# Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2

Suneth D. Ekanayake
University of Warwick, UK
suneth.ekanayake@warwick.ac.uk

István Z. Reguly
Pázmány Péter Catholic University, Hungary
reguly.istvan@itk.ppke.hu

Fabio Luporini
Devito Codes, UK
fabio@devitocodes.com

Gihan R. Mudalige
University of Warwick, UK
g.mudalige@warwick.ac.uk

## ABSTRACT

In this paper, we investigate data movement-reducing and communication-avoiding optimizations and their practicable implementation for large-scale unstructured-mesh applications. Utilizing the high-level abstraction of the OP2 DSL for the unstructured-mesh class of codes, we reason about techniques for reduced communications across a consecutive sequence of loops – a *loop-chain*. The careful trade-off with increased redundant computation in place of data movement is analyzed for distributed-memory parallelization. A new communication-avoiding (CA) back-end for OP2 is designed, codifying these techniques such that they can be applied automatically to any OP2 application. The back-end is extended to operate on a cluster of GPUs, integrating GPU-to-GPU communication with CUDA, in combination with MPI. The new CA back-end is applied automatically to two non-trivial applications, including the OP2 version of Rolls-Royce's production CFD application, Hydra. Performance is investigated on both CPU and GPU clusters on representative problems of 8M and 24M node mesh sizes. Results demonstrate how for select configurations the new CA back-end provides between 30 – 65% runtime reductions for the loop-chains in these applications for the mesh sizes on both an HPE Cray EX system and an NVIDIA V100 GPU cluster. We model and examine the determinants and characteristics of a given unstructured-mesh loop-chain that can lead to performance benefits with CA techniques, providing insights into the general feasibility and profitability of using the optimizations for this class of applications.

## CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; *Model development and analysis*; • **Applied computing** → *Aerospace*.

## KEYWORDS

Communication-avoiding algorithms, OP2, Unstructured-mesh.



*ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA*

ACM ISBN 979-8-4007-0843-5/23/08.
https://doi.org/10.1145/3605573.3605604

**ACM Reference Format:**
Suneth D. Ekanayake, István Z. Reguly, Fabio Luporini, and Gihan R. Mudalige. 2023. Communication-Avoiding Optimizations for Large-Scale Unstructured-Mesh Applications with OP2. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3605573.3605604

## 1 INTRODUCTION

The end of frequency scaling in the middle of the last decade has led processor architectures to move towards increasingly massively parallel designs. As a result, modern processors have featured a proliferation of arithmetic capability in the form of increasing discrete processor cores, both on traditional CPUs as well as accelerator devices such as GPUs. For example, the number of processor cores on a high-end CPU currently has close to 100 cores, while GPUs are designed with even more parallelism; over 2000 "cores", albeit cores that are comparatively much simpler. Large-scale clusters of these devices have reached hundred-thousands to millions of cores as can be seen from the recently unveiled exascale supercomputers. However, the speed of memory and network channels interconnecting the processors and system/device memories have largely lagged behind, leading to significant memory bandwidth bottlenecks. As a result, the performance of many conventional algorithms, optimized for floating-point operations, has stalled. Developing algorithms with reduced data movement, or *communication-avoiding* (CA) algorithms have therefore become an intense area of research (e.g. [14, 33]). The underlying motivation of these works is to exploit the significantly high computing capability of these processors in place of communications, aiming to obtain higher performance.

The main challenge in adopting communication-avoiding optimizations or techniques in real-world applications is the significant effort and difficulty in implementing them and the subsequent maintenance of the code. The specific optimizations are highly-complex and involved, usually obfuscating the source code with platform-specific low-level features. A large body of work has developed the underlying theory for CA techniques including tiling [18, 27, 29] and reduced distributed-memory systems communications [13, 18]. Compile time application of these, such as optimizations based on the polyhedral model [16, 31] have been developed within compiler frameworks such as LLVM's Polly [12] and Pochoir [30]. Another strand of works such as by Demmel et al. [9, 10] have successfully developed libraries that applications can use to access CA-optimized numerical methods. More recently, domain-specific languages (DSLs) and similar high-level frameworks have demonstrated a pathway in applying these exotic optimizations to larger

non-trivial applications. Key DSLs with CA capabilities include Firedrake [15] for unstructured-mesh-based FE applications and OPS [27] and Devito [19] for structured-mesh-based applications.

The underlying goal of this paper is to examine and apply key data movement-reducing techniques to large-scale, real-world applications. To this end, we build upon previous work by Luporini et al [17] bringing together techniques for reducing data movement for the unstructured-mesh applications class, codifying them through the OP2 DSL [20], an embedded domain-specific language for developing unstructured-mesh applications. We begin with a focus on distributed-memory parallel execution and then its combination with on-node parallelization on many-core (GPU) processors. A new CA back-end for OP2 is designed such that any application using OP2 can utilize the optimizations. The new back-end is then applied to our main application of interest, Hydra, a production computational fluid dynamics (CFD) application used for aero-engine design at Rolls-Royce plc. More specifically, we make the following contributions:

- We design a new CA back-end for the OP2 DSL, focusing on its distributed-memory parallel operation and reason about techniques for reducing the number of MPI messages exchanged during the execution of a sequence of consecutive loops, a *loop-chain*. A key new feature of the back-end is its ability to run standard loops over unstructured-mesh sets interspersed with selected loop-chains with CA to obtain the best overall performance of the application (Section 3).
- The performance of loop-chains with CA is analytically modeled. We investigate the careful trade-off of increasing computations at shared MPI halos to satisfy loop dependencies in place of data movement via message passing. The analytic model provides insights to characterize the loop-chains and reason about whether a given loop-chain will benefit from the CA back-end of OP2 (Section 3.2).
- The distributed-memory back-end is extended to work on GPU clusters leveraging reduced MPI message passing when executing on a cluster of GPUs, enabling lower overheads in GPU-to-GPU communication with CUDA (Section 3.3).
- Finally, the CA back-end is applied to Hydra, using its recently re-engineered OP2 version [21], OP2-Hydra for use cases, with mesh sizes of 8M and 24M. Benchmarking is carried out on the ARCHER2 supercomputer, an HPE Cray EX system with AMD EPYC 7742 cores and the Cirrus GPU cluster with NVIDIA V100 GPUs at EPCC (Section 4).

Results indicate significant performance gains: up to 65% on select node counts on ARCHER2 (CPU) and Cirrus (GPU) clusters. However, they also point to the need of carefully selecting which loop-chains have CA optimizations turned on, as in some chains they lead to performance degradation over the non-CA version. To our knowledge the practical implementation of CA techniques for large-scale applications, particularly for large production codes such as Hydra from the unstructured-mesh domain, are limited in literature, not to mention benchmarking and performance analysis on both multi-core (CPU) and many-core (GPU) cluster systems. Reasoning about the viability and profitability of these optimizations for real-world codes is also novel, particularly through the development of an analytic model. Our work provides insights into the key determinants and characteristics of a given general unstructured-mesh loop-chain that can lead to performance benefits with CA optimizations. These insights, we believe are more broadly applicable, even to applications developed without OP2. On the other hand, the work also demonstrates how a high-level abstraction framework such as OP2 can seamlessly deliver these complex and exotic code transformations to real-world applications without affecting the science source, maintaining performance portability.

# 2 BACKGROUND

## 2.1 Communication-avoiding Algorithms

Communication-avoiding algorithms have a rich literature with reducing data movement identified as a fundamental optimization. Out of these techniques, improving data locality by restructuring loops or rescheduling loop iterations, generally known as tiling or loop-blocking, have long been well understood [32]. The theoretical underpinnings of these loop transformations have been comprehensively described through the polyhedral model [7].

Many frameworks and libraries have been developed based on the polyhedral model. Key works include PLuTo [8], a fully automatic polyhedral program optimization system, Polly [12], an LLVM framework for high-level loop and data locality optimizations, Pochoir [30], a compiler and runtime system for implementing stencil computations on multicore processors and PolyMage [22] and Halide [25] which specifically targets image processing pipelines. All these works create polyhedra – multi-dimensional sub-iteration spaces, within loops with static, regular access patterns/dependencies. Such loops can be readily viewed as iterations over a structured mesh with regular stencils defining the dependency neighborhood. The extension of these algorithms to distributed-memory systems requires them to account for the dependencies in the loop nests when the iteration space is spread over a number of processes, or disparate memory areas. This leads to the need for those dependencies to be satisfied through communications of extra halo layers. These extensions have been developed for several of the aforementioned frameworks, for PLuTo in [6] and Distributed Halide in [11]. However, applying these optimizations, particularly the ideas of the polyhedral model for unstructured-mesh applications are limited in literature as a consequence of the added complexity of managing the irregular dependencies of the unstructured-mesh. The dependence structure arises due to their characteristic indirect memory accesses, specifically indirect increments (e.g. `D[map[i]] += f(...)`) and indirect reads, through explicit connectivity *mappings* [17]. The dependence analysis needs to be done via mappings, as opposed to the static dependence neighborhoods (e.g. specified by a stencil) commonly exploited in general static loop optimizations. As such, the analysis is significantly more involved and needs to be carried out dynamically at runtime, as demonstrated by Luporini et al. [17] based on the *loop-chain* abstraction introduced in [14].

