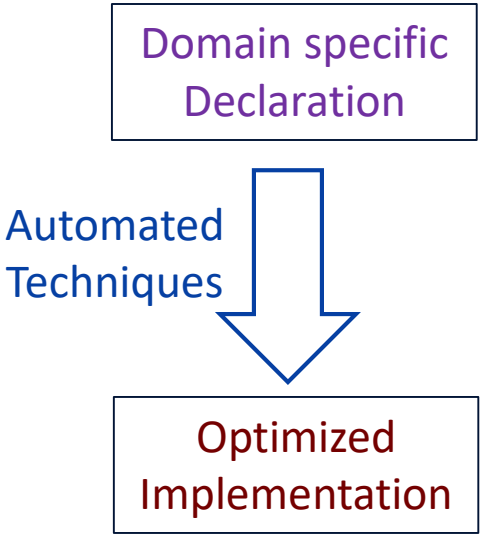
- Currently the common approach for utilizing novel hardware is to
  - Manually port legacy applications
    Usually converting key kernels of the application
  - Complete re-write of existing code to gain performance
    e.g. to reduce communications
  - Maintain several “efficient” versions of the source code

The development of HPC applications designed to ``future proof'' their continued performance and portability on a diverse range of hardware and future emerging systems is of critical importance.

# Domain Specific High-level Abstractions

One such approach, is the use of domain specific high-level abstractions (HLAs)

- Provide the application developer with a domain specific abstraction
  - To declare the problem to be computed
  - Without specifying its implementation
  - Use domain specific constructs in the declaration
- Provide a lower implementation level
  - To apply automated techniques for translating the specification to different implementations
  - Target different hardware and software platforms
  - Exploit domain knowledge for better optimisations on each hardware system



# PREVIOUS WORK

- The use of HLAs have previously shown to have significant benefits
  - For developer productivity
  - Gaining near-optimal performance

- Several Flavours
  - Domain Specific Language (DSL)
  - Embedded in general purpose host language (usually as a domain specific API)
  - Compiler Extension

- Example HLAs
  - Structured mesh – Pochoir (MIT), SBLOCK (Cambridge), STELLA (MetroSwiss)
  - Unstructured mesh – OP2 (Oxford ), Liszt (Stanford)

# Motivations and Contributions

- HLAs still remain as experimental research projects and have not yet been adopted by a wider HPC community.
  - A lack of HLAs that are actively used at creating production level applications.
  - Existing frameworks are only applied to a few application domains
- We need to further explore the utility of high-level abstraction frameworks
  - From a range of application domains
  - For industry representative and production-grade applications
- In this work we focus on hydro-dynamics applications
  - Important class of codes forming a key part of the HPC workload at AWE
  - Use of a previously developed mini-application called Cloverleaf that implements algorithms of interest
  - We explore the pros and cons of re-engineering this code with an HLA framework.

# CLOVERLEAF

- "Mini-app" – Representative, but lightweight application
  - Small (~6K LoC) – compared to a production application (~0.5M LoC)

- Open source software - part of the Sandia NL's Mantevo project (https://software.sandia.gov/mantevo/)

- Have been ported to execute on many of the current parallel hardware using a range of parallelizations

  - Single-instruction-multiple-data (SIMD) - SSE and AVX

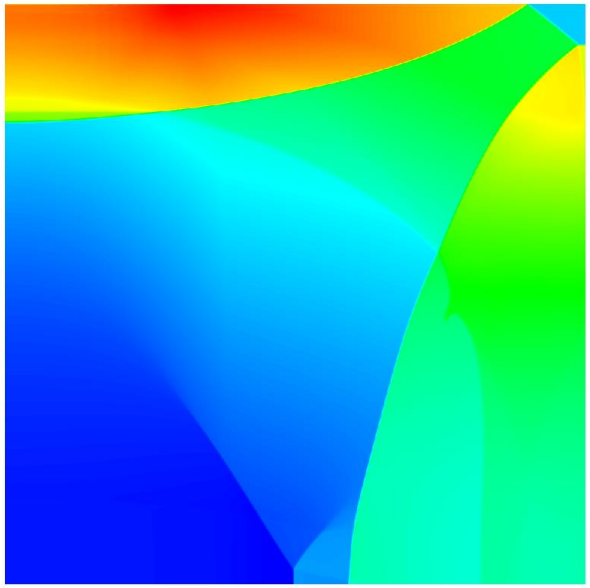
  - Shared memory multi-threading for multi-core CPUs - OpenMP

  - Single-instruction-multiple-thread (SIMT) - CUDA, Open-CL and OpenACC for GPUs and the Intel's Xeon Phi

  - Distributed memory parallelization (MPI) for clusters of CPUs/GPUs.

# CLOVERLEAF – NUMERICAL METHOD

- 2D Structured Hydrodynamics
- Explicit solution to the compressible Euler equations
- Finite volume predictor/corrector Lagrangian step followed by an advective remap
- Single material
- Common base to all interested physics models at AWE



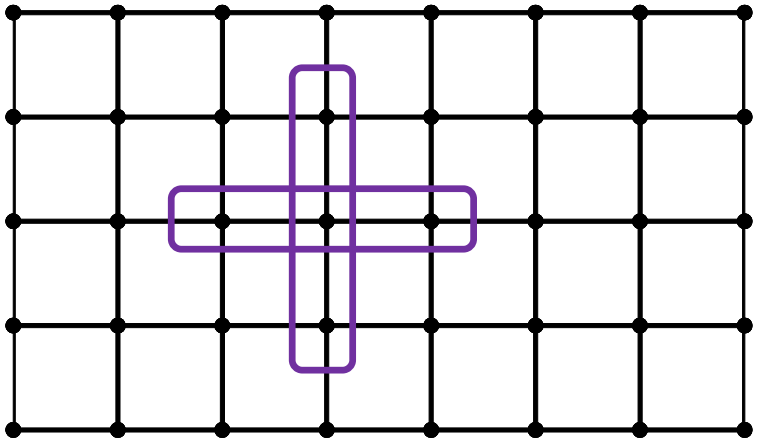
# CONTRIBUTIONS

- Re-engineer Cloverleaf to utilize a domain specific high-level abstraction framework, called OPS (Oxford Parallel Library for Structured-mesh solvers)
  - Results in a single high-level application source
- Use automated code generation techniques of OPS to generate a range of parallel implementations.
- Compare the resulting parallel applications to that of the original hand-tuned Cloverleaf applications.
  - Developer productivity – how difficult is it to future-proof code at design ?
  - Performance on an extensive range of contrasting architectures and systems
- Aim is to demonstrate to a wider HPC audience how HLAs might help in solving various scientific simulation challenges on future emerging systems.

