Cache-efficient algorithms for finite element methods on arbitrary discretizations

Snel, H.D.

Award date:
2015

Department of Mathematics and Computer Science
Algorithms Research Group

2IM91 – Master's Thesis

Cache-efficient algorithms for finite element methods on arbitrary discretizations

Hugo Snel, BSc.

Supervisor:
dr. Herman Haverkort

December 14, 2015

Abstract

In the field of numerical simulation the finite element method is one of the most popular general purpose techniques for computing accurate solutions. Since memory latency is one of the most serious bottlenecks in high-performance computing, I/O-efficient algorithms which minimize this latency have been developed for finite element methods defined on grid-based discretizations by ordering the data accesses along a space-filling curve. In this thesis we investigate the design of I/O-efficient algorithms for finite element methods for arbitrary discretizations in two and three dimensions. In 2D we are successful in extending the 2D algorithm to arbitrary subdivisions at the cost of doubling the traversal length. In 3D an optimal solution was not achieved, but a highly efficient solution which seems to scale well with the input size is still obtained by using heuristic approaches. Experimental evaluation shows that for tetrahedral subdivisions of point sets up to $80,000$ the cache-miss rate can be reduced to as low as 5 %.

Contents

1 Introduction

In Scientific Computing the key objective is to do virtual experiments on a computer regarding problems in the physical sciences. By capturing natural phenomena in mathematical models (usually based on partial differential equations (PDEs)), numerical simulation techniques may approximate real-life behavior. Examples include how heat distributions or fluid dynamics change (or settle) over time. In this field the finite element method is one of the most popular general purpose techniques for computing accurate solutions [1]. Just like all other methods it relies on the availability of a discretization of the input domain.

Although numerical simulation techniques are a heavily studied subject due to their relevance in many fields of science, according to Schweitzer[2] relatively little attention is paid to the practical details of programming these algorithms. By taking the hierarchical memory design of a computer into account during the design stages of an algorithm data input and output operations (I/O-operations) can become much faster and less frequent, consequently avoiding the associated memory latency and decreasing the running time of the algorithm. In modern computers this memory latency is one of the most serious bottlenecks in high-performance computing [3]. The external-memory model of computation by Aggarwal et al.[4] analyses an algorithm in terms of its I/O-efficiency, i.e. the number of I/O-operations as a function of the input size. A distinction is made between cache-aware algorithms, which require knowledge of the hardware parameters such as the block size and memory size, and cache-oblivious algorithms, which do not. For finite element methods defined on two- and three-dimensional grid-based discretizations, Gunther et al. [3] developed the first cache-oblivious algorithm using the Peano space-filling curve in 2006 (later dubbed the Stack & Stream algorithm [5]).

Since then this approach has been generalized to other grid-based discretizations – using other space-filling curves – but never to arbitrary discretizations [5, Chapter 14]. In this thesis we investigate the design of cache-efficient algorithms for finite element methods for arbitrary discretizations in 2D and 3D. We develop an abstract data access model which expresses efficient data access as a minimization problem and show that previous work on grid-based discretizations effectively found an optimal solution in this model. In 2D we are successful in finding an optimal solution for arbitrary discretizations. In 3D an optimal solution was not achieved, but a highly efficient solution which scales well with input size is still obtained by using heuristic approaches.

The remainder of this thesis is structured as follows: Section 2 describes the finite element method computation and the external-memory model of computation. In Section 3 we explain the Stack & Stream algorithm, after which our generalization to arbitrary 2D discretizations is given in Section 4. Section 5 discusses the generalization step to 3D, presents the abstract data access model, and discusses (heuristics for) two approaches which minimize the number of I/O's. We have implemented both approaches, the experimental results of which are given in Section 6. Finally, we list areas of interest for future work in Section 7 and conclude in Section 8.

2 Model of computation

In this section we explain the context in which we have developed our algorithm. First, we will talk about the kind of data used in finite element method type calculations and how it is used. Secondly we introduce the external-memory computational model and how it differs from the standard RAM model. Lastly, we look at how the external-memory model model differs when applied at the cache level of the memory hierarchy.

2.1 Finite element methods

In the field of numerical simulation the finite element method (FEM) is an approximation scheme which by iterative computations eventually converges towards the solution of the simulation problem. Examples include how heat distributions or fluid dynamics change (or settle) over time. There exist many different kinds of finite element methods and problems, each one slightly different based on their application. We will define a typical input first and discuss variants afterwards.

The finite element method is defined on a discretized input space and uses time steps to advance the simulation. The accuracy of the model depends on the granularity of the discretizations and the time steps. The discretization results in a subdivision which can be characterised by a set of vertices $V = (v_1, \ldots, v_n)$, a set of non-intersecting edges $E \subseteq (V \times V)$, a set of non-intersecting elements or cells $C = (c_1, \ldots, c_m)$ and, in three and higher dimensions, a set of 2D faces F. Although the finite element method can be defined for any subdivision, we restrict ourselves to triangulations in 2D or tetrahedralizations in 3D unless explicitly stated otherwise, i.e. subdivisions where each cell is a triangle in 2D or tetrahedron in 3D respectively. Input sizes can range into the millions of cells, depending on the accuracy required.

All vertices $v \in V$ have a variable(s) associated with them (also referred to as an 'unknown' in literature). Examples are a vector field with velocity in d dimensions and pressure, or variables from Maxwell equations and temperature. During an iteration the values of these variables are updated to their new value for the next iteration. The new value of a variable is calculated from weighted linear independent contributions of its current value, and the variable values of those vertices in the subdivision which it shares a cell with [1, Section 4.1]. Due to the independent nature of the contributions, we can define the computation on a per-cell basis, which is known as an element-oriented approach. Consider a cell $c \in C$ of vertex degree k. Let vector $\mathbf{v_c} = [v_1, \ldots, v_k]^T$ hold the variable values associated with its k vertices, and let A_c be a $k \times k$ matrix. Then all vertex-to-vertex contributions at c of a single iteration can then be calculated by $\mathbf{v'_c} = A_c\mathbf{v_c}$ after the values of A_c (i.e. the weighting of the contributions) are properly defined [1, Section 6.1.1]. The total computational work of a single iteration is the summation of the computational work done at each cell, i.e. the computation of $A_c\mathbf{v_c}$ for all $c \in C$.

The matrix A_c is known as the reference matrix, cell matrix or element (stiffness) matrix and its values depend on the shape and orientation of c in the subdivision. If the subdivision is completely uniform (e.g. a uniform grid) then A_c is identical for each cell, but if it is completely irregular it can be unique for each cell. We will assume that all initial values (i.e. vertex variables and cell matrices) are given as an input and abstract from concrete problem-specific values.

For each variable at a vertex an associated accumulating variable (also known as 'residual') is also stored. The role of the accumulating variable is to sum up the contributions from $\mathbf{v_c}'$ at each adjacent cell c of the vertex. At the end of an iteration the resulting value of the accumulating variable replaces the current variable value. The number of such variable and accumulating variable pairs depends on the problem at hand. In the remainder of this thesis we refer to all vertex variables as 'the vertex'. Idem for cell variables.

Depending on the application the finite element method might be different from this descrip-

tion. Differences include models which include variables on edges, hanging vertices (i.e. vertices which lie in the interior of an edge), or calculations that depend on the computation order of the cells. For some applications may also be desirable to dynamically change the subdivision during the computation. Points may be added or removed based on some criteria which we abstract from, and subsequently the subdivision has to be re-triangulated. The algorithm needs to be able to adopt these changes in between (a couple of) iterations. We do not consider these variations unless explicitly stated otherwise.