## 2.2 Loop-chain Abstraction and Sparse Tiling

A loop-chain is a sequence of consecutive loops without any global synchronisation points (such as global reductions) in between loops, specified or annotated with information to facilitate runtime dependency analysis. The information should be provided by the programmer or automatically derived from the code either through a code parser or a DSL's API. This information then enables us to

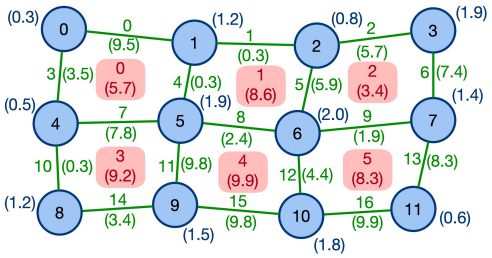



**Figure 1:** Example unstructured-mesh with nodes, edges and quadrilateral cells (data values in parenthesis)

```
1  for (int t = 0; t < tmax; t++) { //main iteration loop
2      ...
3      // loop over edges, updating nodes: update residuals
4      for (int iter = 0; iter < nedges; iter++) {
5           int mapidx1 = en[it*2+0]; int mapidx2 = en[it*2+1];
6           res[2*mapidx1+0] += pres[2*mapidx1+0]-pres[2*mapidx1+1];
7           res[2*mapidx1+1] += pres[2*mapidx2+0]-pres[2*mapidx2+1];
8
9           res[2*mapidx2+0] += pres[2*mapidx2+1]-pres[2*mapidx2+0];
10          res[2*mapidx2+1] += pres[2*mapidx1+1]-pres[2*mapidx1+0];
11     }
12
13     // loop over edges, updating nodes: calculate edge flux
14     for (int iter = 0; iter < nedges; iter++) {
15         int mapidx1 = en[it*2+0]; int mapidx2 = en[it*2+1];
16         int mapidx3 = ec[it*2+0]; int mapidx4 = ec[it*2+1];
17
18         flux[2*mapidx1+0] += res[2*mapidx1+0]*cw[4*mapidx3+0]
19                            - res[2*mapidx1+1]*cw[4*mapidx3+1];
20
21         flux[2*mapidx1+1] += res[2*mapidx2+1]*cw[4*mapidx3+2]
22                            - res[2*mapidx1+0]*cw[4*mapidx3+3];
23
24         flux[2*mapidx2+0] += res[2*mapidx2+1]*cw[4*mapidx4+2]
25                            - res[2*mapidx1+0]*cw[4*mapidx4+3];
26
27         flux[2*mapidx2+1] += res[2*mapidx1+0]*cw[4*mapidx4+0]
28                            - res[2*mapidx1+1]*cw[4*mapidx4+1];
29     }
30     ...
31 }
```

**Figure 2:** Example *2-loop-chain*

reason about the dependencies of the sequence of the loops collectively. For loop-chains over unstructured-meshes, as detailed in [17], a *sparse tiling schedule* can be created from the analysis, providing an execution ordering of the iterations over the mesh. This ordering which is semantically equivalent to the original can then be used to carry out the loop iterations. For example, consider the two *sequential* loops detailed in Figure 2, over the unstructured-mesh shown in Figure 1. The mesh consists of nodes, edges and cells, where a loop over edges, updating the nodes, at each end of the edge would require explicit connectivity information specified by a mapping of *edges-to-nodes*, `en`. Two such loops, occurring within a larger time-marching iterative loop, are illustrated in lines 4-11 (`update`) and 14-29 (`edge_flux`) in Figure 2. Both the loops increment data held on the nodes, `res` and `flux` respectively, indirectly via the mapping array `en`. The `edge_flux` loop indirectly reads data, cell wights (`cw`), held on the two cells next to an edge, via the mapping *edges-to-cells*, `ec`. The `update` and `edge_flux` loops taken consecutively can be viewed as a loop-chain with two *parallel* loops and specified using the OP2 DSL's API [20] as in Figure 3.

Using the definition in [17] the loop-chain can be defined as follows:

- Loop-chain $\mathbb{L} = L_0, L_1, ..., L_{n-1}$, an ordered sequence of $n$ loops : `update`, `edge_flux` where $n = 2$, declared as `op_par_loops`.

```
1  inline void update (double* res1, double* res2, double* pres1,
2    double* pres2) {
3      res1[0] += pres1[0]-pres1[1];   res1[1] += pres2[0]-pres2[1];
4      res2[0] += pres2[1]-pres2[0];   res2[1] += pres1[1]-pres1[0];
5  }
6
7  inline void edge_flux (double* flux1, double* flux2, double* res1,
8    double* res2, double* cw1, double* cw2) {
9      flux1[0] += res1[0]*cw1[0] - res1[1]*cw1[1];
10     flux1[1] += res2[1]*cw1[2] - res2[0]*cw1[3];
11
12     flux2[0] += res2[1]*cw2[2] - res1[1]*cw2[3];
13     flux2[1] += res1[0]*cw2[0] - res1[1]*cw2[1];
14 }
15
16 op_set nodes = op_decl_set(nnode, "nodes");
17 op_set edges = op_decl_set(nedge, "edges");
18 op_set cells = op_decl_set(ncell, "cells");
19
20 op_map e2n = op_decl_map(edges, nodes, 2, en, "e2n");
21 op_map e2c = op_decl_map(edges, cells, 2, ec, "e2c");
22
23 op_dat dres  = op_decl_dat(nodes, 2, "double", res,  "res" );
24 op_dat dpres = op_decl_dat(nodes, 2, "double", pres, "pres");
25 op_dat dcw   = op_decl_dat(cells, 4, "double", cw,   "cw"  );
26 op_dat dflux = op_decl_dat(nodes, 2, "double", flux, "flux");
27
28 for (int t = 0; t < tmax; t++) { //main iteration loop
29     ...
30     // loop over edges, updating nodes: update residuals
31     op_par_loop(update,   "update",   edges,
32       op_arg_dat(dres,  0, e2n, 2, "double", OP_INC  ),
33       op_arg_dat(dres,  1, e2n, 2, "double", OP_INC  ),
34       op_arg_dat(dpres, 0, e2n, 2, "double", OP_READ ),
35       op_arg_dat(dpres, 1, e2n, 2, "double", OP_READ ));
36
37     // loop over edges, updating nodes: calculate edge flux
38     op_par_loop(edge_flux,"edge_flux", edges,
39       op_arg_dat(dres,  0, e2n, 2, "double", OP_READ),
40       op_arg_dat(dres,  1, e2n, 2, "double", OP_READ),
41       op_arg_dat(dcw,   0, e2c, 4, "double", OP_READ),
42       op_arg_dat(dcw,   1, e2c, 4, "double", OP_READ),
43       op_arg_dat(dflux, 0, e2n, 2, "double", OP_INC ),
44       op_arg_dat(dflux, 1, e2n, 2, "double", OP_INC ));
45     ...
46 }
```

**Figure 3:** *2-loop-chain* in OP2 API

- Iteration spaces $\mathbb{S} = S_0, S_1, ..., S_{m-1}$, a collection of disjoint iteration spaces representing mesh element types : `edges`, `nodes` and `cells`, $m = 3$, declared as `op_sets`.
- Explicit connectivity between iteration spaces $\mathbb{M} = M_0, M_1, ..., M_{o-1}$, where $M : S_i \rightarrow S_j^a$ is a map with arity $a$ : `en` and `ec`, $o = 2$, declared as `op_maps`. For example, `en` has an arity of 2, mapping an edge to two nodes.
- Access descriptors, one or more 2-tuples of the form of <$M$, `mode`> associated with a loop $L_i$ where $M$ is a map indicating indirect access or `ID` (identity mapping) indicating direct access on data (specified by `op_dats` in OP2) defined on the iteration space of the loop. `mode` is the mode of data access, read (`OP_RW`), write (`OP_WRITE`) or increment (`OP_INC`). In OP2, the access descriptors are defined using the `op_arg_dat` API.

The above definition provides information to carry out an inspection or analysis phase to create a set of tiles, i.e. the aforementioned sparse tiling schedule. The schedule will guarantee those data dependencies are not violated such that each tile can be executed in its entirety without any data access to/from outside the tile. Executing a tile $T_i$ entails executing all the iterations from $L_0$ belonging to that tile, followed by all the iterations in that tile for $L_1$ and so on up to $L_{n-1}$. Then the next tile $T_{i+1}$ is executed in a similar manner, continuing this pattern of execution until all the tiles have been completed. It is important to note that sparse tiling assumes

that the order of execution of loop iterations does not affect the final result, at least within machine precision. This encompasses a large number of explicit numerical schemes, particularly when solving PDEs in numerical simulation applications. For the full description of the loop-chaining abstraction and the development of sparse tiling inspection/execute phases, we refer the reader to Luporini et al. [17].

