UnSNAP: a mini-app for exploring the performance of deterministic discrete ordinates transport on unstructured meshes.

Workshop paper: WRAp 2018

Tom Deakin*, Simon McIntosh-Smith*, Justin Lovegrove†, Richard Smedley-Stevenson† and Andrew Hagues†
*Department of Computer Science, University of Bristol, Bristol, UK
Email: {tom.deakin}{S.McIntosh-Smith}@bristol.ac.uk
†Atomic Weapons Establishment, Aldermaston, UK.

***Abstract*—Solving the deterministic discrete ordinates neutral particle transport equation is a computationally expensive application. On an unstructured mesh, the discontinuous Galerkin finite element method is used for discretisation of the spatial domain. Additionally, an upwind dependency is applied forming wavefront sweeps across the spatial mesh for each iteration of the solve. We present a new mini-app, UnSNAP, which can be used to investigate the performance of arbitrarily high-order finite element unstructured transport on modern architectures. A new schedule appropriate for such architectures is presented. Finally, we show performance results for the mini-app on CPUs with high numbers of cores.**

***Index Terms*—deterministic discrete ordinates transport, finite element, discontinuous Galerkin, sweep, unstructured mesh**

I. Introduction

The deterministic discrete ordinates neutral particle transport equation is a Boltzmann balance equation that models how neutral particles move and interact through a mesh of materials of varying properties. The solution of this transport equation is computationally intensive, mainly resulting from the inversion of the streaming-collision operator and the data dependency imposed in the construction of the numerical method for approximate solution. The solution to the equation is of high dimensionality: space, energy, direction and optionally time; and as such demands a large memory footprint, often using the full capacity of the available memory. For machines which must run this class of application, it is often the memory demands of the neutral particle transport codes that determine the memory capacity of the system. Although one could select one of the many off-the-shelf linear solver libraries to solve this equation, due to the high dimensionality and convergence properties, it is typical to write codes which solve this equation specifically. The solution of transport is relatively expensive, as can be seen on a survey of the usage of United States Department of Energy supercomputing systems where it was estimated that 50-80% of simulation time is taken up by solution of the transport equation [1].

The Discrete Ordinates (S_n) Application Proxy (SNAP) from Los Alamos National Laboratory was developed to understand the performance of a transport code [2]. The mini-app uses artificial problem data which is auto-generated based on input parameters. SNAP operates on a structured, regular Cartesian grid and utilises a finite difference spatial discretisation method. The performance of SNAP has been extensively studied [3]–[5].

Unstructured meshes are however commonly used in computational science. They offer better representation of complex geometries than structured meshes. An unstructured mesh must detail the connectivity of these cells so that neighbours are listed explicitly rather than defined implicitly as per a structured mesh. However, the simple finite difference method used for structured meshes is unsuitable for solving the equation on unstructured meshes, where each cell is represented as a hexahedron (a potentially deformed cube). A higher order finite element discretisation is used instead.

In this paper we investigate the performance of the solution of the transport equation on an unstructured mesh using the discontinuous Galerkin finite element method. The SNAP mini-app is extended to support unstructured meshes solved using this finite element method.

In particular, we make the following contributions:

- We have developed a discontinuous Galerkin finite element implementation of the deterministic discrete ordinates transport mini-app SNAP, nicknamed *UnSNAP*. This will enable a holistic approach to understanding the performance of such codes on advanced architectures.
- Arbitrarily high order Lagrange elements are used within the mini-app which enables us to investigate the performance characteristics of both low and high order finite elements. To this end, we evaluate the relative cost of the local matrix assembly and solution using a simple hand-written small dense linear solve and the Intel Math Kernel Library.

- A sweep schedule is presented which allows for high levels of parallelism on each process within the MPI distributed spatial domain. This will ensure that sufficient concurrent work is available to leverage the increasing parallel demands of advanced multi- and many-core architectures.
- Initial performance results of the UnSNAP mini-app are presented on the latest CPUs which have a high core count. These results highlight the areas of future study of the performance of this algorithm.

A. *Related work*

The Tycho 2 mini-app implements linear discontinuous Galerkin finite elements in the solution of the transport equation on an unstructured grid of tetrahedral elements [6]. The focus of our study is on hexahedral meshes — the complexities of sweeping triangular meshes are well known [7]. Tycho 2 has also been designed to study the scheduling properties of the transport equation, building on the work of Pautz [8]. The schedules proposed by Pautz are designed to ensure that the upwinding dependency is maintained and that dependencies are satisfied as quickly as possible as the sweep progresses across the mesh. Our port of SNAP to unstructured meshes will allow for a holistic study of both the performance of the solve as well as the scheduling of parallel work.