2.2 The external-memory model of computation

For algorithm analysis the standard computational model, also known as the RAM model, assigns a unit cost to each operation (for example a load, store, or computational operation). Since we are interested in asymptotic worst-case behaviour we abstract from any constant factors in the algorithm and express the complexity in terms of its input or parameter size. This leads to the familiar big-O notation, e.g. $O(n \log n)$ and $O(kn)$ for sorting n numbers consisting of at most k digits with merge sort and radix sort respectively. The assumption of a unit cost per operation is one that is justified in practice for most algorithms and has shown to result in a model that is often a good predictor of the running time. However, when the input size of a problem exceeds the main memory of the machine, the performance bottleneck of an algorithm often shifts from computational complexity to I/O-complexity due to the high latency associated with an I/O-operation (input/output from/to disk). This is where the external-memory model of computation [4] becomes a far more accurate running time predictor.

The standard external-memory model is based on a two-level memory hierarchy. The main memory or internal memory is small and fast, but cannot hold all data at once, whereas the external memory (usually a magnetic disk) is slow and viewed as being infinitely large. Computations can only be done on data that is present in the internal memory. Whenever data is necessary that is not present, a miss is said to occur and an I/O-operation is executed to retrieve the data from external memory, replacing a block of internal memory data that is consequently moved to external memory to make room. When necessary we may assume that we have complete control over the replacement policy, i.e. the choice which block of data to move to disk.

In the external memory model it is assumed that the latency incurred from I/O-operations throughout the algorithm dominates the running time and that any computational operations done in internal memory are negligible. Therefore the complexity analysis of an algorithm in the external memory model uses the I/O-operation as the unit cost. It is expressed in terms of the hardware specifications of the machine that will run the algorithm.

On all machines both the internal memory and external memory are partitioned into blocks and whenever a data element has to be read or written the I/O-operation reads or writes the whole block it resides in – blocks are the only unit for I/O-operations. The block operations are used to amortize the high latency costs associated with it over the useful data elements loaded. This is a form of pre-fetching data, banking on correlated data accesses (e.g. accessing sequential elements in an array). Given the size of a data element (e.g. a variable) we can calculate the number of contiguous data elements that are read or written at once during an I/O-operation. This number is denoted by B, and analogously we use M to denote the number of data elements that fit into internal memory. A usual assumption is that $M > B^2$.

Algorithms which try to minimize the number of I/O-operations sometimes require the values of B and/or M as an input parameter. These algorithm are called *cache-aware*. When I/O-efficiency is achievable without knowledge of B and M the algorithm is called *cache-oblivious*.

In the external memory model there are two main principles that decrease the number of I/O's and increase the efficiency of an algorithm. If we ensure that data elements which are stored close to each other on disk are needed around the same time we cater to the assumption underlying block I/O-operations and thus increase their efficiency. The extra data that was pre-fetched will

still reside in internal memory when needed and will not require its own I/O-operation. This is known as the *spatial locality principle*. Secondly, when the moments of access to the same data element are clustered in time, it is likely to still be present in the internal memory and we do not need multiple I/O's to fetch the same data. This is the *temporal locality principle*. Adhering to these principles is the key to make the most of each I/O-operation and reduce the I/O-complexity of an algorithm.

2.3 Modelling cache-efficiency

The cache-efficient model is very similar to the external-memory model described in Section 2.2, but it is applied at a different level of the memory hierarchy. We again have a two-level hierarchical memory model, but now the small on-chip cache memory is the internal memory and the main memory is regarded the external memory. The internal and external memories still communicate using block I/O-operations [1] and the spatial and temporal locality properties still apply. However, there are also some differences.

1. The latency associated with an I/O-operation between main memory and cache memory is smaller than it for I/O's between disk memory and main memory. Thus it is less dominant for the running time and disregarding all internal memory operations completely is not justified.

2. Usually there are several levels of cache memory, each one a bit bigger and slower than the previous, to bridge the I/O-latency disparity between the fastest cache and main memory. Usually we have L1 and L2 cache, but L3 caches also exist.

3. In the external memory model we had assumed that we had complete control over the replacement policy and blocks, but the replacement policy at cache level is implemented in hardware and rarely has control instructions. Furthermore, this implementation is usually also n-way associative instead of fully associative. Each block of main memory data can go to one of n slots in cache-memory (usually $n \in \{1, 2, 4, 8\}$). Thus we cannot write a block of data to our memory location of choice.

We could try to incorporate all differences into a new model, but the reality is that there is a trade-off between simplicity and accuracy. The more details you incorporate the more accurate your model will be, but it will also become increasingly harder to take all of them into account when designing or analysing your algorithm.

There is some work dedicated to these differences. Sen et al. [6] present a model that maps any I/O-efficient algorithm at disk level to a cache efficient version of it and bounds the resulting number of I/O-operations by a constant while respecting the restrictions of the replacement policy and associativity. Furthermore, it also incorporates the internal memory operations into the complexity analysis.

Frigo et al. [7] do not address the differences, but instead argue why they are minor. They make the ideal-cache assumption which assumes that the differences listed in bullets 2. and 3. do not affect the asymptotic analysis when applying the external memory model at the cache level, and additionally assume that the replacement policy is optimal. These are strong assumptions, but they theoretically justify each of them. The main justification comes from the work of Sleator et al. [8], who tell us that any machine implementing a least recently used (LRU) replacement policy has at most twice as many misses as an optimal machine with half the memory, thus bounding the miss rate by a constant.

We note that in the papers by Sen and Frigo the framework of internal and external memories with block data transfers as well as the spatial and temporal locality design principles remain

[1] In literature a block of cache memory is often referred to as a cache line.

intact. The differences that exist have a minor impact or can be dealt with by using the mapping strategy of Sen et al. In the remainder of this thesis we therefore work on the premise that the analysis and assumptions of the external-memory model are also applicable and accurate at the cache level of the memory hierarchy and will use the terms cache-efficient and I/O-efficient interchangeably. We will introduce a few more in-depth hardware concepts in Section 6.5, where we discuss the results on this thesis in the context of typical hardware specifications.

3 Stack & Stream traversals

The Stack & Stream algorithm is the original work this thesis builds on. It is a cache-oblivious algorithm for the types of calculations described in Section 2.1 which allows for adaptive refinement of the mesh while retaining its I/O-efficiency. In this section we will explain the main idea behind the algorithm. We refer the interested reader to the book of Bader [5, Chap. 14] where a full description, including implementation details, is given. The original work was published by Gunther et al. [3].

To minimize the number of I/O-operations the goal will be to only use stack data structures. Stacks only have push and pop operations which access only one address in the memory and subsequently move the head pointer to the new head or previous element. Therefore all additional data elements loaded by the block I/O-operations are those next in line to be popped, i.e. stacks have perfect spatial locality when implemented in contiguous memory. Furthermore, the elements which have not been recently used will also not be used in the near future because they are buried deep in the stack. The cache's least recently used replacement policy therefore matches our intended use of the data. We will now discuss the order in which the calculation should be executed to allow for stack data structures.

3.1 Traversal order

In the Stack & Stream algorithm the element access order is determined by the order in which a space-filling curve running through the cells of the subdivision visits them. We will first talk about the structure of a single traversal (i.e. FEM iteration) and later show how it easily extends to many iterations.

A space filling curve is a curve running through a refinement pattern that can be applied recursively and which, when refined ad infinitum, occupies the whole domain of unit space it is defined on. An example of this is the Hilbert curve illustrated in Figure 1.

The refinement pattern of the Hilbert curve is a square that has been bisected on both axes to obtain four subsquares. If any square would be recursively refined then a (mirrored or rotated variant of the) refinement pattern determines the course of the space-filling curve through the new squares such that its endpoints connect to the predecessor and successor of the original square. The orientation in which the refinement pattern is recursively applied unique in the case of the 2D Hilbert curve, but for the 3D variant there are multiple orientations [5, Chap. 8]. Many space-filling curves exist each with their own refinement pattern and orientations. Examples include the Peano curve whose refinement pattern is defined on a square that is divided into nine subsquares by trisecting both axis, and the Sierpinski curve which is defined on a triangular space that is bisected along the hypotenuse. For now we will work with the Hilbert curve example.