The dependency analysis allows to exploit communication-avoidance at two levels of parallelism. On a distributed-memory parallel level, as we will demonstrate, the idea would be to move all communications to the beginning of the loop-chain, eliminating per-loop halo exchanges between neighboring processes, in place of a larger aggregated message. In this case, each mesh partition held by an MPI process can be thought of as a single "tile", which will be executed without halo exchanges within the loop-chain. On a shared-memory multi-threaded parallelization level, the idea would be to select sufficiently sized tiles allowing to keep a working set of data in the fast cache memory of processors reducing the number of accesses from/to slower main memory per loop. The implementation of these ideas within OP2, creating a new communication-avoiding back-end and applying it to large-scale unstructured-mesh applications, investigating performance, form the main contribution of this paper. We will (1) initially codify sparse tiling for distributed-memory execution and then (2) extend it for GPU clusters integrating the back-end to work with CUDA code generated by OP2.

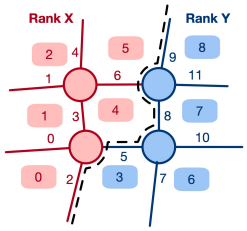
## 2.3 Related Work

Sparse tiling from [17] has been previously used in Firedrake [15], a high-level DSL framework for the automated solution of finite element computations. It requires problems to be specified in the Unified Form Language (UFL) [5]. The specification is then used to generate parallel executables on multi-core CPU nodes and clusters. In [17], both shared-memory and distributed-memory sparse tiling are implemented in Firedrake and the performance of a seismic exploration benchmark application, Seigen is evaluated on a CPU cluster. In contrast, in this paper, we demonstrate the performance of sparse tiling in a significantly larger (≈100K LoC F90, ≈500 loops), production application, Hydra on both CPU and GPU cluster systems. A new analytic performance model is also developed to gain insights into the performance behaviour with CA.

# 3 A COMMUNICATION-AVOIDING BACK-END FOR OP2

OP2 uses an *owner compute* model for parallelizing computations on a distributed memory parallel system [20]. In this model, the unstructured-mesh, defined by mappings and data is partitioned among several processes so that each process *owns* some of the mesh elements. A process will only perform computations to update elements in their own partitions but will require data from elements in other partitions held in separate processes, specifically at the boundaries of the partitions. Thus, copies of data held in foreign partitions need to be communicated following the standard "halo" exchange mechanisms when using a message passing parallel implementation.

Figure 4 illustrates this halo setup and configuration in OP2, where the back-end separates the iteration space such that it's




| | Set | *core* | *export halo* | | *import halo* | |
|---|---|---|---|---|---|---|
| | | | exec | non-exec | exec | non-exec |
| X | edges | 0,1,3,4,6 | 2 | - | 5,8,9 | - |
| X | cells | 0,1,2,4,5 | - | 0,4,5 | - | 3,7,8 |
| Y | edges | 7,10,11 | 5,8,9 | - | 2 | - |
| Y | cells | 3,6,7,8 | - | 3,7,8 | - | 0,4,5 |

**Figure 4:** OP2 partitioning over two MPI ranks and resulting halos on each rank [20]

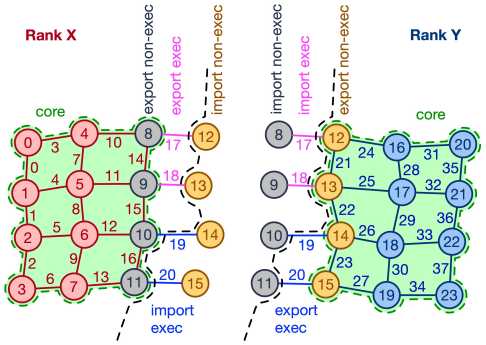



**Figure 5:** Halo layer with depth 1

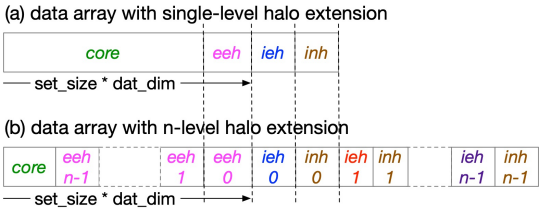



**Figure 6:** `op_dat` and `op_map` data structures with (a) single and (b) multiple halo levels

segmented into: (1) *core* set of iterations that do not need to access halo data, (2) an *export halo* consisting of mesh data to be sent from the local process to some foreign process and (3) an *import halo* consisting of mesh data received from some foreign process. The elements in the import and export halos are further separated into two groups depending on whether redundant computations will be performed on them. For example, in the mesh in Figure 4, a loop over edges updating cells will require edges no: 5,8 and 9 executed on rank X to update cells no: 4 and 5 (i.e. an import execute halo, *ieh*). However, a loop over edges reading data on cells will require cells 0,4 and 5 imported onto rank Y (i.e. an import non-execute halo, *inh*). The *inh* is essentially a read-only halo. These then have a corresponding export execute (*eeh*) and export non-execute (*enh*) halos on each of the local processes.

## 3.1 Multi-layered Halo Data Structure

In OP2, the mesh data in `op_maps` and `op_dats` are held in 1D arrays with the `core`, `export` and `import` halos structured as illustrated

**Algorithm 1:** OP2 loop execution

**Input:** op_par_loop $l$ over set $S_l$
**Result:** Execute loop

```
1  MPI_Isend(eeh, enh);
2  MPI_Irecv(ieh, inh);
3  foreach iteration I ∈ S_l^c do
4  |   execute_iteration(I);
5  end foreach
6  MPI_Wait(eeh, enh, ieh, inh);
7  foreach iteration I ∈ S_l^e, S_l^i do
8  |   execute_iteration(I);
9  end foreach
```

**Algorithm 2:** Loop-chain execution with CA

**Input:** Loop-chain, $\mathbb{L} = \{L_0, ..., L_{n-1}\}$, op_dats, $\mathbb{D}$ used in $\mathbb{L}$
**Result:** Execute loop-chain with *CA*

```
// Find op_dats requiring halo syncs: D^h ⊆ D
1  D^h ← halo_exch_dats(D, <M, mode>, L);
// Compute #halo layers required for each loop in L
2  HL_l ← calc_halo_layers(L, D^h);
// Find core, exec & non-exec halo #iters for loops in L
3  S_l^c, S_l^h, S_l^n ← calc_iters(S_l, HL_l);
// Rearrange and renumber multi-layered core, eeh & enh
// for each op_dat d ∈ D^h
4  S^eeh, S^enh ← restructure_elements(D^h, HL_l, S_l^c, S_l^h, S_l^n);
5  m^{eeh+enh} ← create_grouped_msg(D^h, S^eeh, S^enh);
6  MPI_Isend(m^{eeh+enh});
7  MPI_Irecv(m^{ieh+inh});
8  foreach loop l ∈ L do
9  |   foreach iteration I ∈ S_l^c do
10 |   |   execute_iteration(I);
11 |   end foreach
12 end foreach
13 MPI_Wait(m^{eeh+enh}, m^{ieh+inh});
14 foreach loop l ∈ L do
15 |   foreach iteration I ∈ S_l^h do
16 |   |   execute_iteration(I);
17 |   end foreach
18 end foreach
```

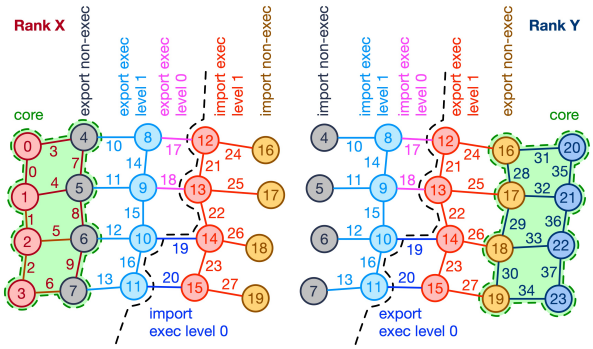



**Figure 7:** Halo layer with depth 2

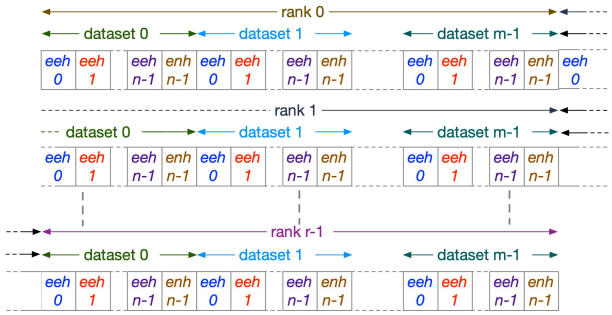



**Figure 8:** Grouped halo array

in Figure 6(a). This ordering then enables latency hiding where the loop iterations in an op_par_loop, corresponding to the *core* can be carried out while halo exchanges are in-flight as these iterations do not access halo data. This latency-hiding algorithm is detailed in Alg 1. OP2 exchanges the *ieh* and *inh* in separate messages. After all the messages have been sent/received, execution over execute halos can be performed. In an op_par_loop, halos for an op_dat are exchanged only if (1) it is indirectly accessed as a read (OP_READ) or a read/write (OP_RW) and (2) it has been modified by a preceding loop, i.e. the halos need updating. A *dirty-bit* is used to keep track of when an op_dat is updated (by an OP_RW, OP_WRITE or an increment, OP_INC) in a loop.