# Structured Mesh Applications and OPS

- Structured grids
  - Logical *( i , j )* indexing in 2D, *(i, j, k)* in 3D, with <u>implicit</u> connectivity
  - Easy to parallelize, including on GPUs with L1/L2 caches

- Operations involve looping over
  - a "rectangular" multi-dimensional set of grid-points
  - using one or more "stencils" to access data



# OPS API

- The abstraction breaks down the specification of the problem into five parts
  1. The structured mesh (or block ) worked on
     - For single structured mesh (as in Cloverleaf) there is only one block
     - Plans to extend to multi-block in the EPSRC project
  2. Data defined on a block
     - Holds the data associated with each block
  3. Stencil
     - Defines the stencil that is used in a computation
  4. Loop over mesh
     - Defines the loop over a block
     - Indicates the data used in the loop, the stencil used to access this data and access type – read only, write only read/write, increment
  5. User Kernel – declared as an out-lined C function
     - Defines the elemental operation to be performed for each iteration of the loop
     - Declared as an out-lined kernel

# Porting Cloverleaf to OPS – Declaring Block and Data

```
/* Declare a single structured block */
ops_block cl_grid = ops_decl_block(2, dims, "clover grid");


int size[2] = {x_cells+5, y_cells+5};
double* dat = NULL;

/* Declare data on block */
ops_dat density0 = ops_decl_dat(cl_grid,1,size,dat,"double","density0");
ops_dat energy0  = ops_decl_dat(cl_grid,1,size,dat,"double","energy0");
…
…
ops_dat pressure = ops_decl_dat(cl_grid,1,size,dat,"double","pressure");
ops_dat volume   = ops_decl_dat(cl_grid,1,size,dat,"double","volume");
```

# Porting Cloverleaf to OPS – Original Loop

```
DO k=y_min,y_max
   DO j=x_min,x_max
        v=1.0/density(j,k)
        pressure(j,k)=(1.4 - 1.0)* density(j,k)*energy(j,k)
        pressurebyenergy =(1.4-1.0)* density(j,k)
        pressurebyvolume = - density(j,k)* pressure(j,k)
        sound_speed_squared = v * v *(pressure(j,k)*
                                  pressurebyenergy- pressurebyvolume)
        soundspeed(j,k)= SQRT(sound_speed_squared)
   ENDDO
ENDDO
```

# Porting Cloverleaf to OPS – OPS Loop

The original Cloverleaf 2D application written in Fortran 90 was converted to the OPS API by manually extracting the user kernels, outlining them in header files and converting the application to the OPS's C/C++ API.

```
int rangexy_inner[] = {x_min,x_max,y_min,y_max}; //mesh execution range

/*single point stencil*/
int s2D_00[] = {0,0};
ops_stencil S2D_00 = ops_decl_stencil( 2, 1, s2D_00, "00");

/*ideal_gas loop*/
ops_par_loop(ideal_gas_kernel,clover_grid, 2, rangexy_inner,
        ops_arg_dat (density0,  S2D_00, "double", OPS_READ),
        ops_arg_dat (energy0,   S2D_00, "double", OPS_READ),
        ops_arg_dat (pressure,  S2D_00, "double", OPS_WRITE),
        ops_arg_dat (soundspeed,S2D_00, "double", OPS_WRITE));
```