Let the finite element method subdivision be a quadtree grid, i.e. a subdivision that can be obtained by recursively splitting a square into four equal quadrants, subquadrants, and so on. This grid does not need to be uniform – some quadrants may be refined more than others. When we draw the Hilbert curve through the grid we obtain an order in which the cells and vertices are visited (illustrated in Figure **??**). Whenever two vertices are access at the same time (i.e. the same cell) ties are resolved by the order in which they would have been accessed if that cell was refined once more. Furthermore we categorize all vertices as lying either to the left (green) or right (red) hand side of the curve.

The order of the finite element computation matches the order in which the space-filling curve visits the cells. At each cell all its adjacent vertices are loaded to make the matrix-vector computation. In the next section we will show how the properties of this order and categorization can be used to set up an efficient stack data structure.

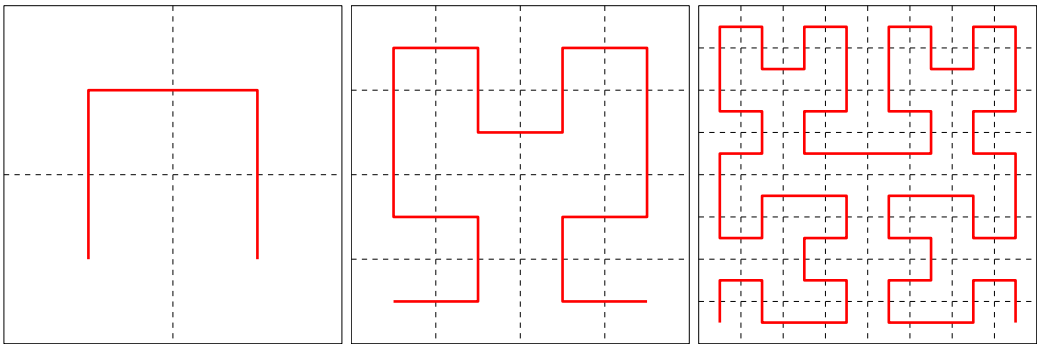


Figure 1: The first three iterations of the 2D Hilbert curve.

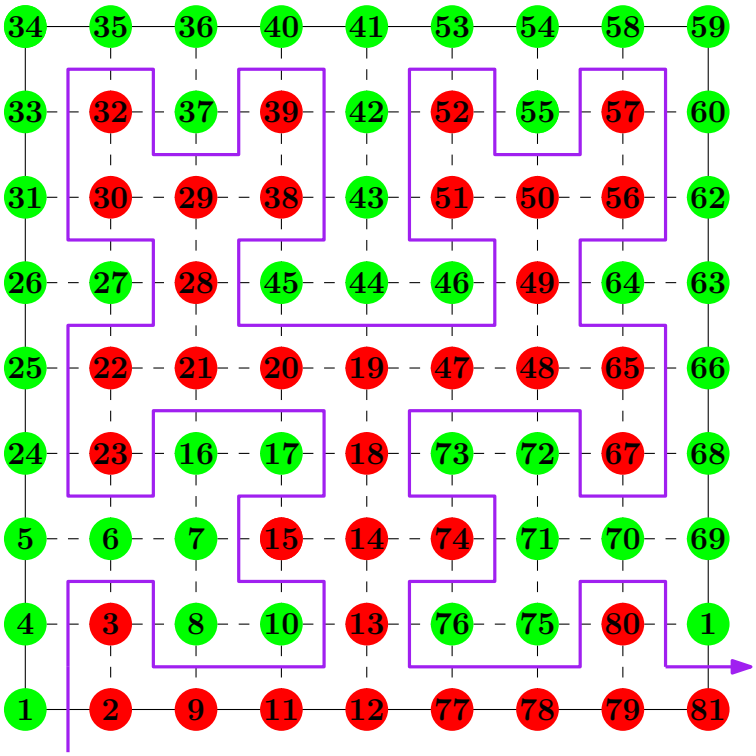


Figure 2: The enumerated access order of vertices in an uniform quadtree grid.

3.2 Stack structure

We are now ready to define the data structure behind the element accesses. We need to provide four I/O-efficient operations: loading and storing the data associated with cells (cell matrix), and loading and storing the data associated with vertices ((accumulating) variables). We work with the worst-case assumption that all cell matrices are unique and need to be loaded for each cell separately.

The cell data structure has a near-trivial design since all cells are only used once. We can order the cells on their access time and store them sequentially in contiguous memory for perfectly I/O-efficient operations. However, if we would use an array implementation refining a cell of the grid (i.e. splitting it into four equal subquadrants) would take $O(n)$ time to insert the subquadrants into the array if we want to maintain the correct order. The solution is to instead

use a stream implemented by two stacks: an input and an output stack. During the traversal cells are continuously popped from the input stack and pushed to the output stack. Whenever a cell needs to be refined we pop it from the input stack and, instead of the cell, push the four new subquadrants to the output stack in the order in which the refined space-filling curve visits them. Conversely, for a coarsening operation we can load the four quadrants that descended from the same parent consecutively from the input stack and replace them by a single cell on the output stack.

The vertex data structure is a bit more sophisticated. We again have an input and output stack which are ordered along the space-filling curve, but because vertices are accesses more than once a simple sequential traversal is not possible without duplicating vertices. We need a structure to store the vertices at in between their first a last access – the intermediate vertex stack. Whenever a vertex is accessed for the last time we move it to the output stack, otherwise it is stored on the intermediate stack. This strategy is possible because the Hilbert curve (and all other 2D edge-connected space-filing curves, i.e., curves whose subsequent subdomains at least share a common edge) satisfy the *stack property*: all accesses to the vertices of the same color in Figure 1 (i.e. red or green) occur in a strictly most recently used scheme, i.e. follow a stack principle. Thus if we employ an intermediate stack for both the left and right hand side all vertex load and store operations are supported by stack data structures.

The access scheme of a vertex is now as follows: At the first access it is popped from the input stack and pushed to the intermediate stack. At intermediate accesses it is both popped from, and pushed to, the intermediate stack. Finally, at the last access it is popped from the intermediate stack and pushed to the output stack.

Analogue to the cell refinement and coarsening, we can again replace vertices as needed in an on-line fashion.

So far we have shown how we can efficiently support vertex load and store operations by cleverly distributing them over the two intermediate stacks, but we have not addressed how to distinguish between them and the input/output stacks during a load or store operation. It turns out that the refinement pattern of the space-filling curve is structured enough to locally determine whether an adjacent vertex is accessed for the first or last time. We refer to Bader [5, Section 14.2] for a full discussion on how to maintain the curve's local orientation during the traversal.

3.3 Iterations

Now that we have defined the structure of a single traversal extending it to multiple ones is surprisingly simple. Observe that during the first forward iteration all elements of the input stacks have been moved to their respective output stack in a last-access order. Thus the elements currently residing on the output stacks exactly match the first-access order if we now do a back-to-front traversal along the space-filling curve. As a consequence many consecutive iterations can efficiently be executed by inverting the interpretation of the input and output stacks, and the traversal direction along the space-filling curve at the end of each iteration.