The UMT2013 mini-app performs transport calculations on an unstructured grid but uses a unique upstream corner-balance spatial discretisation [9]. It is a rather large mini-app at approximately 50,000 lines of Fortran, C and C++ and is therefore not flexible enough to be used as a research vehicle for our purposes. In particular, it is not feasible to replace the corner-balance method with the finite element method in this code. For comparison, SNAP consists of 4,000 lines of Fortran (according to their counting script), and the UnSNAP mini-app introduced in this paper consists of 3,000 of C++.

II. Spatial discretisation of the transport equation

The stationary transport equation is solved for the unknown angular flux ψ in space $\vec{r}$, angle $\hat{\Omega}$ and energy E. The total material cross sections are given by σ, with the scattering cross sections σ_{s}. The cross sections relate to the material properties and describe the probability that a neutral particle will interact with the material. The total cross section σ is the chance of an interaction occurring from absorption or a change in direction or energy. The scattering cross section σ_{s} describes the chance of the interaction resulting in a change of direction and/or energy.

$$\hat{\Omega} \cdot \vec{\nabla}\psi(\vec{r}, \hat{\Omega}, E) + \sigma(\vec{r}, E)\psi(\vec{r}, \hat{\Omega}, E) = \\ q_{\text{ex}}(\vec{r}, \hat{\Omega}, E) + \int dE' \int d\Omega' \sigma_{\text{s}}(\vec{r}, E' \to E, \hat{\Omega}' \cdot \hat{\Omega})\psi(\vec{r}, \hat{\Omega}', E')$$

The left hand side of the equation describes the loss of particles; the streaming term captures the loss of particles due to their movement out of the domain, and the loss of particles due to interaction with the material is described in the collision term. The right hand side describes the gain in particles; a fixed source contribution is given by q_{ex} and represents a gain in particles that come from outside the physics modelled by the equation; these particles may come from the material itself or from outside the boundary. Added to this is the gain of particles into a particular direction and energy and is given by the scattering source. It is often convenient to notate the right hand side as $S(\vec{r}, \hat{\Omega}, E)$.

The equation is discretised in angle using discrete ordinates and in energy using the multi-group approximation. A detailed explanation of the derivation and discretisation of the transport equation can be found in Lewis and Miller [7] and readers unfamiliar with transport are referred to this text.

The solution of the transport equation is via simple iterations on the scattering source, and as such is an iterative method. Jacobi iterations are used to solve the group-to-group coupling in the source term.

A. *Finite difference in space*

For the finite difference discretisation of the transport equation, simple diamond difference equations are defined and included with the transport equation to create a system of equations. These extra equations state that the value of the angular flux ψ in the centre of each cell is equal to the average of the solution on each opposite pair of sides.

The finite difference equations are substituted into the streaming operator, forming an *upwinding* dependency, allowing the equation to be solved at the cell centre given known values on the boundaries of adjacent cells. This data dependency causes a *sweep* across the mesh for each angle. The diamond difference equations are such that the cell centred value is equal to the average of the top and bottom edge values and simultaneously equal to the average of the left and right edge fluxes. A sweep is performed for each angle $\hat{\Omega}$ with the incoming cell edges/faces chosen appropriately to respect the angle direction through the cell.

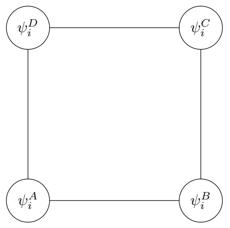
B. *Finite element in space*

The discontinuous Galerkin finite element method is used to discretise the spatial domain into Lagrange hexahedral elements (cubes). The finite element method involves multiplying the transport equation by a test function and integrating over the area of the element (cell), integrating the gradient term by parts. An assembly of one small linear system per element, per group, per angle is formed. Table I shows the matrix size for a number of finite element orders, along with the storage space required for the matrix using double precision floating point values (FP64).

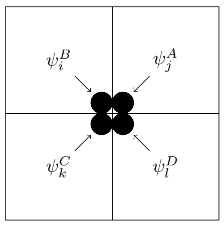
The angular flux is approximated as a linear combination of trial basis functions which exist at *nodes* within the cell. For linear elements, each of the trial functions is associated with a vertex of each cell, as shown in Figure 1a. Higher order elements add additional nodes on the edges, faces and within the volume of the cell. A value for the angular flux ψ is computed at each of the nodes in each cell. The

TABLE I: Size of local matrix for different finite element orders

Order	Matrix size	FP64 footprint (kB)
1	8×8	0.5
2	27×27	5.7
3	64×64	32.0
4	125×125	122.1
5	216×216	364.5



(a) Linear nodes



(b) Four nodes share the same physical location

Fig. 1: 2D discontinuous elements

discontinuous nature of the discretisation means that where points on adjacent elements share one physical position as in Figure 1b, they are treated as separate unknowns in the global finite element solution; at mesh convergence we would expect these values to be equal.

An upwinding scheme equivalent to that employed in the finite difference method is used to determine which fluxes are known from neighbouring cells and included on the right hand side b, and those unknowns for which coefficients are included in the matrix A.