# Porting Cloverleaf to OPS – OPS Loop

"User kernel" – applied to every grid point

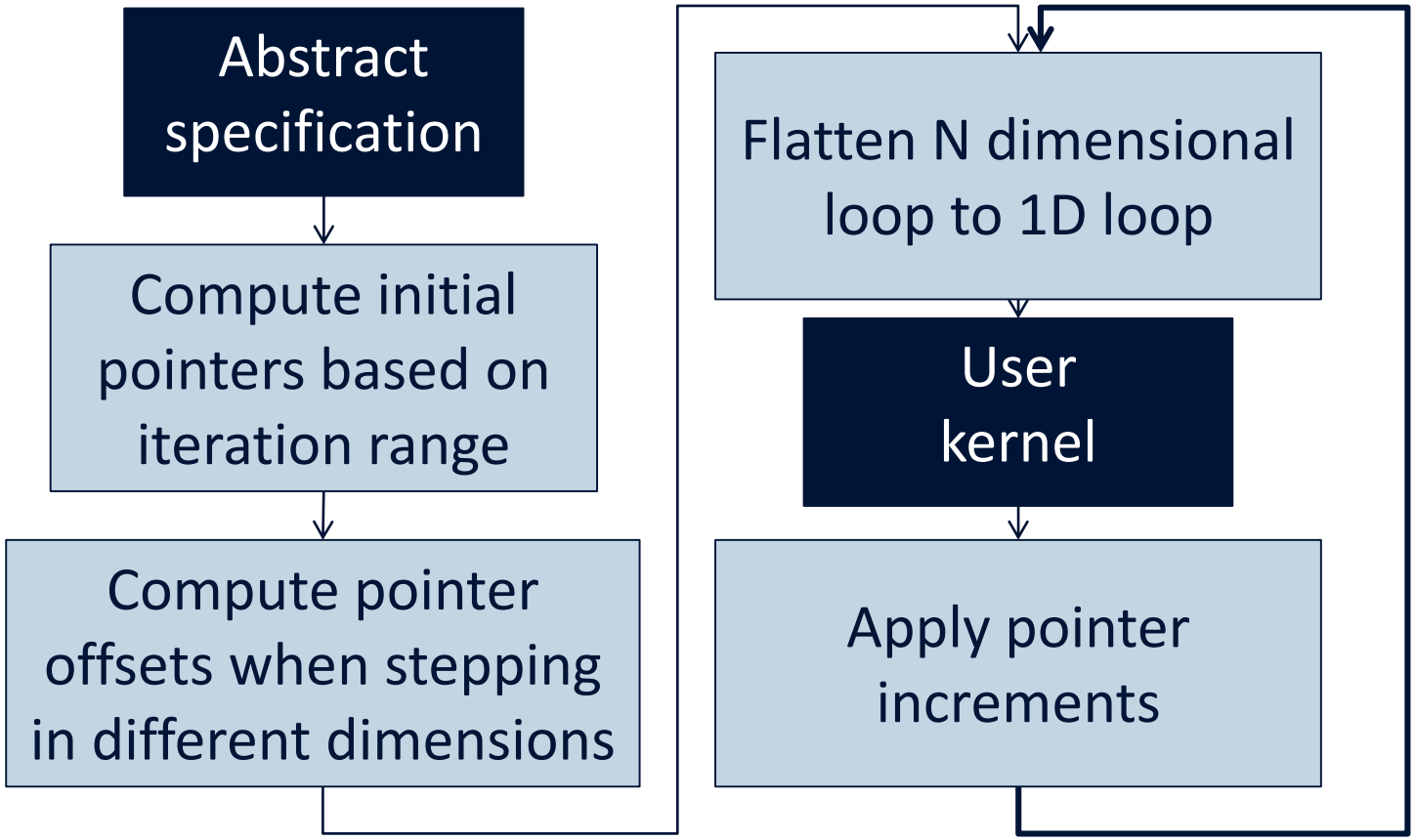
```
void ideal_gas_kernel( const double *density, const double *energy,
                       double *pressure, double *soundspeed) {

  double sound_speed_squared, v, pressurebyenergy, pressurebyvolume;

  v = 1.0 / density[OPS_ACC0(0,0)];
  pressure[OPS_ACC2(0,0)] = (1.4 - 1.0) * density[OPS_ACC0(0,0)] *
                                          energy[OPS_ACC1(0,0)];
  pressurebyenergy = (1.4 - 1.0) * density[OPS_ACC0(0,0)];
  pressurebyvolume = -1*density[OPS_ACC0(0,0)] *
                      pressure[OPS_ACC2(0,0)];
  sound_speed_squared = v*v*(pressure[OPS_ACC2(0,0)] *
                       pressurebyenergy-pressurebyvolume);
  soundspeed[OPS_ACC3(0,0)] = sqrt(sound_speed_squared);
}
```

# Separation of the description of computations from parallel execution and data movement

It is the responsibility of the library to arrange the latter two, giving us a lot of room to apply all sorts of transformations

# So what is the "magic"?

- For development: anything that calls the user kernel with the appropriate pointers to user-defined data
- For performance: anything we can think of!



# Single threaded Developer version

```
/* ops_par_loop routine for 4 arguments */
template <class T0,class T1,class T2,class T3>
void ops_par_loop(void (*kernel)(T0*, T1*, T2*, T3*),
     char const * name, ops_block block, int dim, int *range,
     ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
  ...
  ...
  for (int nt=0; nt<total_range; nt++) {
    // call kernel function, passing in pointers to data
    kernel( (T0 *)p_a[0], (T1 *)p_a[1], (T2 *)p_a[2], (T3 *)p_a[3] );

    //figure out the current dimension m
    count[0]--; // decrement counter
    int m = 0; // max dimension with changed index
    while (count[m]==0) {
      count[m] = end[m]-start[m];// reset counter
      m++; // next dimension
      count[m]--; // decrement counter
    }
    // shift pointers to data
    for (int i=0; i<4; i++) {
      if (args[i].argtype == OPS_ARG_DAT)
        p_a[i] = p_a[i] + (args[i].dat->size * offs[i][m]);
    }
  }
  ...
  ...
}
```