4 Generalizing 2D Stack & Stream to arbitrary subdivisions

In Section 3 we have seen how the space-filling curve based Stack & Stream approach provides an I/O-efficient traversal of the subdivision. However, the power of the space-filling curve approach is not immediately apparent. In an uniform quadtree grid a row-by-row traversal would also satisfy the stack and inversion properties. In fact, in 2D any Hamiltonian path in the dual with end points adjacent to the hull of the subdivision would suffice to define the left/right hand classification and stack structure, but these approaches do not support refinement and coarsening. In this context a space-filling curve can be viewed as a persistent Hamiltonian path under local refinement/coarsening operations. However, using a space-filling curve approach is only possible when the subdivision matches the refinement pattern of the space-filling curve, even for the static variant of the problem. In this section we will present a modified approach which works for arbitrary connected planar subdivisions.

4.1 Traversal order

Now that we are working with arbitrary planar subdivisions space-filling curves are not applicable any more and we need a different curve to make the left/right division in order to use the Stack & Stream algorithm. Any Hamiltonian path across the cells (i.e. the dual of the subdivision) would do, but its end points must be adjacent to the hull of the subdivision otherwise the left and right hand side may overlap (e.g. a spiral after the first 360 degree turn). However, not every subdivision has a Hamiltonian path and finding one is a hard problem to begin with (it is one of Karp's original 21 NP-complete problems [9]). So how can we deal with this?

The solution is simple – we do not. Although a Hamiltonian path would work it is not a necessary condition for the Stack & Stream algorithm. It is only required that the traversal has the stack property and visits every cell and vertex at least once, but it is ok to visit a cell more than once.

A spanning tree defined on the dual of the subdivision touches every cell. Therefore a treewalk along this spanning tree defines a traversal which visits every cell at least once. At each cell we make an anti-clockwise boundary walk, interrupted by recursive boundary walks along the children. We therefore pass every vertex from each of its adjacent cells. This curve is a 2-approximation in terms of the total number of cells visited compared to the Hamiltonian path. Both the cell and vertex access orders are determined by the order in which they are lie incident to the right hand side of the curve during a traversal of it. This also defines the order in which the computations are done. The concept is visualised in Figure 3, and an example of the order from a cell's local point of view is illustrated in Figure 4 (left).

The left-right division of the vertices along the treewalk curve is special because the the left hand side is a void and does not contain any vertices. Nonetheless we will try to employ the same system of stacks as used by the regular Stack & Stream algorithm in Section 3. In the remainder of this section references to 'the curve' and 'the access order' always implicitly refer to the right hand side of the treewalk curve and access order.

It is not straightforward that this approach is immediately applicable. There are still a number of issues that we need to address: (1) Does the treewalk curve commit to the stack-property? (2) What data structure do we use for the intermediate cell storage now that they are accessed multiple times? (3) How do we pass on the vertex-to-vertex contributions now that they are not accessed consecutively at a cell? (4) How do we discriminate between popping from the input or the intermediate stack, and pushing to the intermediate or output stack? (5) Is it still possible to do iterative traversals in the reverse direction? We will address issues (1) and (2) in Section 4.2, and issues (3), (4) and (5) in Sections 4.3, 4.4 and 4.5 respectively.

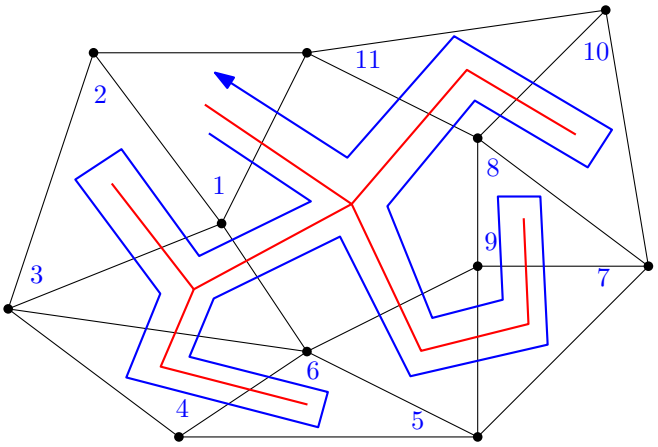


Figure 3: The treewalk curve of a spanning tree defines a traversal through the subdivision. The vertices are order according to their first appearance at the right hand side of the curve.

4.2 Treewalks and stack property

Perhaps the most important of these questions is the stack property. Without it a stack-based traversal as presented in Section 3 is simply not possible. We will now formally prove that the treewalk order commits to the stack property for both the cells and vertices, i.e. that the intermediate data structures can be implemented with a stack.

Theorem 1. *The cell access order of the treewalk commits to the stack property.*

Proof. The treewalk order is a depth-first traversal of the nodes (i.e. cells) of the tree. A cell c stored on the intermediate stack can only be blocked on the intermediate stack by cells that come after it, but due to the depth-first order the last accesses of all cells that come after c occur before the next access of c. Thus they will have been moved to the output stack and c lies on top of the intermediate cell stack. □

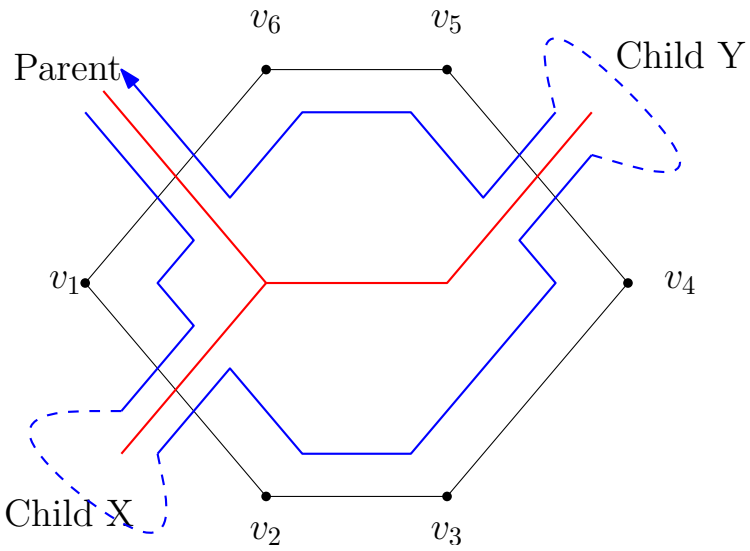
However, this means we load the full cell matrix upon each cell visit even though we only need column i of A_c to calculate all contributions of $\mathbf{v_i}$. We therefore chop up A_c into its columns and order them in contiguous memory according to the order in which we need them to calculate the contributions. The resulting order can then, once again, be streamed. The stack property and structure is still required for passing on the vertex-to-vertex contribution and discriminating between stacks though, as we will discuss in Section 4.3.

Theorem 2. *The vertex access order of the spanning tree treewalk commits to the stack property.*

Proof. If a vertex v is only accessed once (i.e. adjacent to only one cell) it is directly stored on the output stack and does not pose a problem. Now consider the case where v is accessed multiple times. We will prove that v can always be found at the top of the intermediate vertex stack for all accesses except for the first, where it is found on top of the input stack instead. The vertex labelled '7' in Figure 3 may serve as a visual aid during this proof.

We enumerate all vertex accesses along the treewalk curve to obtain an order of the points in time at which vertices are accesses. Let t and $t+k$ be two consecutive access times to vertex v in this order, and p the subcurve of the treewalk curve that starts at t and ends at $t+k$. At each cell the treewalk curve makes a boundary walk incident to the cell's vertices interrupted by recursive boundary walks along its children. Thus subcurve p starts and ends at different cells, but its start and end point lie incident to the same vertex.

If we connect the start and end point of p at v then we form a cycle which encloses a part of the subdivision. All vertices which lie on the inside (right hand side) of this cycle are accesses along p after t and can potentially block access to v on the intermediate vertex stack at time $t+k$.



Parent
v_1
Child X
v_2
v_3
v_4
Child Y
v_5
v_6
Parent

Figure 4: An example of the traversal order from a cell's local point of view (left) and the matching order of the vertex input stack (right). "Parent", "Child X" and "Child Y" denote the vertices visited by the curve at other cells, not the actual cells.