C. Trade-offs between the methods

As can be seen from the descriptions of the spatial decompositions, the finite difference (FD) method is relatively simple compared to the finite element method (FEM). The FD method involves the inversion of the transport equation and evaluation of the diamond difference relations for the outgoing fluxes. The number of floating point operations for each diamond difference relation is just a single multiply-add. In contrast, the FEM requires many more floating point operations, in particular for the solve of the small matrices.

Using LAPACK's `dgesv` for solution requires $0.67N^3$ operations; in 3D where $N = 8$ this is over 300 FLOPS. Higher order elements will produce many more FLOPS still. The assembly of the linear system for each cell/angle/group also requires additional floating point operations. Therefore much more work is required to solve the flux for the finite element method on the same sized grid.

The FEM stores a solution for the angular flux on each node of the cells in the grid for each angular direction and energy group. The FD method on the other hand has a single value per grid cell for each angular direction and energy group. Neighbour fluxes are stored as temporary variables and are not significant to the memory footprint. The memory overhead for a 3D mesh for linear finite elements is therefore 8 times that of the FD method. Discrete ordinates transport has an enormous memory footprint, proportional to the product of the size of each dimension in the problem. Therefore for a *fixed mesh size* the finite element solution requires much more memory than the finite difference solution.

However, the FEM approach provides a higher-order accuracy solution than FD. The linear finite elements used here are third-order accurate, whereas the finite difference approach is second-order accurate. Higher order elements will provide increased levels of accuracy with further reduced error. In practice therefore for a *given error*, the finite element method allows the use of larger cells and thus coarser grids; fewer cells are required to provide a suitable answer, resulting in a reduced memory footprint.

Although both methods may be used on a structured grid (our previous work explored FEM on a structured mesh [3]), the FEM is also applicable to unstructured meshes. The node positions in the elements allow the cell to be deformed to create an irregular shape rather than a regular cube. The method even supports elements with curved edges and faces. As such more complicated geometries may be constructed.

III. Parallel implementation

The SNAP mini-app is used as the basis for our unstructured version, which we have nicknamed *UnSNAP*. UnSNAP uses the same artificial data, source calculation and iteration structure as SNAP. The unstructured mesh is formed by first forming the original SNAP mesh but storing it in an unstructured format, maintaining appropriate lists of cell-to-cell dependencies in a new mesh data structure. The reliance on this data structure for resolving neighbouring element connectivity is a key differentiator between the treatment of a structured and unstructured grid. Each cell is based on a hexahedral element (like a cube); each face has a single neighbouring element. To ensure that the mesh is truly treated as unstructured, a new input option allows the mesh to be twisted slightly along a single axis, and therefore each cell is no longer a perfect cube.

The spatial mesh needs to be distributed between MPI processors, and the original SNAP approach to this is used. A 2D decomposition of the 3D domain is performed, similar to the KBA style decomposition for a structured grid [10], as this was shown to often be optimal for sweeping unstructured meshes [11]. This decomposition occurs during the construction of the mesh derived from the structured mesh, and so more complex mesh partitioning could be avoided.

SNAP considers that energy groups may be swept concurrently, and we retain this assumption for UnSNAP. SNAP also allows angles within an octant to be solved concurrently, but octants are swept in turn. This is also true for a 3D unstructured grid.

An overview of the algorithm is shown in Figure 2. For each angular direction in the problem, a sweep schedule is constructed by following the outgoing faces of the elements. This schedule can then be followed, where for each element

```
for all angular directions do
  for all elements in angle schedule do
    for all energy groups do
      Assemble matrix $A$ from $S_n$ quadrature, cross sections and element basis functions
      Assemble vector $b$ from source terms, element basis functions and upwind neighbour $\psi$
      Solve $A\psi = b$
    end for
  end for
end for
```

Fig. 2: Pseudocode for solving the transport equation

the angular flux for all energy groups can be calculated using the finite element method. The central computation at the heart of the sweep requires the construction and solution of a small dense linear system: assembling a matrix A and a vector b to form $A\psi = b$ and solving for the angular flux, ψ.

A. Sweep schedule

The solution of the transport equation requires a *sweep* of the spatial domain for each angular direction. Unlike many grid-based methods, all cells cannot be solved concurrently due to an upwinding dependency between cells, and therefore a schedule is needed to determine the order in which cells can be solved. For an unstructured mesh, the order in which the cells must be computed may be unique for each angular direction; for a structured mesh the order is identical for all angular directions in a given octant. This sweep schedule forms a directed graph, which should be acyclic; for our first version of UnSNAP no mechanism to break cycles is implemented as this is a focus of future work. The order in which the graph is traversed, and thus the order in which the cells are updated, is determined by the choice of a *sweep schedule*. As the graph is distributed between processors according to the spatial decomposition, one must consider the scheduling of work locally (on-process) and globally (between-process).