The OP2 halos described above essentially maintain the data dependencies required to execute each process's partition independently in parallel. Explicit messages are exchanged to update/sync the halos when carrying out computations in each op_par_loop. The dependency neighborhood for a single loop, therefore, can

**Algorithm 3:** calc_halo_layers

**Input:** Loop-chain, $\mathbb{L} = \{L_{n-1}, \ldots, L_0\}$, op_dats requiring halo syncs, $\mathbb{D}^h$, their access descriptors, <$M$, mode>, loops where op_dats ($\in \mathbb{D}^h$) are accessed, $\mathbb{A}_{\mathbb{D}^h}$
**Output:** Halo extensions, $\mathbb{HE}$ for loops in loop-chain, $\mathbb{L}$

```
// Calculate halo extension for individual op_dats in loops
1  foreach op_dat D ∈ D^h do
2  |   halo_ext ← 0;   ind_rd ← false; // True for indirect read
3  |   foreach loop l ∈ L do // Iterate from loop n-1 to 0
4  |   |   HE_{D_l} ← 1;
5  |   |   if D_l<M, mode> ≠ NULL && l ∈ A_{D_l} then
6  |   |   |   if ind_rd && (mode = OP_WR || mode = OP_INC || mode = OP_RW) then
7  |   |   |   |   HE_{D_l} ← halo_ext + 1;        halo_ext ← 0;
8  |   |   |   |   ind_rd ← false;                 continue;
9  |   |   |   end if
10 |   |   |   if M ≠ ID && (mode = OP_RD || mode = OP_RW) then
11 |   |   |   |   halo_ext ← halo_ext + 1;        HE_{D_l} ← halo_ext;
12 |   |   |   |   ind_rd ← true;                  continue;
13 |   |   |   end if
14 |   |   |   if M = ID && (mode = OP_RD || mode = OP_RW) then
15 |   |   |   |   HE_{D_l} ← 1;                   halo_ext ← 0;
16 |   |   |   |   ind_rd ← false;                 continue;
17 |   |   |   end if
18 |   |   end if
19 |   end foreach
20 end foreach
// Calculate effective halo extension for loops
21 foreach loop l ∈ L do
22 |   HE_l = max(HE_{D_l^h});
23 end foreach
```

be viewed as a halo layer with a depth of 1 (see Figure 5). As detailed in [17], in a loop chain with $n$ loops, syncs per loop can be eliminated if a large dependency neighborhood, maximally a layer with depth of $n$ can be communicated at the start of the loop-chain and computed over, redundantly to update the mesh elements to satisfy the dependencies, that would have otherwise been updated as a halo exchange. Thus for the loop-chain with 2 loops detailed in Figure 3, a halo depth of 2 needs to be maintained as shown in Figure 7.

The maximum depth of $n$ is required only when in a loop-chain, $L_0, L_1, \ldots, L_{n-1}$, each loop $L_i$ updates an op_dat $d$ and the next loop $L_{i+1}$ read or read/writes to $d$. This leads to iteration spaces that decrease in size for each loop in the loop-chain. Specifically, to compute $I$ iterations of the last loop in the loop-chain, $L_{n-1}$, the

loops, $L_{n-1}, L_{n-2}, \ldots, L_0$ should be iterating over $I$ plus halo depths of $1, 2, \ldots, n$ respectively. This algorithm is detailed in Alg 2. In the main inspection/setup phase, `halo_exch_dats` identifies the `op_dats` for halo exchange based on their access mode and dirty-bit value as discussed above. Then, `calc_halo_layers` compute the number of halo layers required for each loop in the loop-chain $\mathbb{L}$. Its analysis is detailed in Alg 3. Next, `calc_iters` (in Alg 2) computes the iteration counts within the core and halo layers, while `restructure_elements` separates each halo layer into core and *eeh* as shown in Figure 6(b) and renumbers mappings of core, *eeh* and *enh* accordingly. OP2's halo data structure was extended to support this multi-layered halo setup. The loop-chain executes *core* followed by $eeh_{n-1}$ to $eeh_0$, then all the import halos $ieh_0$ to $ieh_{n-1}$ (see lines 8-18 in Alg 2). Given a loop-chain, Alg 3 calculates the minimum halo extension required for each `op_dat` in each loop according to its individual data access patterns. Then, the maximum halo extension required for a loop is obtained based on the halo extensions calculated for each individual `op_dat` in the loop, finally making the loop's halo extension effective for all the `op_dats` in the loop.

To further reduce the number of messages exchanged for the whole loop-chain, the halos from multiple loops and `op_dats` are organized into a structure where a single message to be sent/received per MPI rank can be constructed (line 5 in Alg 2) as illustrated in Figure 8. Alg 2 also does latency hiding where the `core` iterations of each loop in the chain are first executed while halos are in flight. After completing send/receives, iterations in the multiple halo layers are executed.

## 3.2 Analytic Model for Loop-chain Performance

An analytic performance model for reasoning about the runtimes of OP2 loops and an equivalent loop-chain executed with the communication-avoiding setup can be developed based on a parameterization of loop characteristics, communication patterns and machine properties. Considering the execution of a single OP2 loop, $l$ as detailed in Alg 1, its runtime can be modeled by (i) the time taken for computing $S_l^c$ (core) and $S_l^e + S_l^i$ (execute halo) number of iterations, (ii) time taken to synch/communicate halos, minus any overlap of computation and communication. If we note $S_l^e + S_l^i = S_l^h = S_l^1$ to indicate that this is execution over a single halo layer, then the time taken by an OP2 loop is given by:

$$T_{op2,l} = MAX[\ g_l S_l^c,\ 2d_l p_l (L + m_l^1/B)\ ] + g_l S_l^1 \tag{1}$$

Here, $g_l$ is the *compute* time for one iteration of the loop body, $d_l$ is the number of `op_dats` with halos to be synced, $p_l$ is the maximum number of neighboring processes to communicate halos with per MPI process. $m_l^1$ is the maximum message size (in bytes) sent to a neighbor for either *eeh* or *enh* halos. The superscript 1 indicates the number of halo layers exchanged (1 is the default for OP2 loops). The multiplier 2 accounts for the time for sending a separate message for *eeh* and *enh*. The maximum message size and number of neighbors per MPI process are only known at runtime after the mesh partitioning [20]. The maximum is used for each of the components above to model the critical path of the runtime, where for example we assume that the halo exchange cost between processes only completes as the slowest exchange between a pair of processes. $L$ and $B$ are the latency and bandwidth of the network respectively. Then, the time taken for $n$ number of OP2 loops in a loop-chain $\mathbb{L}$ is simply the sum of the time taken by the individual loops:

$$T_{op2,\mathbb{L}} = \sum_{l=0}^{n-1} T_{op2,l} \tag{2}$$

When the loop-chain is executed with the communication-avoidance setup using the multi-layered halo data structures, a single grouped halo message is exchanged per neighbor process. This and the execution steps in Alg 2 lead to a total runtime of the full loop-chain:

$$T_{ca,\mathbb{L}} = MAX\left[\sum_{l=0}^{n-1} g_l S_l^c,\ p(L + m^r/B + c)\right] + \sum_{l=0}^{n-1} g_l S_l^h \tag{3}$$

As detailed in Alg 2, $S_l^h$ includes iterations from the execute halos of multiple levels for each loop. The message size, $m^r$ is the maximum grouped message size sent to each neighbor, which combines both the *eeh* and *enh* into a single message, as discussed before. $r$ (where $r \leq n$) indicates the maximum number of halo layers required to carry out the CA algorithm for the loop-chain. Thus, $m^r$ is given by :

$$m^r = \sum_{l=0}^{n-1} \left( \sum_{d=0}^{d_l - 1} \left( S_d^{eeh,h_l} + S_d^{enh,h_l} \right) \times \delta \right) \tag{4}$$

Here, $d_l$ is the number of halo synching dats in loop $l$, $h_l$ is the halo extension for loop $l$, $S_d^{eeh,h_l}$ - *eeh* size up to level $h_l$ of the set on which the data set $d$ is defined, $S_d^{enh,h_l}$ - *enh* size of level $h_l$. Not all *enh* levels up to $h_l$ are packed into the message, only the levels updated are included. Given that a larger number of messages are grouped for communication, an additional compute cost (a packing and unpacking cost) $c$ is added to communication per neighbor. Finally, $p$ is the maximum number of neighbors communicated by a process, when exchanging the grouped message and $\delta$ is the size of a data element of the `op_dat` $d$ in bytes.

From equations (2) and (3), we can see that the CA version saves time to communicate with the single message sent per neighbor, but this gain only becomes applicable when the time to compute the core iterations, $g_l S_l^c$ is smaller than the communication time due to the latency hiding setup of the execution. Thus, we can hypothesize that any performance gains would more likely appear at high processor counts (i.e. at large machine size) in a strong scaling scenario, when the core iterations per partition (one partition is assigned per process in OP2) is smaller. The communication time further depends on the maximum number of halo layers $r$ determining the message size $m^r$. If $r$ is significantly smaller than $n$, then the gains are likely to be large. However, any performance gains from faster communications can be diminished if the sum of the times to execute over the multiple halos given by $\sum_{l=0}^{n-1} g_l S_l^h$ in (3) is significantly larger than the sum of times to execute over a single halo region given by $g_l S_l^1$ in (2). Hence, again the maximum number of halo layers involved in the CA execution of the loop-chain makes a significant impact.