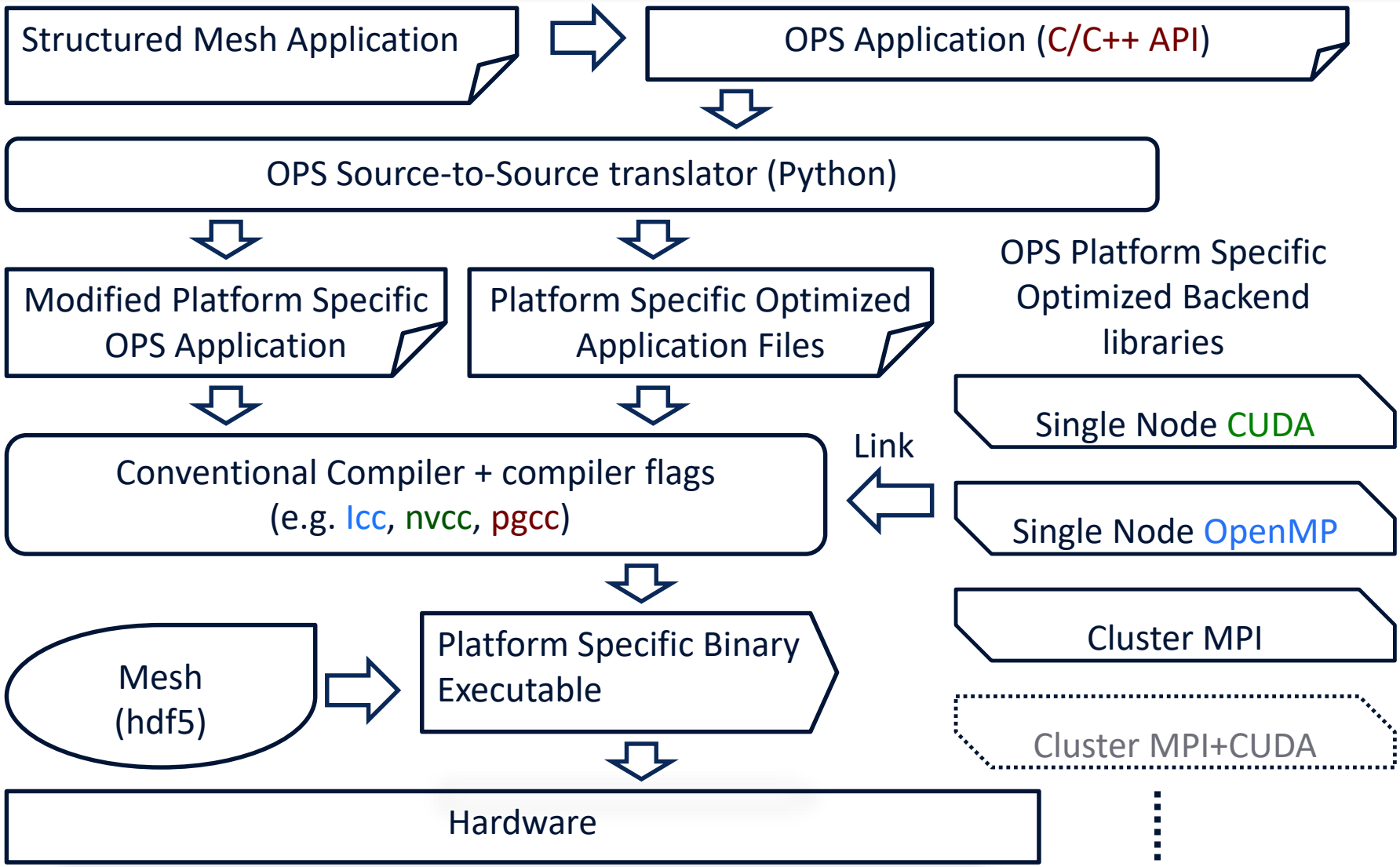
Template function

N dimensional loop flattened to 1D

Determine current dimension

Increment pointers

# PUTTING IT ALL TOGETHER



# Code Generation – Single threaded CPU Vectorization

```
void ops_par_loop_ideal_gas_kernel(char const *name, ops_block block, int dim, int* range,
 ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
  ...
  ...
  //set up initial pointers
  ...
  ...
  int n_x;
  for ( int n_y=start[1]; n_y<end[1]; n_y++ ){
    for( n_x=start[0]; n_x<start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x+=SIMD_VEC ) {
      //call kernel function, passing in pointers to data -vectorised
#pragma simd
      for ( int i=0; i<SIMD_VEC; i++ ){
        ideal_gas_kernel( (double *)p_a[0]+ i*1, (double *)p_a[1]+ i*1, (double *)p_a[2]+ i*1,
           (double *)p_a[3]+ i*1 );
      }
      //shift pointers to data x direction
      p_a[0]= p_a[0] + (dat0 * off0_1)*SIMD_VEC;
      p_a[1]= p_a[1] + (dat1 * off1_1)*SIMD_VEC;
      p_a[2]= p_a[2] + (dat2 * off2_1)*SIMD_VEC;
      p_a[3]= p_a[3] + (dat3 * off3_1)*SIMD_VEC;
    }
    for ( n_x=start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x<end[0]; n_x++ ){
        //call kernel function, passing in pointers to data - remainder
        ideal_gas_kernel( (double *)p_a[0], (double *)p_a[1], (double *)p_a[2],
           (double *)p_a[3] );
        //shift pointers to data x direction
        p_a[0]= p_a[0] + (dat0 * off0_1);
        p_a[1]= p_a[1] + (dat1 * off1_1);
        p_a[2]= p_a[2] + (dat2 * off2_1);
        p_a[3]= p_a[3] + (dat3 * off3_1);
    }
    //shift pointers to data y direction
    p_a[0]= p_a[0] + (dat0 * off0_2);
    p_a[1]= p_a[1] + (dat1 * off1_2);
    p_a[2]= p_a[2] + (dat2 * off2_2);
    p_a[3]= p_a[3] + (dat3 * off3_2);
  }
```

Outer loop

Inner loop (divisible by vector length)

Call user kernel

Increment pointers

Inner loop (remainder)

Call user kernel

Increment pointers

Increment pointers

# Code Generation – Multi-threaded OpenMP

```
#pragma omp parallel for
  for ( int thr=0; thr<nthreads; thr++ ){
    int y_size = end[1]-start[1];
    int start_i = start[1] + ((y_size-1)/nthreads+1)*thr;
    int finish_i = start[1] + MIN(((y_size-1)/nthreads+1)*(thr+1),y_size);

    //get addresss per thread
    int start0 = start[0];
    int start1 = start[1] + ((y_size-1)/nthreads+1)*thr;

    //set up initial pointers
    ...
    ...
    for ( int n_y=start_i; n_y<finish_i; n_y++ ){
      for ( int n_x=start[0]; n_x<start[0]+(end[0]-start[0])/SIMD_VEC; n_x++ ){
        //call kernel function, passing in pointers to data
#pragma simd
        for ( int i=0; i<SIMD_VEC; i++ ){
          ideal_gas_kernel((double *)p_a[0]+ i*1,(double *)p_a[1]+ i*1,(double *)p_a[2]+ i*1,
          (double *)p_a[3]+ i*1);
        }
        //shift pointers to data x direction
        ...
        ...
      }
      for ( int n_x=start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x<end[0]; n_x++ ){
        //call kernel function, passing in pointers to data - remainder
        ideal_gas_kernel((double *)p_a[0],(double *)p_a[1],(double *)p_a[2],(double *)p_a[3]);
        //shift pointers to data x direction
        ...
        ...
      }
      //shift pointers to data y direction
      ...
      ...
```