1) Global scheduling: A parallel block Jacobi schedule is chosen for processor-to-processor coupling. This results in a halo exchange every iteration in order to share the outgoing data between processor domains. The convergence rate will therefore depend on the number of MPI ranks selected. Note that each process can begin computation on its own subdomain concurrently, unlike with the KBA schedule in the SNAP mini-app where processors must wait to begin work.

Garrett has previously investigated the convergence properties of a parallel block Jacobi spatial decomposition where it was found that this approach, as expected, did not converge as quickly as sweeps which respect the upwinding dependency between processor boundaries [6]; however this was only tested for small scale runs and the performance of the solve itself was not considered. One of UnSNAP's goals is therefore to enable a robust investigation on a variety of architectures into the effect of the Jacobi schedule on the time to solution rather than simple scalability or convergence rates. Additionally, the focus on enabling on-node parallelism will allow for reduced numbers of MPI ranks which should limit the degradation in convergence rates associated with a large number of Jacobi blocks resulting from high numbers of MPI ranks.

The other schedules proposed by Garrett and Pautz (see Section I-A) are designed around a lightweight task of solving a single angle-group-element. Priorities are assigned to each task, with the optimality of the schedules relying on having many of these tasks available in order to reduce idle time. Historically these schedules were developed for the IBM Blue Gene/Q supercomputer, which consisted of a very large number of nodes with good nearest-neighbour communication and low cost (global) synchronisation [12]. The nodes themselves consisted of simple, energy efficient cores with a small amount of memory per core, and as such applications were required to run at very large core counts in order to achieve high aggregate performance. This trend for many energy efficient cores in supercomputer design has waned, with the current state-of-the-art supercomputers being designed around heavyweight nodes consisting of a high number of complex cores and often coupled with accelerators such as GPUs. Therefore, the number of nodes to which an application must scale is much less than on a Blue Gene/Q. Additionally, each node must be supplied with sufficient parallel work in order to leverage the large amount of computing resource available. The schedules of Pautz are designed to be serial on each node and therefore have limited applicability on the latest hardware. As such, although the numerical convergence properties of a block Jacobi schedule are reduced in comparison to the Pautz schedules, they provide us with a baseline performance for heavyweight computational nodes. It is the subject of future work to explore other schedules.

2) Local scheduling: For the scheduling of work on the nodes, the standard sweep order is followed without breaking any dependencies (as has been done for the block Jacobi scheme as previously stated). Each process therefore computes a sweep schedule for its own local domain, with each angle in the S_n quadrature potentially having a unique sweep schedule. The schedule used in our implementation calculates the *tlevel* of each element for each angle (see Pautz for a definition [13]), and places cells with the same tlevel in a *bucket*. The buckets represent the cells on each hyperplane/wavefront as the sweep progresses across the mesh. For each angle, elements where the incoming faces are satisfied by problem or neighbouring boundary conditions can be solved initially and form the contents of the first bucket. Counters on each of the neighbouring cells on the outgoing faces of these elements are then incremented to show that their upwind dependency is satisfied. When a cell's dependency counter has been incremented sufficiently to match the number of incoming faces for this angular direction, it can be placed in the next bucket. This process continues until all neighbours have been followed. Note that we have assumed cyclic dependencies do not occur in our first version of UnSNAP.

B. Sweep schedule concurrency

The concurrency scheme for the processing of this local sweep schedule will determine the available parallelism on each MPI rank. Cells within the bucket may be computed concurrently, but the buckets must be processed in order. Energy groups may be computed concurrently, along with angles within the same octant. The assembly and solution of the finite element matrix for each angle-group-cell introduces an additional level of potential concurrency (compared to the structured solve). This concurrency is formed from constructing the matrix and right hand side vector, where the entries can be updated in parallel, as well as the solution of the small linear system. It is clear that the matrix size (equivalently element order) determines the amount of available concurrency.

This block Jacobi schedule allows for each MPI process to be implemented in a highly parallel way. This is in contrast to the Pautz schedules which are designed to propagate outgoing data across the MPI processor grid as soon as possible so as to reduce idle time. However, this means that each MPI rank must be capable of processing fine-grained tasks in serial. Aggregating larger tasks to create more parallelism on the node destroys the nice properties of the sweep in reducing processor idle time. As such, the schedule presented in this paper is more applicable to the 'fat-nodes' which form the basis of modern supercomputer design. Indeed, the available concurrency in the unstructured mesh is the same as for a structured mesh, with the additional level introduced by the finite element matrix solution. Our previous work has shown that maximal concurrency was required to leverage high levels of performance of structured transport on GPU architectures [4], [5], and so retaining this high level of concurrency for unstructured meshes is also desirable.

The Pautz schedules also determine a priority for each task. For our schedule this is redundant as the high level of concurrency means that 'all' the outgoing data is made available at each stage of the sweep and so no choices need to be made as to which should be computed first to maintain low idle time.