The treewalk curve is non-intersecting and, because the cycle is formed by connecting p at a vertex, no other part of the curve can reach the area enclosed by p since it is defined on the cells (dual) of the subdivision. Therefore all accesses to the enclosed vertices occur along p. This includes the last access at which they are moved to the output stack. Hence all vertices that come in between two successive accesses of v will have been moved to the output stack and at time $t+k$ vertex v can be found on top of the intermediate vertex stack. □

4.3 Passing on contributions

Recall that at each cell c we need to compute $\mathbf{v'_c} = A_c\mathbf{v_c}$ where $\mathbf{v_c}$ is a vector holding the variable values associated with each vertex adjacent to c, A is the cell matrix which defines the weighting of the vertex-to-vertex contributions, and vector $\mathbf{v'_c}$ accumulates the vertex-to-vertex contributions and holds the new variable values at the end of the iteration.

Say c is a triangle, then $\mathbf{v'_i} = A_{i,1} \cdot v_1 + A_{i,2} \cdot v_2 + A_{i,3} \cdot v_3$. In the original Stack & Stream algorithm a cell's adjacent vertices are all treated consecutively, thus for $i = 1$ it is straightforward to add the contributions of $A_{1,2} \cdot v_2$ and $A_{1,3} \cdot v_3$ to $\mathbf{v'_1}$. But now that we have separated the calculation in time we need to address how to pass on these contributions.

We have proven in Section 4.2 it is possible to implement the cell accesses with an intermediate stack. In the end we choose not to use it because it introduced overhead in the form of loading unnecessary data, but it can still be used as an efficient data structure for storing data that is shared between vertices of the same cell. Thus we can pass on a contribution from one vertex to another by storing it to the cell intermediate stack, but because we can only pass on information in the direction of the traversal at the end of the current iteration not all variable values of $\mathbf{v'_c}$ will be up to date. However, a vertex can only be missing contributions from vertices that come after it. Therefore, since we inverse the traversal direction after each iteration, all stored contributions will still make it to their destination vertex in time (i.e. before its next use).

For the original Stack & Stream algorithm we had worked with the worst-case assumption that all cell matrices are unique and need to be loaded for each cell separately. This is why we had to

design a cell data structure, even though the best or realistic case might not require it. As it turns out this effort has not been in vein – for the generalized variant it is required for every subdivision to pass on the vertex contributions.

4.4 Discriminating between stacks

Using space-filling curves it was possible to locally deduce whether a vertex should be loaded from or stored to an intermediate or input/output stack due to the regular structure, but for arbitrary subdivisions we do not have this structure and we need to store it explicitly. All additional information can be stored on the cell stack. We augment the data associated with a cell of d vertices with two bit-vectors and one integer.

Since we have only one intermediate stack one bit of information is enough to discriminate between which stack to load from or store to respectively. Thus we store a length d vector of bit-pairs $(l, s)_i$ which each cell, where $l, s = 1$ if we need to load from or store to, the intermediate stack and 0 otherwise. Additionally we need to know whether we should recurse to a child or not at each edge during the anti-clockwise traversal of the cell (excluding the edge adjacent to the parent cell). For this we need another bit-vector of length $d - 1$. Finally, to be able to continue the boundary walk where we left off after returning from a child we need a single integer in the range $[1, d - 1]$, i.e. another $\log(d - 2)$ bits.

From Euler's formula it follows that vertices in planar graphs have an average degree of 6 so the total additional data is linear with a very small constant. In the case of a triangulation, where the cell size is constant, the overhead amounts to 3*2+2+1 = 9 bits per cell [2].

4.5 Iterations

The first-access order of vertices and cells during the backwards traversal should again match the last-access order during the forward traversal. This is indeed the case. Thus consecutive iterations can still efficiently be executed by inverting the interpretation of the input and output stacks, and the traversal direction along the treewalk curve at the end of each iteration.

Since the direction of the boundary walk at each cell has been inverted the extra data stored at each cell has to be interpreted backwards as well. Of the bitpair (l, s) load bit l now denotes where to store to, and s where to load from. The recursion bitvector should also be read in end-to-start order. Note that all changes are conceptual and require no changes to the data.

Finally we note that the reasoning in the proofs of Theorem 2 and 1 also applies to the backwards traversal order.

4.6 Incorporating edge variables

In the description of the finite element method model we noted that there also exist variants with variables on edges. So far we have not looked at edges, but we will now show how they can be incorporated without changing the data structure.

Edges are a new element of the subdivision which, similar to vertices, have to be accessed at each adjacent cell to make a computation. We make a distinction between two types of edges: Those which are intersected by the spanning tree, and those which are not.

We observe that edges which are not intersected by the spanning tree always connect two vertices who are accessed consecutively along the right hand side of the treewalk curve. Therefore we can insert the edge's variable(s) in between them on the vertex stack and load it as we traverse along the treewalk curve. When an edge is intersected by the spanning tree then it marks a point during the traversal where we cross from one cell to another. Hence it is accessed in between two

[2] Technically the root cell can have up to three children and form an exception. However, this can be easily avoided by choosing it adjacent to the hull of the subdivision.

cells which are accessed consecutively along the curve, and we can insert the edge's variable(s) in between them on the cell stack .

These changes do not require a structural modification of the algorithm. The information stored at the recursion bit-vector of each cell already provides all necessary information. During the boundary walk of a cell we consult the i'th bit of the recursion vector when we traverse along its i'th anti-clockwise edge.

- If the bit is equal to '1' (i.e. we recurse to the cell's child) then we know we need to access an edge from the cell stack. We load the edge from the cell input stack, calculate its contribution, push the current cell to the intermediate cell stack, load the child from the cell input stack, calculate the contribution of the edge to the child cell, push the edge to the intermediate cell stack, and finally continue our boundary walk at the child.
- If the bit is equal to '0' (i.e. we do not recurse) then we know we need to access an edge from the vertex stack. We load the edge from the same vertex stack as the last vertex, calculate its contribution, store it to the same vertex stack as the last vertex, and continue our boundary walk at the next vertex.

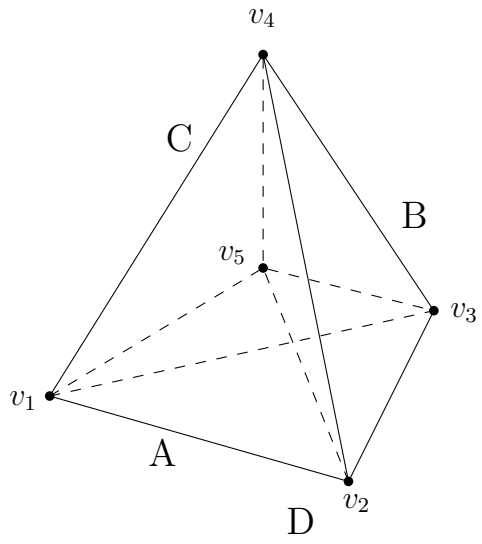
The same information is also sufficient for the second access to the edge, and can also be applied to the backwards iteration.

5 Generalizing 3D Stack & Stream to arbitrary subdivisions

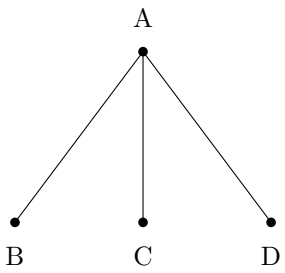
The approach in Section 4 successfully generalized the Stack & Stream approach to arbitrary connected subdivisions for the case of the static finite element method calculation. We will now look at the 3D variant of this problem. First we will show why the spanning tree approach does not generalize to 3D in Section 5.1, after which we present a model which captures the data access problem in Section 5.2. Using this model, we give two approaches for efficient intermediate data storage based on another system of stacks and a memory-constrained data structure in Sections 5.3 and 5.4 respectively.