Split along the last dimension

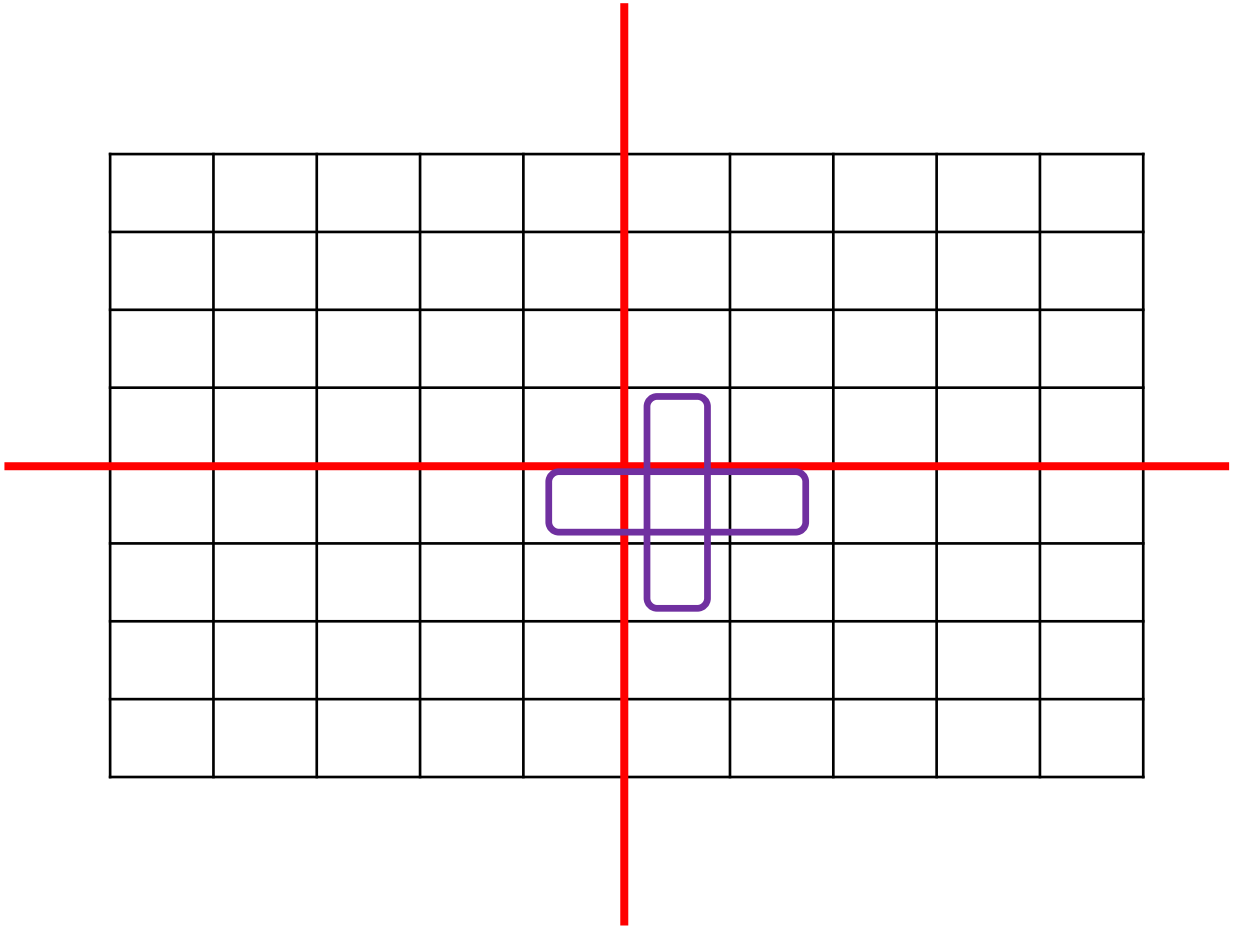
# Code Generation – GPU with NVIDIA CUDA

```
__global__ void ops_ideal_gas_kernel(
const double* __restrict arg0, const double* __restrict arg1,
      double* __restrict arg2,       double* __restrict arg3,
      int size0, int size1 ){
  int idx_y = blockDim.y * blockIdx.y + threadIdx.y;
  int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
  arg0 += idx_x * 1 + idx_y * 1 * xdim0_ideal_gas_kernel;
  arg1 += idx_x * 1 + idx_y * 1 * xdim1_ideal_gas_kernel;
  arg2 += idx_x * 1 + idx_y * 1 * xdim2_ideal_gas_kernel;
  arg3 += idx_x * 1 + idx_y * 1 * xdim3_ideal_gas_kernel;

  if (idx_x < size0 && idx_y < size1) {
    ideal_gas_kernel(arg0, arg1, arg2, arg3);
  }
}
// host stub function
void ops_par_loop_ideal_gas_kernel(char const *name, ops_block Block, int dim, int* range,
 ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
  ...
  ...
  int x_size = end[0]-start[0];
  int y_size = end[1]-start[1];
  ...
  ...
  dim3 grid( (x_size-1)/OPS_block_size_x+ 1, (y_size-1)/OPS_block_size_y + 1, 1);
  dim3 block(OPS_block_size_x,OPS_block_size_y,1);
  ...
  ...
  ops_H_D_exchanges_cuda(args, 4);

  //call kernel wrapper function, passing in pointers to data
  ops_ideal_gas_kernel<<<grid, block >>> ( (double *)p_a[0], (double *)p_a[1],
          (double *)p_a[2], (double *)p_a[3],x_size, y_size);
```