C. Performance characteristics

The runtime is dominated by the assembly and solve of the local linear system for each angle/element/group. The assembly of the matrix requires reading from 13 different arrays to populate the linear system. These arrays are of different dimensionalities and so the reuse pattern is complex: for example the S_n cosines have only an angular index and so the data can be reused for all elements and groups, whereas the pre-computed integration of basis function pairs are unique to each element and shared across angles and groups. The constructed matrices are small and so should be cached.

Vectorisation of the implementation along the element nodes required using the OpenMP `simd` constructs as the tested compilers were not able to auto-vectorise due to perceived dependencies: a common issue for C/C++ codes where there is a lack of true Fortran-style multi-dimensional array support. The assemble/solve routine therefore consists of a number of vectorised loops, but it otherwise has limited branching. A data dependent branch occurs for determining if the upwind contribution should be added to the matrix or right-hand side.

Therefore the rate limiting factor is therefore going to be memory access. For linear elements, we previously showed that the Computational Intensity for this routine under the Roofline model was low (0.25 FLOPs per byte) [3]. For higher order elements, the number of FLOPs will increase as described in Section II-C. The access patterns of the arrays results in heavy reuse of small arrays coupled with limited reuse (streaming access) of a very large array (angular flux). Although FP64 values are currently used, it may be possible to exploit lower precision in different parts of the algorithm; this is one avenue for future work that UnSNAP enables.

IV. Results

Initial performance results of the UnSNAP mini-app are presented. Different concurrency schemes for computing the solution according to the sweep schedule introduced in Section III-A are investigated. Additionally, the performance of the solution of the local dense linear system is explored by comparing a hand-written Gaussian Elimination route with the `dgesv` routine from the Intel Math Kernel Library.

Results were recorded on a single node of a Cray XC40 supercomputer, 'Swan'. The node is a dual-socket configuration using Intel Xeon Platinum 8176 (Skylake) 28-core at 2.1 GHz processors with 192 GB of DDR4-2666 memory. The Intel 2018 compiler was used for all results.

A. Parallel schemes

Parallelism within the sweep on each MPI rank can be found in the following dimensions:

- Element nodes, in assembling and solving the linear system
- Energy groups
- Elements in the same 'bucket' in the sweep order (those on the same wavefront plane)
- Angles within an octant

The implementation utilises compiler vectorisation over the element nodes during the assembly and system solve to ensure that the CPU vector units are utilised. The remaining dimensions may therefore be threaded to allow for concurrent execution. Element indices are found via an indirection to the bucket, and so the order in which elements are accessed does not follow a simple, regular pattern; their location in memory will also depend on the element numbering scheme. The angular and energy dimensions offer more predictable access as they can be organised in a regular manner in the various arrays. As such, it is interesting to explore the various combinations of loop order and their matching data layouts, along with which of these are threaded; such a methodology has been previously explored with the KRIPKE mini-app [14]. A single MPI rank was run on the dual-socket system, with the number of OpenMP threads changes up to the total number of physical cores (no hyperthreading).

The following problem was used:

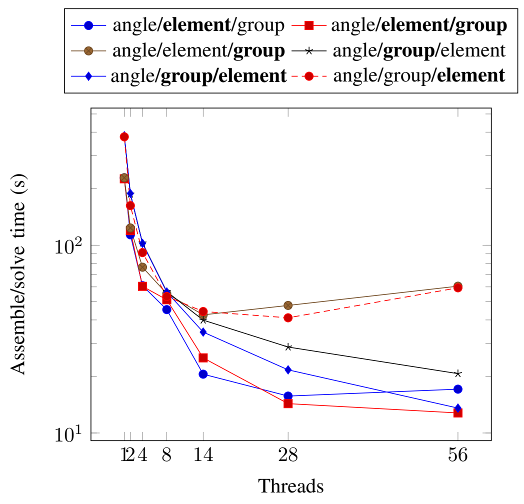


Fig. 3: Thread scaling of parallel sweep for different loop orderings for linear elements

- 16 x 16 x 16 elements
- 36 angles per octant with isotropic scattering
- 64 energy groups, with Source and Material 'Option 1'
- Linear and cubic finite elements
- Mesh twisting of up to 0.001 radians
- 5 inners and 1 outer

Note that the iteration count selected results in too few iterations to converge the solution; it does however ensure that there are multiple iterations of the solve for timing robustness and that the iteration count is constant.