5.1 Infeasibility of applying the 2D approach

We will now prove infeasibility of the spanning tree Stack & Stream approach for arbitrary connected subdivisions in 3D. Consider the tetrahedralization given in Figure 5a. It consists of four vertices (v_1, v_2, v_3 and v_4) which define the hull, and one interior vertex v_5 which splits up the subdivision into four cells labeled A through D which can be defined by their adjacent vertices as follows: $adj(A) = \{v_1, v_2, v_4, v_5\}, adj(B) = \{v_2, v_3, v_4, v_5\}, adj(C) = \{v_1, v_3, v_4, v_5\}$ and $adj(D) = \{v_1, v_2, v_3, v_5\}$.



(a) A simple 3D tetrahedralization.



(b) A spanning tree of the subdivision in Figure 5a.

Figure 5: A simple 3D tetrahedralization. It consists of four vertices (v_1, v_2, v_3 and v_4) which define the hull, and one interior vertex v_5 which splits up the subdivision into four cells labeled A through D.

Due to symmetry this simple 3D subdivision only has four distinct spanning tree. We will present the proof for one given in Figure 5b, but similar proofs can be constructed to proof infeasibility for the other trees (although they require a few more deductive steps). Let the spanning tree be rooted at cell A, and let cells B, C and D be its children in left-to-right order. The spanning treewalk then results in the cell sequence $A - B - A - C - A - D - A$. All vertices adjacent to a cell need to be popped from either the input or intermediate stack at some point when we visit that cell in the traversal sequence.

Consider vertex v_5 during the traversal, which lies adjacent to all four cells. Some of the cells will be visited multiple times and we can choose at which point in time we will make the computation (e.g. cell A), but a leaf of the spanning tree will only be visited once. We therefore immediately know at which point in time three out of the four accesses to v_5 occur and in between

them v_5 has to be stored somewhere. These storage time intervals correspond to the colored regions marked in Figure 6. We are purposefully leaving out the access at cell A at the moment because it complicates things and the extra access does not influence the proof.

The stack property dictates that every vertex that is put onto the intermediate stack should be the top element of the stack at the next time it has to be accessed. Therefore it is not allowed to put any other vertex on top of v_5 that will not be moved to the output stack before the next access to v_5. Conversely, all vertices underneath v_5 cannot be accessed until v_5 is moved to the output stack, so we should make sure they are not required until that point in time. The points in time at which v_5 has to be stored on the intermediate stack coincide with the colored regions in Figure 6. Therefore for each of the other vertices all accesses to it should occur within exactly one region, otherwise they will at some point block access to v_5 on the intermediate stack, or vice versa.

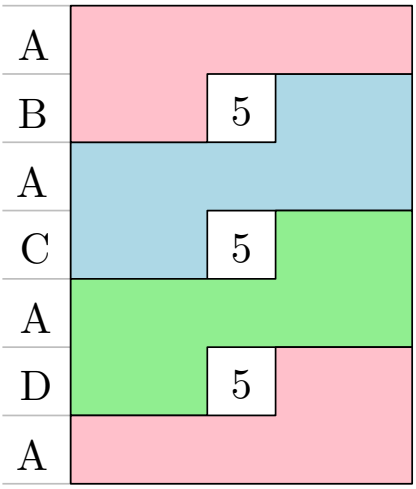


Figure 6: The storage regions defined by accesses to vertex v_5. This figure excludes accesses to cell 'A'.

The regions of Figure 6 can be defined in terms of the cells that they cover in the traversal sequence. We find that $Pink = \{A, B, D\}, Blue = \{A, B, C\}$ and $Green = \{A, C, D\}$. Recall that we can express vertices in terms of the cells at which (at some point) they need to be accessed. We can now map the access times of the vertices to the regions by finding a region which spans all the cells at which a given vertex needs to be accessed. The results are shown in Table 5.1 (right).

Region	**Covered cells**
Pink	$\{A, B, D\}$
Blue	$\{A, B, C\}$
Green	$\{A, C, D\}$

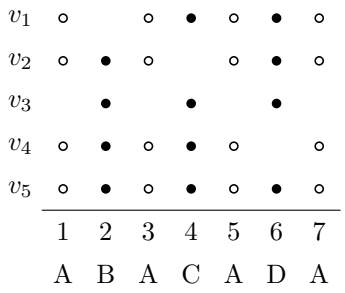
Vertices	**Maps to region**
$v_1 = \{A, C, D\}$	Green
$v_2 = \{A, B, D\}$	Pink
$v_3 = \{B, C, D\}$	-Is not contained by a single region-
$v_4 = \{A, B, C\}$	Blue

Table 1: The region problem defined by v_5 in Figure 6. The cells covered by the regions (left), and the mapping of vertices to the regions (right).

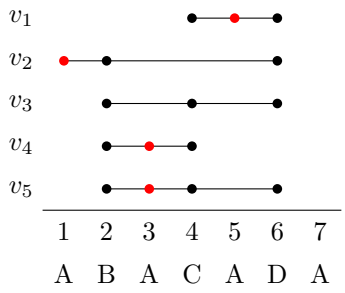
It is clear that there does not exist a region which spans all cells that lie adjacent to vertex v_3. Therefore it is impossible to find an access order that respects the stack property; at some point vertex v_3 will block access to v_4 on the intermediate stack, or vice versa. Because we have only made forced decisions (since leaf nodes are only visited once), it follows that the Stack & Stream approach with one intermediate stack is not possible for this spanning tree.

	Cells			
Vertices	**A**	**B**	**C**	**D**
v_1	✓		✓	✓
v_2	✓	✓		✓
v_3		✓	✓	✓
v_4	✓	✓	✓	
v_5	✓	✓	✓	✓

(a) The vertex-cell adjacency relations of the subdivision in Figure 5.



(b) The marked access times for each vertex along the traversal. The filled circles mark definitive access times – the vertex must be accessed here. The open circles represent a choice problem – among all of them one should be chosen as a definitive access time.



(c) An example of the resulting intervals after making arbitrary choices (marked in red) for the choice problem in Figure 7b.

Figure 7: An overview of the access model applied to the subdivision of Figure 5.

Of course adding more intermediate stacks would solve the problem. The trivial solution uses n stacks, but in this case all I/O-efficiency will be lost. In Section 5.3 we will discuss the problem of minimizing the number of intermediate vertex stacks. An alternative solution which uses a memory-constrained data structure is discussed in Section 5.4.

5.2 The abstract data access model

As we have seen in Section 5.1 the access times of a vertex during the tree traversal create regions which impose constraints on when other vertices can be accessed. If we do not respect these regions the stack property will be violated and consequently it will not be possible to implement Stack & Stream using the stack data structure. Although sufficiently clear for the counterexample regarding straightforward generalization to 3D, we did not define these "regions" and the restrictions they impose on the data accesses in full detail yet. The remainder of this section is dedicated to this problem.

If we enumerate the cell order of the traversal every region of the counterexample forms a time line on which we can mark the points in time at which a vertex is accessed. Each pair of consecutive marks now defines a time interval in between accesses during which the vertex has to be stored somewhere. We can apply this technique to each vertex and mark the time line according to the vertex-cell adjacency relations, resulting in a set of well-defined consecutive intervals in the domain $[1, 7]$ for each vertex. A vertex which lies adjacent to n cells has $n-1$ intervals associated with it. When a vertex lies adjacent to a cell which is visited multiple times a decision has to be made at which point in time the vertex access should occur. This choice problem exists for any such cell. Figure 7 gives an overview of the vertex-adjacency relations, the resulting access time options and one of the possible resulting interval sets after the choices have been made. Note that for this traversal order only cell A is visited multiple times, so only vertices adjacent to A have a choice problem (i.e. v_3 does not).