One thread per grid point

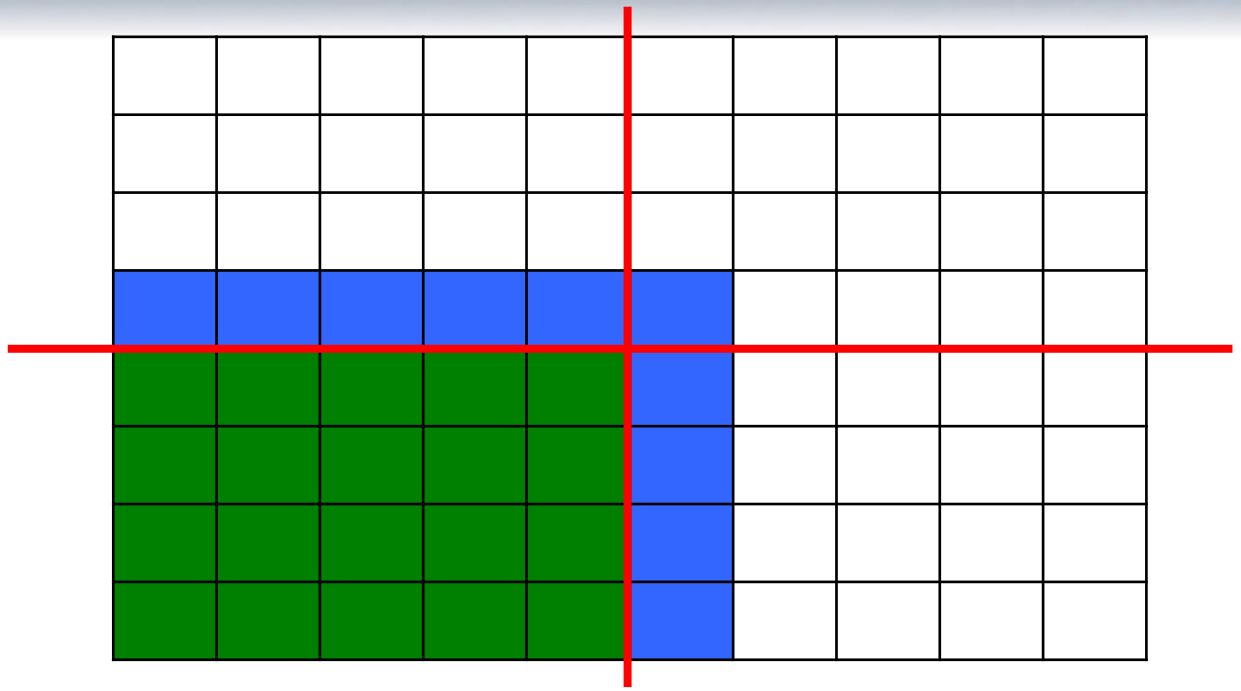
Launch CUDA kernel

# Distributed Memory with MPI



- MPI – Cartesian decompostion
- Halo regions duplicated and updated
- But how deep? We can tell based on the abstraction
- Fully automatic, the user does not have to be aware

# Distributed Memory with MPI



If iteration range is restricted, only one of the processes may execute, but it still needs data from its neighbors – every process carries out execution range and data dependency range computation and keep track of changes accordingly – minimal number of MPI messages and only when needed

# Lessons Learnt

- Iterative development:
  - Constantly trying new things, making alterations to the API – making it more intuitive to use and at the same time deliver high performance
  - The way data is passed – array of pointers based on stencil vs. a macro and a single pointer – turns out it's cleaner code and it compiles much more efficiently
- Porting a code is still an error-prone process – typos, precedence, etc.
  - Implement self-checking code:
    - Can check if stencil access matches pre-declared stencil
    - Can check if the correct macro is used for different variables
  - Dumping data for binary comparison, visualizing data
- After the 2D version, we got access to the 3D version of the same code
  - Same structure, we just had to add new kernels
  - 2 days writing code, 3 days debugging for typos – some in the reference code!
  - The real power of an HLA like OPS: introducing support for 3D in all the back-ends took very little time, but the developers of the original will have to implement and debug each and every one of those by hand!

# Performance – Benchmark Systems

| System | Broomway | Ruby | ARCHER (Cray XC30) |
|---|---|---|---|
| Node Architecture | 2×8-core Intel Xeon E5-2680 2.70GHz (Sandy bridge) | 2×Tesla K20c + 2×8-core Intel Xeon E5-2640 2.50GHz (Sandy bridge) | 2×12-core Intel Xeon E5-2697 2.70GHz (Ivy bridge) |
| Memory/Node | 64GB | 5GB/GPU (ECC off), 64GB | 64GB |
| Interconnect | - | - | Cray Aries |
| O/S | Red Hat Enterprise Linux ES release 6 | Red Hat Enterprise Linux Server release 6.4 (Santiago) with CUDA 5.0 | CLE |
| Compilers | Intel CC 14.0.0 OpenMPI 1.6.5 | Intel CC 14.0.2 OpenMPI 1.6.4 | Cray C Compilers 8.2.1 cray-mpich/6.1.1 |
| Compiler flags | -O3 IEEE_FLAGS[1] | -O3 IEEE_FLAGS[1] -gencode arch=compute_35,code=sm_35 | -O3 -Kieee |

# PERFORMANCE

Table 1: Cloverleaf single threaded CPU performance - Intel Xeon E5-2680 @ 2.70GHz (Sandy bridge)

| Seq. Version | Run time (seconds) | % runtime difference |
|---|---|---|
| Original reference | 885.89 | – |
| OPS - developer | 1892.09 | -53.17 |
| OPS - optimized | 940.66 | -5.8 |

- The "developer" version (flat 1D loop, completely general) inhibits compiler optimizations, it's not very fast, but it's not *meant* to be fast
- The code generated sequential version does permit compiler optimizations such as auto-vectorization, performance matches the original
  - Evidence that a high-level approach, even on such a low level, does not imply performance loss

# PERFORMANCE – OPENMP, CUDA

Table 2: Cloverleaf multi-threaded CPU performance - Intel Xeon E5-2680 @ 2.70GHz (Sandy bridge) 32 cores

| OpenMP. Version | Run time (seconds) | % runtime difference |
|---|---|---|
| Original reference (32 OMP) | 135.53 | – |
| OPS (32 OMP) | 106.30 | 21.57 |