## 3.3 Cluster of GPUs

Extending the CA distributed-memory execution to a cluster of GPUs can also be carried out, given that OP2 code-generates CUDA with MPI. In the GPU CA version, the halos are transferred via

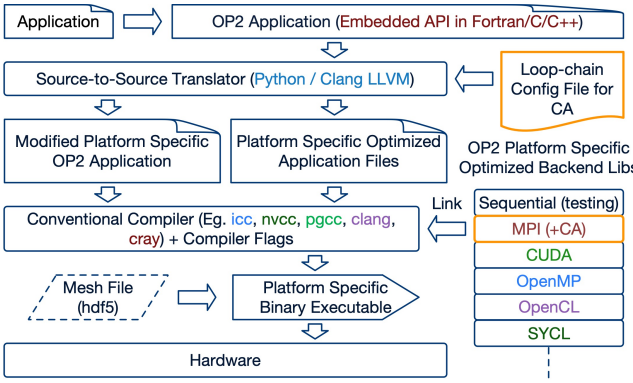



**Figure 9:** OP2 Code Generation with CA

MPI, by first copying it to the host over the PCIe bus. This implementation does not utilize NVIDIA's GPUDirect [23] technology for transferring data between the GPUs. Instead, a communication pipeline is setup, allowing for maximum overlap of kernels, memcopy, communication and core computations. We found that this performs better than GPUDirect, which in many cases did not run simultaneously with the computing kernels. Again the multi-layered halo setup will remain the same, but an extra data copy from host to device and vice versa will occur during halo exchange as detailed before. This cost can be approximated by a larger communication latency $\Lambda$ replacing $L$ in equation (3). Additionally, $g_l$ will need to be estimated for a GPU.

## 3.4 Automatic Code-generation

The integration of CA optimizations to OP2 is done such that any application developed with the OP2 API can immediately utilize the new capabilities without modifications to the high-level source. The only addition to OP2's existing automatic code-generation process [20] (as detailed in Figure 9) is the use of a configuration file specifying the list of loops to be chained in the application. The file details loop names, loop count and maximum halo extension of loops. The OP2 code-generator was extended so that it can automatically generate the loop-chain execution template in Alg. 2 for these selected loops. The generated code is human-readable, similar to all code generated by OP2. Finally, the code can be compiled using a conventional compiler linking the new CA back-end library to generate the executable to run on the selected hardware.

# 4 PERFORMANCE

We now investigate the performance of the communication-avoiding back-end, applying it to two existing applications developed with OP2 : (1) a representative CFD mini-app, MG-CFD [24] used for benchmarking and co-design and (2) the recently developed OP2 version of Hydra, OP2-Hydra [21], a large-scale production CFD application used at Rolls-Royce plc.

Performance is benchmarked on two systems, namely the HPE Cray EX system, ARCHER2 and the SGI/HPE 8600 GPU cluster, Cirrus both located at EPCC UK. Table 1 briefly details the key hardware and system setup of these two systems. Each ARCHER2 node consists of two AMD EPYC 7742 processors each with 64 cores (128 total cores) arranged in an 8×NUMA regions per node (16 cores per NUMA region) configuration [4]. Each node is equipped with 256 GB of memory. The nodes are interconnected by an HPE Cray Slingshot, 2×100 Gb/s bi-directional per node network. The GNU compiler collection version 10.2.0 was used on ARCHER2 with compiler flags noted in the table. The Cirrus GPU cluster consists of 4×V100 GPUs per node configuration, each node also consisting of 2×Intel Xeon Gold 6248 (Cascade Lake) processors, each with 20 cores (40 total cores). A node has 384GB main memory and a single V100 GPU has 16GB global memory [3].

**Table 1:** Systems Specifications

| System | ARCHER2<br>HPE Cray EX [4] | Cirrus<br>SGI/HPE 8600 GPU Cluster [3] |
|---|---|---|
| Processor | AMD EPYC 7742<br>@ 2.25 GHz | Intel Xeon Gold 6248<br>(Cascade Lake) @ 2.5 GHz<br>+ NVIDIA Tesla<br>V100-SXM2-16GB GPU |
| (procs×cores)<br>/node | 2×64 | 2×20 + 4×GPUs |
| Mem/node | 256 GB | 384 GB + 40GB/GPU |
| Interconnect | HPE Cray Slingshot<br>2×100 Gb/s<br>bi-directional/node | Infiniband<br>FDR, 54.5 Gb/s |
| OS | HPE Cray LE (SLES 15) | Linux CentOS 7 |
| Compilers | GNU 10.2.0 | nvfortran (nvhpc 21.2) |
| Flags | `-O2 -eF -fPIC` | CUDA 11.6 and sm_70 -O2 -Kieee |
| MPI | Cray MPICH 8.1.23 | MPT 2.25 |

## 4.1 MG-CFD

MG-CFD [24] is a 3D unstructured multi-grid, finite-volume computational fluid dynamics (CFD) mini-app for inviscid-flow, developed by extending the CFD solver in the Rodinia benchmark suite. The mini-app implements a 3D finite volume solution of the Euler equations for inviscid, compressible flow over an unstructured grid. The application uses multi-grid for accelerating the convergence of the solution. MG-CFD has been converted to use the OP2 API and its performance has been previously benchmarked in [28]. For our experiments, we use the NASA Rotor 37 meshes, which represent the geometry of a transonic axial compressor rotor, widely used for validation in CFD.

*4.1.1 Synthetic Loop-chains:* As discussed in Section 3, an OP2 loop will exchange MPI halos for an `op_dat` if it is to be indirectly read (`OP_READ` or `OP_RW`) in a loop, but has been modified (`OP_WRITE`, `OP_INC` or `OP_RW`) in a preceding loop. In this case, we note the `op_dat`'s halos as *dirty* at the start of the loop, triggering an MPI halo exchange before accessing halo values for computation. As such, two consecutive loops, the first modifying an `op_dat` followed by a second reading the same data indirectly will be our target access pattern (i.e. access descriptor in the loop-chain abstraction), for applying sparse tiling. However, consecutive loops with the above access descriptor do not exist in MG-CFD. Nevertheless, the relatively small size and simplicity of MG-CFD meant that we could add a *synthetic* sequence of loops to create the required setup. We are then able to create arbitrarily extendable loop-chains with the above target access pattern allowing to examine the limits of a single loop-chain and observe its consequent performance, reasoning with the performance model. The new loops introduced to MG-CFD consists of two loops [1], similar in structure to Figure 3. The first, `update` kernel modifies the `op_dat`, `dres` through an indirect

**Table 2:** MG-CFD on ARCHER2 - Model Components: OP2 comms ($\sum(2dpm^1)$) - CA comms ($pm^r$) in bytes, OP2 core iterations ($\sum(S^c)$) - CA core iterations ($\sum(S^c)$), OP2 halo iterations ($\sum(S^1)$) - CA halo iterations ($\sum(S^h)$) and performance gain% of CA over OP2

| #Nodes | #Loops | 8M Mesh | | | | | | | 24M Mesh | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | OP2 | | | CA | | | Gain% | OP2 | | | CA | | | Gain% |
| | | $\sum(2dpm^1)$ | $\sum(S^c)$ | $\sum(S^1)$ | $pm^r$ | $\sum(S^c)$ | $\sum(S^h)$ | | $\sum(2dpm^1)$ | $\sum(S^c)$ | $\sum(S^1)$ | $pm^r$ | $\sum(S^c)$ | $\sum(S^h)$ | |
| 4 | 2 | 877600 | 96494 | 14408 | 1763200 | 90460 | 42292 | -17.56 | 2776200 | 340060 | 32220 | 5497800 | 325105 | 99731 | -5.47 |
| | 8 | 3510400 | 385976 | 57632 | 1763200 | 258408 | 169168 | -2.65 | 11104800 | 1360240 | 128880 | 5497800 | 1029981 | 398924 | 6.30 |
| | 32 | 14041600 | 1543904 | 230528 | 1763200 | 351631 | 676672 | 15.62 | 44419200 | 5440960 | 515520 | 5497800 | 1681473 | 1595696 | 2.66 |
| 16 | 2 | 365400 | 23148 | 5850 | 747600 | 20962 | 17485 | -2.50 | 1006720 | 82868 | 13344 | 2039840 | 77673 | 38976 | -20.50 |
| | 8 | 1461600 | 92592 | 23400 | 747600 | 47264 | 69940 | 10.85 | 4026880 | 331472 | 53376 | 2039840 | 217854 | 155904 | -4.20 |
| | 32 | 5846400 | 370368 | 93600 | 747600 | 49646 | 279760 | 26.00 | 16107520 | 1325888 | 213504 | 2039840 | 283708 | 623616 | 16.47 |
| 64 | 2 | 168360 | 5658 | 2260 | 356160 | 4783 | 7277 | -9.70 | 406080 | 20154 | 5348 | 825600 | 18125 | 16183 | -8.76 |
| | 8 | 673440 | 22632 | 9040 | 356160 | 8285 | 29108 | 15.98 | 1624320 | 80616 | 21392 | 825600 | 41653 | 64732 | 7.83 |
| | 32 | 2693760 | 90528 | 36160 | 356160 | 8295 | 116432 | 34.45 | 6497280 | 322464 | 85568 | 825600 | 44690 | 258928 | 31.65 |