The UnSNAP source code was updated so that the order of the loops for the assemble/solve routine are arranged in various ways. The storage arrays of the angular flux, scalar flux and source terms were likewise updated to match the loop ordering. OpenMP directives were added to parallelise different loops. The results for linear elements are shown in Figure 3 and for cubic elements in Figure 4. The graphs show the run time of the assemble/solve routine for varying numbers of threads. The absolute run times are displayed rather than speedups to allow a comparison of the different parallel schemes. The legend describes the order of the loops from outermost on the left to innermost on the right (the element nodes always present as a further inner loop and are not shown in the legend). The bold font denotes the loop was parallelised with OpenMP. For example, the option 'angle/**element**/group' denotes that the loop order processes all groups within each element (according to the schedule) for each angle resulting in the loop order showing in Figure 2, with the elements in the bucket being processed in parallel using threads.

1) Linear elements: For the linear elements shown in Figure 3, at high thread counts the fastest scheme threads over elements in the schedule buckets as well as energy groups. Notice that in general the 'angle/**group/element**' scheme is slower than the 'angle/**element/group**' scheme, although they both have similar run time when using all cores. Note that although both these schemes have the same parallel work, the order of the iterations of these two loops differ according to the OpenMP `collapse` clause semantics which impacts on the order of memory accesses. The loop iterations are collapsed and lexicographically ordered by the inner-most loop. As an example of these semantics, consider looping over the alphabet (inner-loop) multiple times whence the iteration space would be ordered as: $(A,1),(B,1),\ldots,(Z,1),(A,2),(B,2),\ldots$. The order in which memory is accessed therefore follows the serial ordering within each thread, with each thread starting at a different position in this sequence.

The loop over elements in the bucket results in an indirect memory access. The same element index is used for all entries in each SIMD instruction and so this indirection does not cause vectorisation issues such as gathers. The element nodes are organised contiguously in memory for each element as the fastest moving index and so it is always stride one access to load such data. Energy groups are accessed in a regular fashion, and again this iteration space is not vectorised over. The ordering of the array extents determines the strides between accesses of adjacent values in the iteration space.

Taking the 'angle/element/**group**' scheme running at 56 threads, each thread is likely to be accessing nearly adjacent cache lines. Similarly, for the 'angle/**group**/element' scheme, the distance in memory between adjacent element indices is only 64 bytes. As the element index is calculated indirectly from the schedule buckets, threads will not be able to stream data as access will be somewhat random. Access to adjacent energy groups however is predictable in nature, requesting each group index in ascending order resulting in a fixed stride (number of nodes times the number of elements). Therefore, having energy groups moving faster than elements in the array extents means that access remains predictable for larger amounts of memory; 4 kB of stride per element access rather than only 64 bytes. As such, the unstructured nature of the mesh resulting in this non-contiguous indexing of the elements is likely to be somewhat mitigated. Indeed, the 'angle/**element**/group' scheme is often the fastest scheme at smaller thread counts and the schemes with the 'angle/group/element' data layout do not show compelling performance in comparison.

The benefit of collapsing the element and energy group loops can be seen where the number of elements in the schedule bucket is small. Without collapsing the loops this results in limited parallel work just in the element domain. For small thread counts (say 14), the resulting work imbalance will be small whereas at high thread counts (say 56) the lack of work limits the scalability, as can be seen on the 'angle/**element**/group' scheme in Figure 3. By collapsing the threads, more parallel work is made available even for small bucket sizes shown by the scaling of the 'angle/**element/group**' scheme.

2) Cubic elements: Similar conclusions can be drawn from the cubic elements shown in Figure 4. The amount of work has increased compared to the linear elements as a result of the increase in the number of nodes. The fastest scheme is again 'angle/**element/group**'; the number of elements in

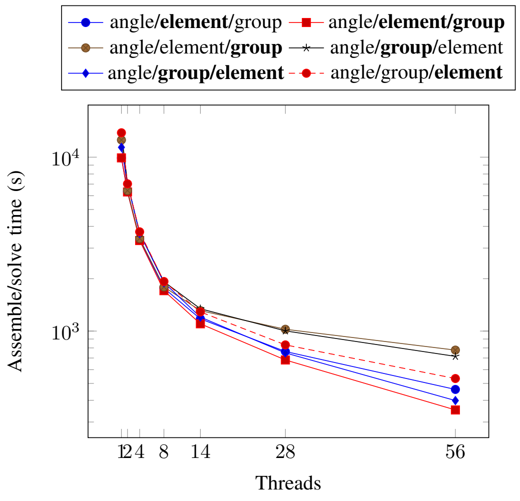


Fig. 4: Thread scaling of parallel sweep for different loop orderings for cubic elements

each bucket is the same for both cubic and linear elements and so the additional parallelism from the energy groups aids in reducing the runtime. As with linear elements, without parallelising over groups, a similar runtime is attained at lower thread counts for the 'angle/**element**/group' and 'angle/**element/group**' schemes, but when utilising all cores available the extra parallelism assists.

These cubic elements have 64 nodes, resulting in a 32 kB stride between adjacent elements in the fastest data layout ('angle/element/group'); note that this is also the capacity of the L1 data cache on these processors. As such the performance of the 'angle/group/**element**' scheme still attains reasonable performance for cubic elements on all threads, unlike for linear elements where the stride between adjacent elements is equivalent in size to a cache line resulting in poor performance.