Before we can employ this access model there is one technicality we need to resolve regarding intervals which share end points. Currently the inter-cell access order has been properly defined, but the intra-cell order is ambiguous. A tetrahedralization has four vertices per cell, so the number of intervals which share an end point (i.e. are accessed at the same cell) is bound by a constant. By reserving a small amount of temporary storage we are able to swiftly deal with the issue. During

the traversal all vertices accessed at the current cell may be temporarily stored here in between being loaded and stored, therefore allowing us to load and store them in any order and not worry about the intra-cell order.

The ability to load multiple vertices of a cell at once and being allowed to re-order them undermines how the regions are used in the argumentation of the counterexample and allows us to store more vertices onto the same stack. In Section 5.3.1 we will show how the subdivision of the counterexample can be traversed with only one intermediate stack.

We have defined a model which describes all access and storage times of the vertices during the traversal, but its characteristics heavily depend on the traversal order and the choices made when a cell is visited multiple times. Assuming that the traversal order and the intervals (i.e. access times) of all vertices are pre-established, we will first look at two approaches which define efficient intermediate data access and storage operations in Sections 5.3 and 5.4. Thereafter Sections 5.5 and 5.6 discuss heuristic solutions for the access time choice problem and the traversal order which improve the input for the two approaches.

5.3 Overlap graph coloring

Given the set of intervals resulting from the access model of Section 5.2 we will now look at constructing an efficient data structure for storing the vertices during the traversal. Considering the success of the 2D Stack & Stream algorithm the first instinct is to once again use a system of stacks. We will now show how such a system can be defined by the *interval overlap graph coloring*.

5.3.1 Sharing stacks

In the context of the abstract data access model we are no longer restricted to the strict left/right division along the traversal order, or to a constant number of stacks. As a result it is possible to pick up a vertex from a stack, but afterwards place it back on a different one. Thus a single interval of a vertex corresponds to the time that it resides on "a" stack, and it can switch stacks at the endpoints. Hence we should not assign vertices to stacks, but rather assign all its intervals to stacks separately.

In order to use a stack data structure we need to carefully plan the data accesses that use it so that they do not block each other. When two intervals do not intersect in time they can trivially be assigned to the same stack. If they do intersect then one is stored on top of the other and it should be removed before the next access to the interval below it. It follows that any pair of intervals I_1 and I_2 can be assigned to the same stack if either (1) I_1 contains I_2 of vice versa, or (2) they are disjunct. Here containment is defined as $I_i \subseteq I_j$. If a pair of intervals does *not* satisfy (1) or (2) we say they *partially overlap*. Two identical intervals do not form a degenerate case because the abstract data access model allows us to abstract from the intra-cell access order. It therefore does not matter if we store one on top of the other, or vice versa.

When the calculations at the current cell are done and we need to store the vertices again we store them in latest-endtime-first order of their next intervals (ties can be resolved either way). This way we guarantee that the vertices whose intervals share an endpoint at the current cell can always be picked up and re-distributed over the stacks without blocking each other.

Using the partial overlap definition and the abstract data access model we are able to schedule strictly more intervals onto a single stack without violating the stack property than Stack & Stream because we do not need to worry about the intra-cell order. In fact, if we look back at Figure 7c we can see that there now exists a solution to the counterexample of Section 5.1 that only uses one stack. However, this all depends on our choice for the access order problem. If v_1 had accessed A

at time 1 instead of 5 then the first interval of v_1 (which would then be [1,4]) would have partially overlapped with the second interval of v_2 ([2,6]) and a second stack would be required.

5.3.2 Assigning stacks

Now that we have a defined which intervals can share a stack we can look at assigning stack numbers to them. Obviously, our aim is to use as few stacks as possible. This minimization problem can be solved by a standard graph coloring algorithm after casting our intervals to a graph as follows. Let graph $G = (V, E)$ be a graph consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. Vertex set V contains a vertex v for every interval in $\mathcal{I}$, and E contains an edge for every pair of intervals $(v, v') \subseteq V \times V$ that partially overlap. Thus no two vertices connected by an edge can be assigned to the same stack. It follows that a set of intervals that can be assigned to the same stack corresponds to an independent set in G.

To find the minimal number of stacks we need to find the minimal number of independent sets that covers V. This matches the vertex coloring problem of G, i.e. assign a color to each vertex in V such that no two vertices who are connected by an edge have the same color. Coloring two vertices the same color thus corresponds to assigning two intervals to the same stack. To find the minimal number of stacks we need to find the minimal number of colors needed, i.e. the *chromatic number* χ of G. We will use the terms vertex coloring and graph coloring interchangeably.

Although every set of intervals can be translated to a graph, not every graph can be translated to a set of intervals on $\mathbb{N}$. It turns out that we are dealing with a special class of graphs called *interval overlap graphs*, or the equivalent class *circle graphs* [10]. For overlap graphs it is known that the coloring problem is NP-hard [10]. However, this does not imply that the coloring problem is hard for our specific kind of overlap graph. The data accesses of the 2D Stack & Stream algorithm can also interpreted by the data access model developed for 3D. The 2D problem can be solved in linear time and only needs a single intermediate stack, which corresponds to an overlap graph with $\chi = 1$. Therefore our graph may be a subclass of the overlap graphs class, but we have not been able to identify any such class. It is an open question whether for arbitrary 3D subdivisions a traversal and vertex access order can be constructed such that the chromatic number of the corresponding overlap graph is bounded by a constant. For now a brute force search is infeasible even for small subdivisions as we need to solve the NP-hard coloring problem for all resulting interval sets of all element orders (i.e. cell orders) and all access time choices. For now the results are constrained by whichever (overlap) graph coloring algorithm uses the least amount of colors in a practically acceptable running time.

5.3.3 Graph construction

The overlap graph of a set of intervals can be trivially constructed in $O(|V|^2)$ time by iterating over all $\binom{|V|}{2}$ possible edges and checking if their corresponding intervals intersect. This is not efficient nor acceptable for the input sizes we are dealing with (as we will see later). We will now present an alternative $O(|V| \log |V| + |E|)$ sweep algorithm.

We start by sorting all intervals on their start time and put them into queue Q. During the algorithm we will sweep over the interval domain, load intervals as we encounter them, and construct the graph on the fly. The current position of the sweep coincides with $startTime(head(Q))$, i.e. $startTime(q)$. During the sweep we will maintain a list structure L which contains all intervals which at that point in time are intersected by the position of the sweep, sorted ascending by end time. Let $\mathcal{I}$ be the set of all intervals. The overlap graph is then constructed by Algorithm *ConstructGraph*.

Algorithm *ConstructGraph*(I)
(∗ Constructs the overlap graph of interval set $\mathcal{I}$. ∗)
1. Sort $\mathcal{I}$ on start time and store the result in queue Q
2. $V = \emptyset$, $E = \emptyset$, $L = \emptyset$
3. **while** $Q \neq \emptyset$
4. **while** $endTime(head(L)) \leq startTime(head(Q))$
5. $removeHead(L)$
6. $q \leftarrow removeHead(Q)$
7. $V \leftarrow V \cup q$
8. $p \leftarrow head(L)$
9. **while** $endTime(q) > endTime(p)$
10. **if** $startTime(p) \neq startTime(q)$
11. $E \leftarrow E \cup (p, q)$
12. $p \leftarrow next(p)$
13. Insert q in L between $prev(p)$ and p
14. **return** (V, E)

During the sweep of Q we maintain the invariants that (1) L contains all intervals intersected by the current position of the sweep, and (2) L is sorted ascending on latest end time.