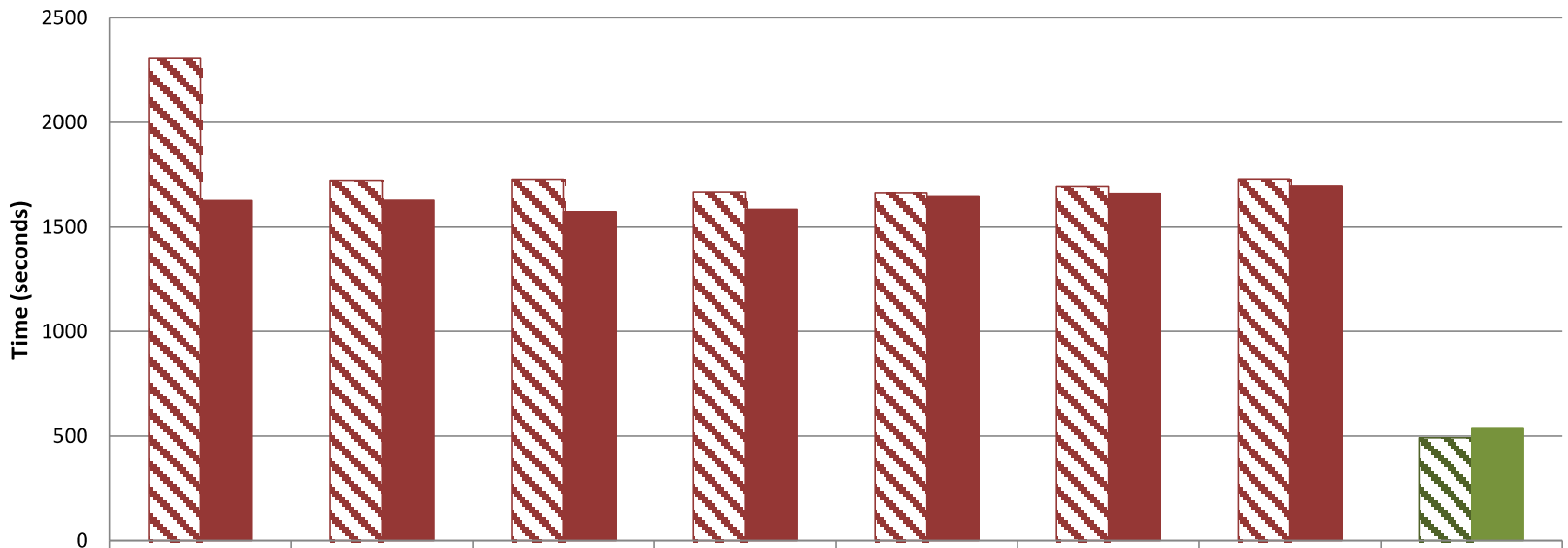
- Our OpenMP version is actually much faster than the original, due to NUMA effects – we allocate and initialise data in a way that avoids this issue
- In practice, a hybrid MPI+OpenMP approach is used to avoid NUMA altogether

Table 3: Cloverleaf GPU performance - NVIDIA K20c GPU

| CUDA Version | Run time (seconds) | % runtime difference |
|---|---|---|
| Original CUDA | 37.59 | – |
| OPS | 43.73 | -16.33 |

- The GPU implementation is a very basic one, relies entirely on caches (Texture, L1, L2) for data reuse
- The original has several optimizations – loops were fused, thus it's not an apples to apples comparison
  - We are working on doing these kinds of optimizations automatically

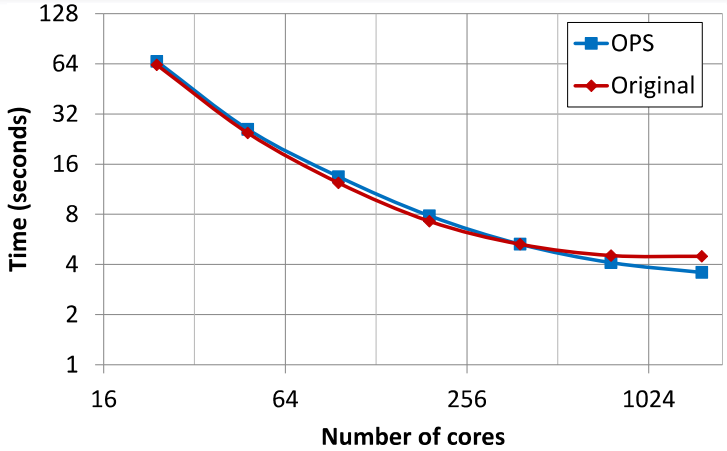
# Performance – Distributed Memory with MPI



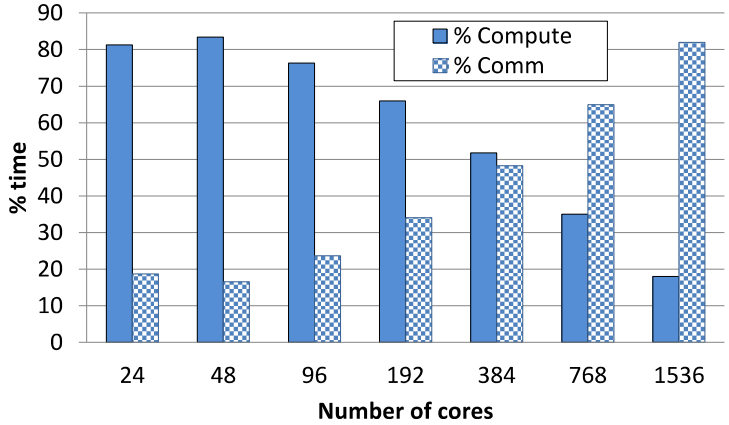

| | 32 OMP | 16 MPI | 32 MPI | 16 MPI x 2 OMP | 8 MPI x 4 OMP | 4 MPI x 8 OMP | 2 MPI x 16 OMP | CUDA |
|---|---|---|---|---|---|---|---|---|
| Original | 2306.37 | 1723.16 | 1728.05 | 1665.36 | 1661.24 | 1696.35 | 1729.82 | 491.17 |
| OPS | 1626.59 | 1627.70 | 1573.27 | 1585.67 | 1645.58 | 1657.72 | 1697.97 | 540.82 |