3) Summary: Two factors in the parallel scheme were required for good performance on high numbers of cores. Firstly, sufficient parallelism was required. This was attained by parallelising across both the independent elements contained within each bucket in the sweep schedule and the energy groups. This ensures that enough parallel work exists when the number of independent elements is small. Secondly, ensuring a large gap in memory between adjacent elements. This was achieved by organising the data arrays so that the extents were such that energy groups had a faster moving stride than the element indexing. It is hoped that this along with the additional problem dimensions of the element nodes somewhat mitigate the expected effects of unstructured access to the mesh data.

Threading over angles within the octant is one parallel scheme which is not displayed in these figures. Due to the scalar flux reduction over this dimension the update of the scalar flux array must be made atomically by each thread. Both OpenMP `atomic` and `critical` regions were tested, although neither allowed for thread scaling. Indeed, the runtime was increasing with the thread count.

B. Matrix solve

Once the local matrix has been assembled and the right hand side constructed using source data and upwind neighbouring element data, the linear system must be solved. UnSNAP offers both a hand-written direct Gaussian elimination solver and the ability to utilise a LAPACK `dgesv` routine. Appropriate vectorisation of the Gaussian elimination was ensured by using OpenMP `simd` constructs.

The following problem size was run using flat MPI (one rank per physical core) on a single dual-socket node:

- 32 x 32 x 32 elements
- 10 angles per octant with isotropic scattering
- 16 energy groups, with Source and Material 'Option 1'
- Mesh twisting of up to 0.001 radians
- 5 inners and 1 outer, to force an identical iteration count

The time spent in the assemble/solve routine is shown in Table II. It is clear that higher order methods are more expensive in terms of runtime than lower order methods: they simply require more work. For each element order however, there are clear differences between the Gaussian elimination implementation and MKL's `dgesv`. For orders up to three where the matrix size is 64×64, MKL does not provide a performance advantage with the handwritten solve providing 1.2X–2.0X better performance. For larger matrices MKL shows an improvement of 1.7X over the handwritten solve for the 125×125 matrix. The fourth order matrix is 125 KB in size, which is much larger than the 32 KB L1 data cache on the Skylake architecture, but should remain resident in the L2 cache. Additionally, the storage space for the matrix is reused for each matrix construction (as we have run the code flat MPI) wherefore the reuse of this array is high and so is unlikely to be evicted from cache. However, as it is larger than L1 cache the solution of the matrix is likely to be improved by the cache-blocking optimisations typically found in linear algebra libraries to improve the performance.

The matrix sizes would generally be classed as 'small', at least in comparison to the large dense matrices typically solved with LAPACK routines. Many applications perform linear algebra on small dense matrices as in this application. Small matrix libraries exist which allow computation on individual small matrices, however they only provide at most Level 3 BLAS functionality and so it is not possible to use them directly without re-implementing the linear solve routine. Solution of a linear system is typically performed using LU factorisation and uses the BLAS routines internally. The PLASMA library implementation of LAPACK routines is also focused on node-level parallelism of the solve itself and so is not applicable where the code is run in a flat MPI style or when threads are programmed in other problem dimensions to take advantage of algorithmic concurrency in the problem dimensions [15]. Similarly, batched routines are provided which can operate on multiple matrices in parallel. For our purposes, the linear systems are constructed and solved on-the-fly and so it is not possible to apply this batched library option. Additionally, these experiments were performed under

TABLE II: Assemble/solve time in seconds on Skylake processors for different finite element orders

Order	GE	% in solve	MKL	% in solve
1	4.29	34%	8.12	55%
2	31.99	54%	64.92	70%
3	205.02	74%	254.20	73%
4	1426.98	87%	859.26	74%

a flat MPI regime, meaning that any batched routine would simply process each matrix in turn, and so cannot provide an advantage. We evaluated the parallel scheme in Section IV-A: a batched routine could be used for the concurrent work under the best parallel scheme there, however all the matrices would need to be constructed in advance of the batched routine which would result in not insignificant storage overheads.

1) Relative cost of the solve: It is important to also consider the relative costs of the matrix solve routine compared to the cost to assemble the linear system in the first place. Constructing the matrix from the precomputed integrals of basis function pairs requires reading these values from memory and thus decreases the computational intensity as the FLOP count is not similarly increased.

Timers were added to record the cost of the solve itself. Adding the time recording increases the runtime due to the overhead of repeatedly calling the timing routines for every matrix solve. For linear finite elements, only 34% of the runtime is attributed to the solution of the linear system, indicating that it is the assembly itself which is the expensive part. In contrast, at higher orders, over 70% of the runtime is in the solution of the system. This balance is somewhat expected: the assembly of the matrix requires $O(N^2)$ operations whereas the linear system solution requires $0.67N^3$ operations for a $N \times N$ matrix. For high finite elements orders therefore the asymptotic behaviour of the solve will dominate the assembly. However, for the commonly used low finite element orders it is important not to consider the matrix solution as a dominating performance cost where the assembly of the system itself is critical to performance. The characterisation or modelling of the performance limiting factors will differ depending on the finite element order increasing the challenge in this space.