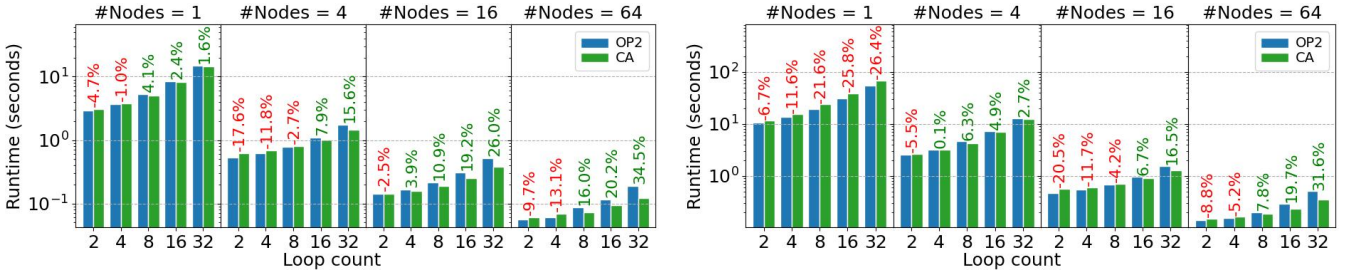



**Figure 10:** MG-CFD CA performance with 8M (left) and 24M (right) mesh on ARCHER 2

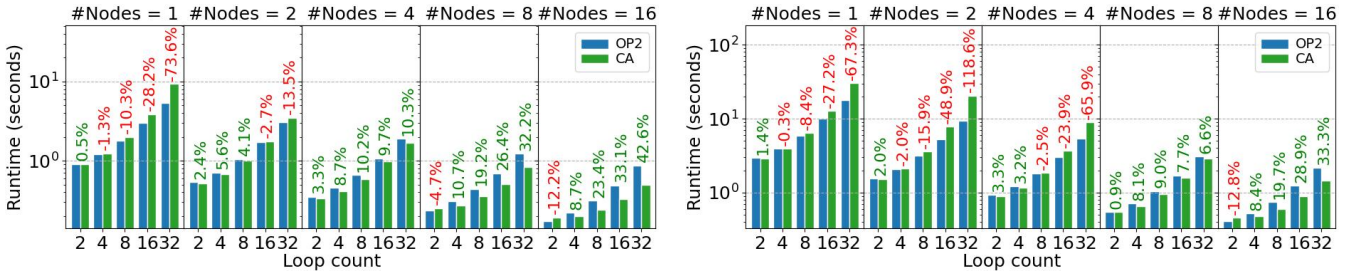



**Figure 11:** MG-CFD CA performance with 8M (left) and 24M (right) mesh on Cirrus

increment, `OP_INC` operation while iterating over the `edges`. Thus, `dres` becomes dirty at the end of this loop. The second, `edge_flux` kernel, indirectly reads `dres`. `edge_flux` kernel is, in fact, a replica of the most time-consuming loop in MG-CFD, `compute_flux_edge` which has the same access modes (full kernels not shown in Figure 3.). This enables us to make an effective comparison between reducing communications, i.e. no halo exchanges in the second loop and the consequent increase in the (redundant) computations over the larger depth halos in the first loop.

This *2-loop* chain is enclosed within an outer loop whereby setting its iteration count `nchains`, we can create longer loop-chains to explore performance [1]. With `nchains` = 1, the execution of the loop chain, according to Alg 2 will not result in a reduction in the number of MPI messages exchanged. However, for `nchains` > 1, taking the resulting sequence of loops as a single loop-chain and applying Alg 2, we can see how multiple halo exchanges can be combined into a single larger message. We use this configuration to explore the performance on ARCHER2 and Cirrus clusters.

*4.1.2 ARCHER2 Results:* Figure 10 presents the execution times (min. of at least 5 runs each, CoV < 0.01) of MG-CFD on ARCHER2 for a mesh size of 8M and 24M. The reported runtime is the time taken by the main iteration loop. We have not included the constant cost for the inspection phase, which gets amortized (and negligible) for a larger number of main iterations as it is typical for real-world applications. Note the log scale of the y-axis. For both cases, we compare the original OP2, and the CA runtimes for loop-chains with loop counts $n = 2, 4, ..., 32$. For each run, we utilized the full 128 cores/node (128 MPI procs/node). Additionally, to obtain the best partitions per process, i.e. smallest MPI halos and least number of neighbors per process, we used the k-way partitioner routine from the ParMETIS library. Increasing $n$ from 2 to 32 will result in the original OP2 loops exchanging 16× more messages per neighbor. Only a single message is exchanged in the CA version. However, the grouped halo message size, $m^r$ for CA can potentially contain a maximum of $n$ halo layers. According to the data access patterns of the synthetic loop-chain, $r$ is set to 2, for our benchmarking.

**Table 3:** OP2-Hydra loop-chains with multiple halo layers ($HE_l >= 1$)

loop-chain: `weight` (loop count = 5)

| Parallel loop ($l$) | Iter. set ($S$) | Access modes and halo ext. | | $HE_l$ |
|---|---|---|---|---|
| | | $\text{mode}_{qo}$ | $HE_{qo}$ | |
| sumbwts | bnd | INC | 2 | 2 |
| periodsym | pedges | RW | 1 | 1 |
| centreline | cbnd | WRITE | 2 | 2 |
| edgelength | edges | RW | 2 | 2 |
| periodicity | pedges | RW | 1 | 1 |

loop-chain: `period` (loop count= 6)

| Parallel loop ($l$) | Iter. set ($S$) | Access modes and halo ext. | | | | $HE_l$ |
|---|---|---|---|---|---|---|
| | | $\text{mode}_{qo}$ | $HE_{qo}$ | $\text{mode}_{vol}$ | $HE_{vol}$ | |
| negflag | pedges | - | 1 | RW | 2 | 2 |
| limxp | edges | RW | 2 | READ | 1 | 2 |
| periodicity | pedges | RW | 1 | - | 1 | 1 |
| limxp | edges | RW | 2 | READ | 1 | 2 |
| periodicity | pedges | RW | 1 | - | 1 | 1 |
| negflag | pedges | - | 1 | RW | 1 | 1 |

loop-chain: `gradl` (loop count = 2)

| Parallel loop ($l$) | Iter. set ($S$) | Access modes and halo ext. | | | | $HE_l$ |
|---|---|---|---|---|---|---|
| | | $\text{mode}_{qp}$ | $HE_{qp}$ | $\text{mode}_{ql}$ | $HE_{ql}$ | |
| edgecon | edges | INC | 2 | INC | 2 | 2 |
| period | pedges | RW | 1 | RW | 1 | 1 |

We investigate the performance for the case where the number of `op_dats` exchanged remains constant at 2 per loop-chain. While this scenario is synthetic, as illustrated in [27] for structured-mesh codes, it is a prevalent case we see in real-world applications. The same was observed for Hydra, an unstructured-mesh code, but within much smaller loop-chains (see Section 4.2).

For both the 8M and 24M meshes, better runtimes can be seen at higher node counts with CA. The performance gains are also larger for higher loop counts. This aligns with the insights from the model, where the CA version is saving on the number of messages sent without an increase in the message size. Up to 35% faster runtimes can be seen with CA, compared to the original OP2. Empirical measurements for the above runs provide further evidence for the performance trends. Table 2 details the computation and communication model component factors for the 8M and 24M mesh execution with MG-CFD on ARCHER2. These were obtained by recording the message sizes and number of MPI neighbours for each tested configuration and substituting these values into the analytic model. Per-node, the amount of data communicated among the neighbours increases in the OP2 version ($\sum(2dpm^1)$) when increasing the loop count but remains constant in the CA version ($pm^r$). The amount of core computations ($\sum(S^c)$) which are performed while the halo exchange is in progress, is always smaller for the CA execution, compared to OP2. However, the number of computations over the halos which are performed after the halo exchange is always higher in the CA version. At increasing node counts, we can see the core computations reducing for both OP2 and CA and communication cost becoming dominant. For the 8M mesh, the dominance of communication and hence better performance with CA due to reduced communications starts appearing from 1 node at the 8 loop count configuration. Then again for 4 nodes at the 16 loop count. For the 24M mesh, this starts at a higher node count, 4 nodes with 4 loop count config.