Figure 15: Cloverleaf performance of 3840×3840 mesh

# Performance – Distributed Memory with MPI

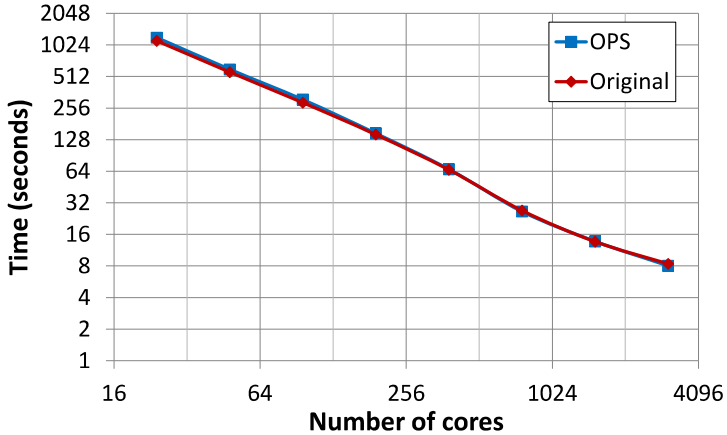



(a) Total runtime

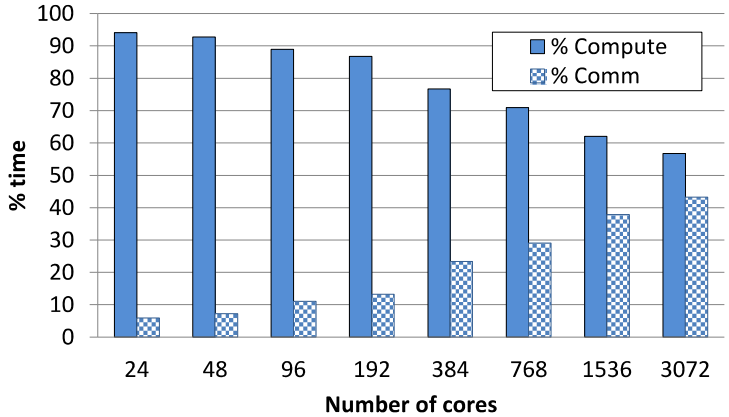



(b) OPS percentage runtime breakdown

Figure 16: Cloverleaf MPI performance on Archer - 960×960 mesh




(a) Total runtime




(b) OPS percentage runtime breakdown

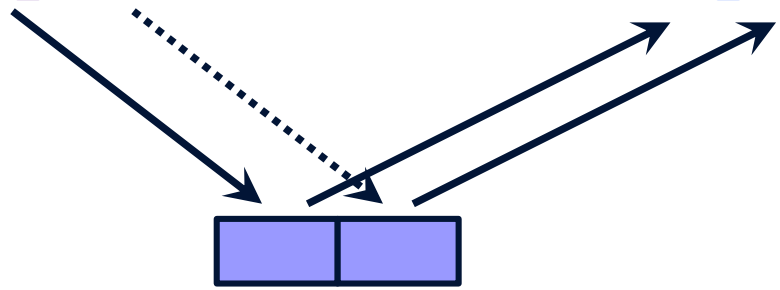
Figure 17: Cloverleaf MPI performance on Archer - 3840×3840 mesh

# Future Work

- So far we have almost everything working they have with the same performance – but with a single source code
- Further optimizations and features that the HLA enables us to do:
  - Loop fusion: loops that move some of the same data and there is no non-trivial data dependency between them can be fused, so common data is only moved once
  - Loop tiling: on top of loop fusion, where non-trivial data dependencies do exist: skewed execution

```
dat(mass_flux_x,00,OPS_WRITE)
dat(work_array7,00,OPS_WRITE)
```

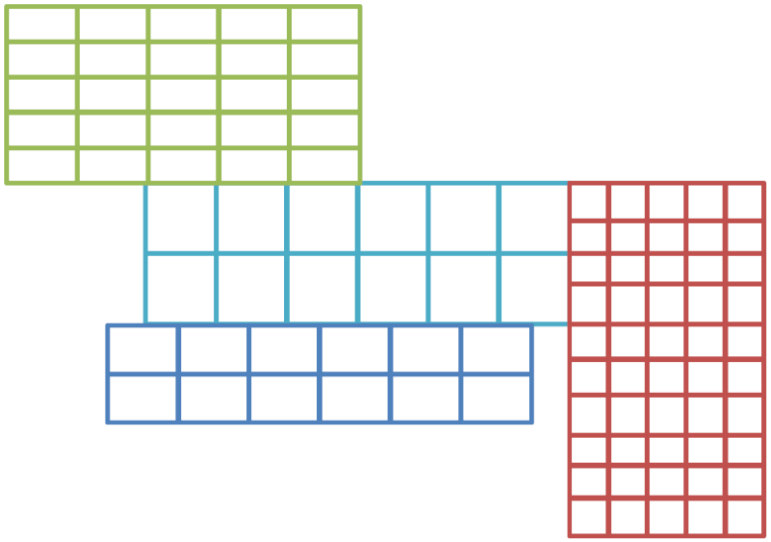
```
dat(mass_flux_x,00_P10,OPS_READ)
dat(work_array7,00_P10,OPS_READ)
```



  - Checkpointing: automatically save all data every once in a while so if something crashes, we can resume from that point – can be fully automated since we know how, when and what data is accessed
- The EPSRC project on multi-block structured mesh applications
  - ROTORSIM (Bristol)

# Extend to Multi-block Structured Mesh Applications

- Multi-Block Structured grids
  - How do you come up with an intuitive API?
  - Mismatching interfaces, sliding planes
  - Dependencies between blocks
  - Semantics of parallel execution of blocks
  - How do you partition? Unstructured mesh of structured meshes
  - Some of the original codes use horrible hacks and workarounds – outperforming them may not be an issue



# CONCLUSIONS

- Aim was to demonstrate to a wider HPC audience how HLAs might help in solving various scientific simulation challenges on future emerging systems.
- Presented a High-Level Abstraction for structured mesh applications
- Developer productivity
  - Development with a high-level framework is no more time consuming nor difficult than writing a one-off parallel program targeting only a single parallel implementation.
  - However the HLA strategy pays off with
    - A highly maintainable single application source
    - Does not compromise performance portability
    - lays the groundwork for providing support for execution on future parallel systems
- Performance matches the original – but at no cost to the developer
- Pros and Cons of the HLA approach
  - Need to select the correct abstraction and level of abstraction
  - Designing an HLA which is sufficiently general to handle a number of different applications, but be narrow enough be efficiently implemented by a single research group
  - Securing funding to maintain an HLA or a DSL so that the users can rely on it in the future

# DOWNLOAD

OPS and Cloverleaf available as Open source software

OPS - https://github.com/gihanmudalige/OPS

Original Cloverleaf - https://github.com/Warwick-PCAV/CloverLeaf