This therefore may lead to alternative optimisation approaches for low and high order elements. For low order elements it may be attractive to pre-assemble (and invert) the matrix as it is invariant across the outer and inner iteration loops. This will clearly increase the memory footprint of the application as a matrix must be stored for each angle-group-element (for linear elements this is a factor of 8 times the already large angular flux array). It is the subject of future work to explore the total time to solution for pre-assembling the matrices and simply reading them from memory to compute the solve in comparison to the current implementation of assembling them on the fly. However this optimisation may be more limited in its effectiveness for higher orders.

V. Conclusion

The SNAP mini-app was extended to explore the performance of solving the transport equation on unstructured meshes using the discontinuous Galerkin finite element method. The unstructured meshes are formed by first constructing the original SNAP mesh and performing a twist. The code, nicknamed UnSNAP, supports arbitrarily high order elements for flexibility in understanding the performance characteristics of this method based on element order.

A sweep schedule suitable for advanced architectures was introduced. Unlike the current state-of-the-art, these schemes focus on enabling on-node parallelism. To this end, different concurrency schemes for following the schedule on each node were investigated. This showed that lots of parallel work is required to enable good performance and that the unstructured access to mesh data can be mitigated by choosing appropriate data layouts to increase the size of the contiguous data.

The finite element method requires an assembly and solution of a small, dense linear system. The performance of the linear solve using the Intel Math Kernel Library was compared to a hand-written vectorised Gaussian Elimination routine. For low order elements, the hand-written solve provided the best performance although the majority of the runtime was associated with the system assembly rather than the solution. For high order elements, the solution of the system dominated the performance and so for these larger matrices the Intel Math Kernel Library offered performance improvements.

Solving the transport equation on unstructured meshes has many challenges and so in future it is hoped that UnSNAP can enable research into how the benefits of other advanced architectures (such as GPUs) can be successfully leveraged.

Acknowledgment

Access to the Cray XC40 supercomputer, Swan, was kindly provided though the Cray Inc. Marketing Partner Network.

References

[1] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications," pp. 330–346, 2000.

[2] R. J. Zerr and R. S. Baker, "SNAP: SN (Discrete Ordinates) Application Proxy - Proxy Description," LA-UR-13-21070, Los Alamos National Labratory, Tech. Rep., 2013.

[3] T. Deakin, "Leveraging many-core technology for deterministic neutral particle transport at extreme scale," Ph.D. dissertation, University of Bristol, 2018.

[4] T. Deakin, S. McIntosh-Smith, M. Martineau, and W. Gaudin, "An improved parallelism scheme for deterministic discrete ordinates transport," *International Journal of High Performance Computing Applications*, sep 2016.

[5] T. Deakin, S. McIntosh-Smith, and W. Gaudin, *Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale*. Cham: Springer International Publishing, 2016, pp. 429–448.

[6] K. Garrett, N. Patel, and K. Procopio, "Tycho 2," LA-CC-16-049, Tech. Rep., 2016.

[7] E. Lewis and W. J. Miller, *Computational methods of neutron transport*. American Nuclear Society, 1993.

[8] S. D. Pautz, "An algorithm for parallel Sn Sweeps on Unstructured Meshes," *Nuclear Science and Engineering. and Eng*, vol. 140, pp. 111–136, 2002.

[9] Lawrence Livermore National Laboratory, "UMT," LLNL-CODE-638452, Tech. Rep., 2013.

[10] K. Koch, R. Baker, and R. Alcouffe, "Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine," *Transactions of the American Nuclear Society*, vol. 65, pp. 198–199, 1992.

[11] S. D. Pautz and T. S. Bailey, "Parallel Deterministic Transport Sweeps of Structured and Unstructured Meshes with Overloaded Mesh Decompositions," *Nuclear Science and Engineering*, vol. 185, no. 1, pp. 1–17, jan 2017.

[12] L. L. N. Laboratory. (2012) Acs sequoia. [Online]. Available: https://asc.llnl.gov/computing_resources/sequoia/index.html

[13] S. Pautz, "An Algorithm for Parallel Sn sweeps on Unstructured Meshes," Tech. Rep., 2000.

[14] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - A Massively Parallel Transport Mini-app," in *Joint International Conference on Mathematics and Computation, Supercomputing in Nuclear Applications and the Monte Carlo Method*, no. ANS MC2015. Nashville, Tennessee: American Nuclear Society, 2015, pp. 1–13.

[15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38 – 53, 2009.