**Table 4:** OP2-Hydra loop-chains with single halo level ($HE_l = 1$).

loop-chain: `vflux` (loop count = 2)

| Parallel loop ($l$) | Iteration set ($S$) | Halo exchanged datasets | $HE_l$ |
|---|---|---|---|
| initres | nodes | - | 1 |
| vflux_edge | edges | qp, xp, ql,qmu, qrg | 1 |

loop-chain: `iflux` (loop count = 2)

| Parallel loop ($l$) | Iteration set ($S$) | Halo exchanged datasets | $HE_l$ |
|---|---|---|---|
| initviscres | nodes | - | 1 |
| iflux_edge | edges | qrg | 1 |

loop-chain: `jacob` (loop count = 3)

| Parallel loop ($l$) | Iteration set ($S$) | Halo exchanged datasets | $HE_l$ |
|---|---|---|---|
| jac_period | pedges | jac, jaca | 1 |
| jac_centreline | cbnd | - | 1 |
| jac_corrections | bnd | jac | 1 |

*4.1.3 Cirrus GPU Cluster Results:* The same problem sizes were solved using CUDA code generated with OP2 plus the CA back-end on the Cirrus GPU cluster and those results are detailed in Figure 11 (CoV < 0.005). The problems were executed on nodes from 1 to 16 each node with 4×NVIDIA V100 GPUs. Each GPU was allocated 1 MPI process. Results again show similar trends to those observed in ARCHER2. However, now the performance gains are achievable even at lower node and loop counts (1.4% on 1 node, i.e. 4 GPUs). At 16 nodes, we see 42% faster runtimes for loop counts of 32.

## 4.2 OP2-Hydra

Our second application is the OP2 version of Rolls-Royce's Hydra CFD application [21, 26]. Hydra is a full-scale production application developed for modeling various aspects of turbomachinery design. It is an unstructured-mesh finite-volume solver for the compressible Reynolds-Averaged Navier-Stokes (RANS) equations in their steady and unsteady formulation (RANS/URANS). It uses a 5-step Runge-Kutta method for time-marching, with multi-grid and block-Jacobi preconditioning. Here, we again use the same 8M and 24M node NASA Rotor 37 meshes. OP2-Hydra consists of around 500 parallel loops with significantly more complex computations performed on the mesh than the loops in MG-CFD.

A number of loop-chains were identified to target CA optimizations. Table 3 and Table 4 detail six loop-chains selected for our benchmarking. The first three chains (Table 3), `weight`, `period` and `gradl` are loop-chains which require multiple layers of halos for the execution. The constituent loops, their iteration set, access modes of `op_dats` that require halo exchanges and required max halo layers for each loop are detailed in the tables. The last three loop-chains (Table 4) require only a single layer of halos. However, these loop-chains consist of the most time-consuming loops in Hydra [26]. The relative costs or the proportional contributions of the loop-chains to the total runtime of Hydra are `vflux` 18%, `iflux` 5%, `gradl` 8% and `jacob` 2%. The loop-chains, `weight` and `period` are inside the setup phase and outside the main time-marching loop.

*4.2.1 ARCHER2 Results:* Performance of each loop-chain on up to 128 nodes (16k cores) is detailed in Figure 12 (CoV < 0.08). This is the cumulative time taken by each loop-chain for 20 iterations

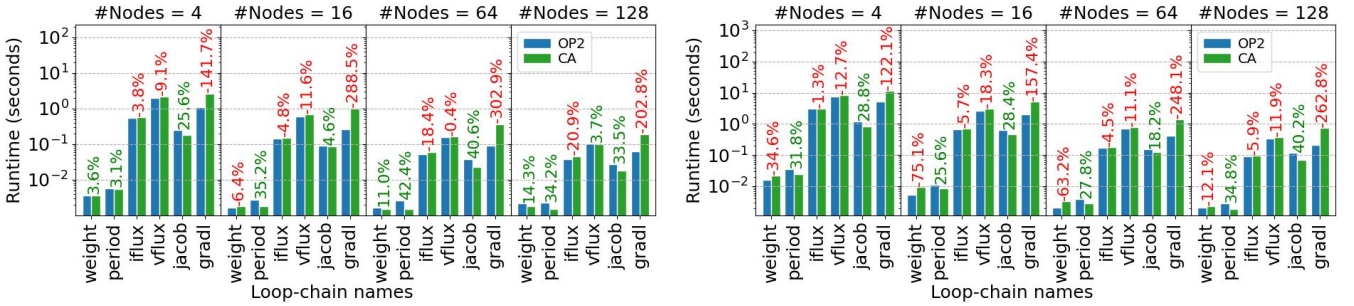



**Figure 12:** Hydra CA performance with 8M (left) and 24M (right) mesh on ARCHER 2

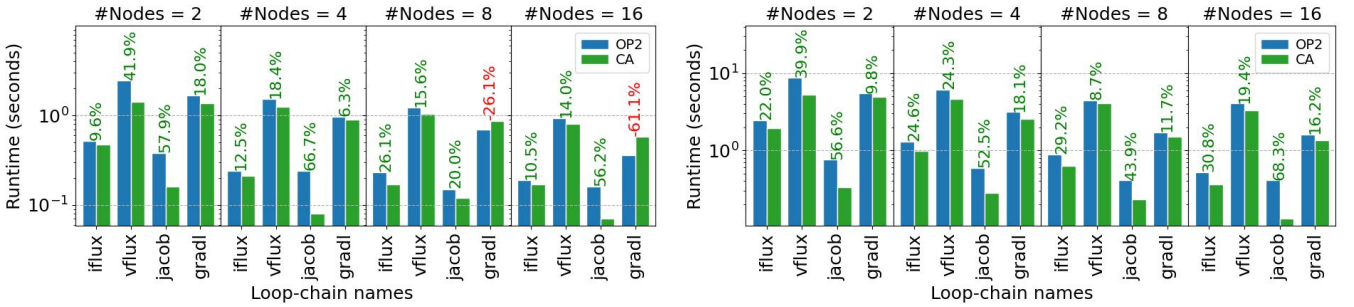



**Figure 13:** Hydra CA performance with 8M (left) and 24M (right) mesh on Cirrus

of the main time-marching loop. Hydra's default partitioner based on the recursive inertial bisection of the mesh is used in all these experiments. Results indicate that the loop-chains with the highest communication reduction, `period` and `jacob` showed performance improvements with CA - 42% and 40% on 64 nodes for the 8M problem, respectively. `weight` loop-chain showed performance gains only with 8M mesh with a maximum of 14% on 128 nodes due to its communication reduction not being adequate enough to outperform the computation increase with 24M mesh. Other loop-chains, executing them as individual OP2 loops gave the best performance. Again the insights from the model as in Table 5, align with these results where loop-chains with higher communication reduction preferably with large loop counts tend to break-even the balance of computations vs communications performance.

*4.2.2 Cirrus GPU Cluster Results:* On Cirrus (Figure 13) (CoV < 0.07), the gains from CA are larger, with loop-chains `vflux`, `iflux` and `jacob` performing up to 42%, 31% and 68% faster on the node counts benchmarked. We see a majority of loop-chains getting a speedup with CA on the GPU cluster compared to the CPU cluster ARCHER2.

Loop-chains with a higher communication reduction compared to the increased redundant computation due to added halo extensions show performance gains in Hydra. The communication reduction and the total core size for latency hiding of `period` is significantly higher than that of `weight`, giving it a higher performance gain as indicated through the model in Table 5. Loop-chains such as `iflux` and `vflux` which reduce the number of messages with a grouped halo, perform latency hiding with core execution, but with no reduction of communication are unlikely to give performance gains on CPU clusters. However, they show improvements on GPU clusters due to cutting down on host-device communications. Loop-chains such as `gradl` which cause an increase in both

**Table 5:** Hydra loop-chains (LCs) on ARCHER2: 8M Mesh - Model Components

| LC(#Loops) | #Nodes | 8M Mesh | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | OP2 | | | CA | | | LC Gain% | Comm Reduc.% | Comp Inc.% |
| | | $\sum(2dpm^1)$ | $\sum(S^c)$ | $\sum(S^1)$ | $pm^r$ | $\sum(S^c)$ | $\sum(S^h)$ | | | |
| weight(5) | 4 | 31694400 | 50206 | 37538 | 21051360 | 50184 | 135189 | 3.57 | 33.58 | 72.23 |
| | 16 | 16387200 | 13045 | 19346 | 8610624 | 13045 | 62767 | -6.42 | 47.46 | 69.18 |
| | 64 | 5121792 | 3458 | 6616 | 2154240 | 3458 | 23373 | 11.01 | 57.94 | 71.69 |
| period(6) | 4 | 51063200 | 93122 | 75076 | 3508560 | 93089 | 271347 | 3.11 | 93.12 | 72.33 |
| | 16 | 26401600 | 22652 | 38692 | 1435104 | 22652 | 128252 | 35.16 | 94.56 | 69.83 |
| | 64 | 8251776 | 5434 | 13232 | 359040 | 5434 | 47115 | 42.35 | 95.64 | 71.92 |
| vflux(2) | 4 | 59867200 | 62842 | 16674 | 59867200 | 62842 | 16674 | -9.11 | 0.00 | 0.00 |
| | 16 | 30953600 | 15403 | 7360 | 30953600 | 15403 | 7360 | -11.58 | 0.00 | 0.00 |
| | 64 | 9674496 | 3737 | 2686 | 9674496 | 3737 | 2686 | -0.44 | 0.00 | 0.00 |
| gradl(2) | 4 | 52824000 | 46548 | 27106 | 105256800 | 46537 | 123131 | -141.71 | -99.26 | 77.99 |
| | 16 | 27312000 | 11326 | 13353 | 43053120 | 11326 | 55039 | -288.51 | -57.63 | 75.74 |
| | 64 | 8536320 | 2717 | 4651 | 10771200 | 2717 | 20065 | -302.87 | -26.18 | 76.82 |
| jacob(3) | 4 | 89800800 | 3658 | 10432 | 45780800 | 3658 | 10432 | 25.59 | 49.02 | 0.00 |
| | 16 | 46430400 | 1719 | 5993 | 23670400 | 1719 | 5993 | 4.57 | 49.02 | 0.00 |
| | 64 | 14511744 | 741 | 1965 | 7398144 | 741 | 1965 | 40.61 | 49.02 | 0.00 |

communication and computation tend to degrade the performance even with a sufficiently large number of latency hiding core computations and grouped halos. On the other hand, loop-chains such as `jacob` which reduce communication with latency hiding and no computation increase always tend to give performance gains.

In general, there is a message size increase and message count decrease in the CA version. However, it does not change the load balance of the tasks divided among the processes compared to the OP2 version. Message exchange between processes is always associated with message packing and unpacking costs. CA/OP2 message packing cost ratio is equivalent to the CA/OP2 total halo

message sizes ratio. However, the OP2 version does not suffer from a message unpacking cost since the messages related to a particular dataset are directly copied to the relevant dataset array when receiving the message. But there is an additional message unpacking cost for the CA version when copying the data elements of multiple datasets received in the same message to relevant dataset arrays. However, this unpacking cost becomes negligible due to the chunk memcopy operations performed on the received messages in the CA version compared to the multiple message exchange cost of the OP2 version.

## 5 CONCLUSION

In this paper, we have presented the implementation of data movement-reducing and communication-avoiding optimizations for large-scale unstructured-mesh applications. The runtime analysis of a loop-chain for creating sparse-tiling schedules and their extension to distributed-memory parallelization as detailed by Luporini et al. [17] is implemented through the OP2 DSL. The new back-end in OP2 for communication-avoidance (CA), codifies these techniques such that they can be applied automatically to any OP2 application. The careful trade-off with increased redundant computation in place of data movement was analyzed for distributed-memory parallelization using a representative CFD mini-app and Hydra, a production CFD code from Rolls-Royce. The performance trade-offs were modeled analytically, examining the determinants and characteristics of a given unstructured-mesh loop-chain that can lead to performance benefits with CA techniques. Performance benchmarks were carried out on a large HPE Cray EX system and an NVIDIA V100 GPU cluster.

Results show that loop-chains with higher communication reduction compared to its computation increase, are generally the ones with higher loop counts, provide increasingly bigger performance gains. This is true for both CPU and GPU clusters. For such loop-chains, the balance of computations to communications also begins to favour the CA-optimized version at larger node counts. We see 30 – 65% runtime reductions. However, identifying such profitable loop-chains would be the challenge in real-world applications. As we see from Hydra, several such chains could be targeted. Nevertheless, the use of OP2's new CA back-end allowed us to elicit the best performance for Hydra without changing the large science source, maintaining performance portability. Future work will explore the performance of further large-scale production applications on a wider range of hardware with code generated through OP2. We will also move to further automate the code-gen process with lazy-evaluation [27]. The new CA back-end for OP2 is available as open-source software at [2] and the loop-chained MG-CFD is available at [1].

## ACKNOWLEDGMENTS

This research was supported by Rolls-Royce plc., and by the UK EPSRC (EP/S005072/1 – Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems – ASiMoV). Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme (INF/R1/1800 12). This work used the ARCHER2 UK National Supercomputing Service and the Cirrus UK National Tier-2 HPC Service at EPCC funded by the University of Edinburgh and EPSRC (EP/P020267/1).

## REFERENCES

[1] 2023. *MG-CFD-app-OP2 github repository.* https://github.com/warwick-hpsc/MG-CFD-app-OP2/tree/feature/icpp2023-commavoid
[2] 2023. *OP2 github repository.* https://github.com/OP-DSL/OP2-Common/tree/feature/icpp2023-commavoid
[3] Accessed Aug 2022. Cirrus. https://www.cirrus.ac.uk/
[4] Accessed Nov 2022. ARCHER2. https://www.archer2.ac.uk
[5] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations. 40, 2 (2014). https://doi.org/10.1145/2566630
[6] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 33, 12 pages. https://doi.org/10.1145/2503210.2503289
[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.
[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (June 2008), 101–113. https://doi.org/10.1145/1379022.1375595
[9] James Demmel, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Michael Mahoney. 2020. *Prospectus for the Next LAPACK and ScaLAPACK Libraries: Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC).* Technical Report 297, ICL-UT-20-07.
[10] James W. Demmel, Laura Grigori, Ming Gu, and Hua Xiang. 2015. Communication Avoiding Rank Revealing QR Factorization with Column Pivoting. *SIAM J. Matrix Anal. Appl.* 36, 1 (2015), 55–89. https://doi.org/10.1137/13092157X
[11] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. *SIGPLAN Not.* 51, 8, Article 5 (feb 2016), 12 pages. https://doi.org/10.1145/3016078.2851157
[12] Lester Kalms, Tim Hebbeler, and Diana Göhringer. 2018. Automatic OpenCL Code Generation from LLVM-IR Using Polyhedral Optimization *(PARMA-DITAM '18)*. Association for Computing Machinery, New York, NY, USA, 45–50. https://doi.org/10.1145/3183767.3183779
[13] Penporn Koanantakool. 2017. *Communication Avoidance for Algorithms with Sparse All-to-all Interactions.* PhD dissertation. University of California, Berkeley.
[14] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. J. Kelly, G. R. Mudalige, B. Van Straalen, and S. Williams. 2013. Loop Chaining: A Programming Abstraction for Balancing Locality and Parallelism. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 375–384. https://doi.org/10.1109/IPDPSW.2013.68
[15] Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. 2016. Efficient Mesh Management in Firedrake Using PETSc DMPlex. *SIAM Journal on Scientific Computing* 38, 5 (2016), S143–S155. https://doi.org/10.1137/15M1026092
[16] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz
[17] Fabio Luporini. 2016. *Automated Optimization of Numerical Methods for Partial Differential Equations.* PhD dissertation. Imperial College London.
[18] Fabio Luporini, Michael Lange, Christian T. Jacobs, Gerard J. Gorman, J. Ramanujam, and Paul H. J. Kelly. 2019. Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling. *ACM Trans. Math. Softw.* 45, 2, Article 17 (May 2019), 30 pages. https://doi.org/10.1145/3302256
[19] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (apr 2020), 28 pages. https://doi.org/10.1145/3374916
[20] Gihan Mudalige, Mike Giles, I.Z. Reguly, C. Bertolli, and Paul Kelly. 2012. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *2012 Innovative Parallel Computing, InPar 2012*, 1–12. https://doi.org/10.1109/InPar.2012.6339594
[21] Gihan R. Mudalige, Istvan Z. Reguly, Arun Prabhakar, Dario Amirante, Leigh Lapworth, and Stephen A. Jarvis. 2022. Towards Virtual Certification of Gas Turbine Engines With Performance-Portable Simulations. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 206–217. https://doi.org/10.1109/CLUSTER51413.2022.00034
[22] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGPLAN Not.* 50, 4 (mar 2015), 429–443. https://doi.org/10.1145/2775054.2694364
[23] NVIDIA. Accessed March 2023. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect
[24] A. M. B. Owenson, S. A. Wright, R. A. Bunt, Y. K. Ho, M. J. Street, and S. A. Jarvis. 2020. An unstructured CFD mini-application for the performance

prediction of a production CFD code. *Concurrency and Computation: Practice and Experience* 32, 10 (2020), e5443. https://doi.org/10.1002/cpe.5443 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5443 e5443 cpe.5443.

[25] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211

[26] Istvan Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H.J. Kelly, and David Radford. 2016. Acceleration of a Full-Scale Industrial CFD Application with OP2. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (May 2016), 1265–1278. https://doi.org/10.1109/tpds.2015.2453972

[27] István Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. 2018. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 873–886. https://doi.org/10.1109/TPDS.2017.2778161

[28] Istvan Z. Reguly, Andrew M. B. Owenson, Archie Powell, Stephen A. Jarvis, and Gihan R. Mudalige. 2021. Under the Hood of SYCL – An Initial Performance Analysis with An Unstructured-Mesh CFD Application. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 391–410. https://doi.org/10.1007/978-3-030-78713-4_21

[29] Michelle Mills Strout, Fabio Luporini, Christopher D. Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J. Ramanujam, and Paul H.J. Kelly. 2014. Generalizing Run-Time Tiling with the Loop Chain Abstraction. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1136–1145. https://doi.org/10.1109/IPDPS.2014.118

[30] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1989493.1989508

[31] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software (ICMS'10) (LNCS 6327)*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer-Verlag, 299–302.

[32] M. Wolfe. 1989. More iteration space tiling. In *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. 655–664. https://doi.org/10.1145/76263.76337

[33] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. 2015. CA-SVM: Communication-Avoiding Support Vector Machines on Distributed Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 847–859. https://doi.org/10.1109/IPDPS.2015.117