From the definition of the sweep line it follows that (1) is equivalent to '$startTime(q)$ is contained by all intervals of L'. Thus q partially overlaps with an interval p from L if its end time is not contained by it, i.e. $endTime(q) > endTime(p)$. An exception exists when p and q share the same start time, in which case p contains q and they do not partially overlap. Because L is sorted on end time we can sequentially scan through it and report all intervals partially overlapping with q as we go. When we reach the point where $endTime(q) \leq endTime(p)$ we know all remaining intervals of L contain contain q. We insert q into L, thus maintaining invariant (2), and stop. In conjunction with the while loop at lines 4-5 this also maintains invariant (1) for the next iteration.

The total execution time of lines 4-5 is $O(|\mathcal{I}|) = O(|V|)$ because every interval is only added to, and removed from, L once. Furthermore every iteration of the outer while loop (lines 3-13) directly corresponds to a vertex in V, and every iteration of the inner while loop (lines 9-12) directly corresponds to an edge in E (barring a constant factor which shares the start time of q). Therefore the total running time of Algorithm *ConstructGraph* is output sensitive and, if we include the sorting step, we find a running time of $O(|V| \log |V| + |E|)$.

5.3.4 Coloring algorithm

The running time of any algorithm that obtains a coloring should not be of order $O(\mathcal{I}^2)$ or higher in order to be feasible due to the high number of intervals. However, most literature concerned with graph colorings either provides theoretical hardness bounds or is happy with an approximation in polynomial time where the polynomial degree is not of much concern because it is still a massive improvement over exponential time. Heuristic solutions do not provide a guaranteed bound on the result, but they are often straightforward and quick to implement. We therefore use the heuristic used by the Iterated Greedy algorithm [11], which is a linear time algorithm with minimal additional data structures.

The Iterated Greedy algorithm determines an order in which the vertices will be colored up front (i.e. a permutation of V). Thereafter they are colored in order using the heuristic of assigning the minimum color that does not cause a conflict, i.e. the minimum color (starting from '1') that does not occur among those vertices connected to it by an edge. In the worst case this heuristic yields an $O(n)$-approximation as demonstrated in Figure 8. This result is then improved upon by running many iterations of the heuristic for which, by careful choice of the permutation, it can be proven that they use no more colors than the previous iteration.

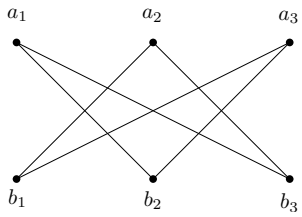


Figure 8: A worst case example for the proposed heuristic. The graph consists of $2n$ vertices with edges between a_i and b_j for all $i \neq j$. The order $a_1, b_1, a_2, b_2, \ldots$ results in n colors whilst the graphs is 2-colorable.

It is straightforward to see that the running time of one iteration of Iterated Greedy is n times the complexity of assigning a color to a vertex, so how do we assign a color? Initially all vertices are uncolor, which we mark by color number zero. Now consider for example a vertex of degree six. Iterating over its adjacent vertices we see the colors $c_5, c_3, c_1, c_5, c_2, c_6$ and we should deduce that c_4 is the minimum color that does not cause a conflict (we cannot color with color number zero). The straightforward solution is to sort all colors and then find the smallest positive unlisted integer with a sequential scan. This would results in an algorithm of $O(\sum_{v \in V} degree(v) \log degree(v))$. However, we would like to avoid the sorting complexity bound and instead assign colors in time linear in the degree of the vertex. This can be achieved as follows.

Let n be the degree of a vertex v in the graph. When we iterate over its adjacent vertices in the graph we see n colors which may or may not contain duplicates. This means that there must be at least one color in the range $[1, n+1]$ which does not occur in this series. Therefore the problem of finding the smallest unlisted number reduces to the problem of finding the smallest unlisted number less than $n+1$. This means that we do not have to keep track of numbers that are greater of equal to n. Hence it suffices to use a boolean array of size n to keep track of the colors so far. When we see the color c_i we force the boolean variable at index i to true. After we have iterated over all adjacent vertices the minimum color which can be assigned without conflict then corresponds to the lowest index of the array which holds a false value (or $n+1$ in the case that all values are true). This index may be found by a sequential scan.

The Delaunay tetrahedralization of n 3D points chosen uniformly at random from a sphere the expected number of cells is $6.77n$ [12]. Since every tetrahedron contributes to four adjacent vertices this means we have approximately $4 * 6.77 = 27.08n$ vertex accesses in total which result in an average of 26.08 intervals per vertex. Since the graph contains a vertex for each interval it will be massive (subdivision inputs range into the millions of cells) and Iterative Greedy is a costly algorithm. However, the first iteration can be implemented efficiently by a sweep procedure very similar to Algorithm *ConstructGraph* of Section 5.3.3 without actually constructing the graph. We therefore choose to only implement the first iteration. The structure of *ConstructGraph* and its invariants are maintained, but all graph operations are replaced by the coloring heuristic. The result is Algorithm *OverlapColoring*.

The while-loop at lines 9-12 accumulates the colors used by those intervals who partially overlap with q (i.e. which would be connected to q by an edge in the overlap graph) in $usedColors$. Note that all interval in L have already been colored. We mark all colors in the range $[1, |usedColors|]$ in boolean array $b[]$. The minimum color for q that does not cause a conflict is then found by a scan of $b[]$ for the lowest index of the array which holds a false value, and is assigned to q. The running time and correctness analysis is identical to that of Algorithm *ConstructGraph* except that we can no longer call it output-sensitive because the overlap graph is never constructed. The introduction of the scan over $b[]$ does not increase the complexity bound because its use sums up to the order of the number of edges in the graph. Thus we are able to obtain a valid coloring in

the same time it would otherwise take to just construct the graph.

Algorithm *OverlapColoring*(I)
(∗ Constructs an overlap coloring of interval set $\mathcal{I}$. ∗)
1. Sort $\mathcal{I}$ on start time and store the result in queue Q
2. $L = \emptyset$
3. **while** $Q \neq \emptyset$
4. **while** $endTime(head(L)) \leq startTime(head(Q))$
5. $removeHead(L)$
6. $q \leftarrow removeHead(Q)$
7. $p \leftarrow head(L)$
8. $usedColors \leftarrow \emptyset$
9. **while** $endTime(q) > endTime(p)$
10. **if** $startTime(p) \neq startTime(q)$
11. $usedColors \leftarrow usedColors \cup color(p)$
12. $p \leftarrow next(p)$
13. Let $b[]$ be a size $|usedColors|$ boolean array initialized to false
14. **for all** $c_i \in usedColors$
15. **if** $i \leq size(b)$
16. b[i] ← true
17. $color(q) \leftarrow$ Smallest false index of b
18. Insert q in L between $prev(p)$ and p

5.4 Interval graph coloring

We will now look at an alternative approach for constructing an efficient data structure for storing the vertices during the traversal. It is aimed at making the best selection of vertices to keep inside cache memory and is based on the concepts of cache coloring. We will show that it is closely related to the previous approach in the sense that it again boils down to a graph coloring problem, this time involving the *interval graph coloring* (note the absence of the 'overlap' infix). We again assume that the set of intervals resulting from the access model of Section 5.2 is pre-established.

5.4.1 Memory-constrained data structures

Instead of optimizing the efficiency of the I/O-operations the memory-constrained data structure approach is based on making the most of the data that already resides in the internal memory, i.e. the cache memory. The subdivision as a whole does not fit into the cache memory as a whole, but by making a smart selection of which parts we keep cached at certain points during the iteration we aim to reduce the number of cache misses. Vertices are therefore assigned to slots in the internal memory instead of stacks, where the number of slots is equal to the cache size (in bytes) divided by the size of a vertex's variables. The stack data structure for the cell data remains unchanged.

We have already seen how the intervals of the abstract data access model relate to the points in time during which a vertex needs to be stored somewhere. Similar to the stack approach we now need to define how intervals can make use of the same storage structure (i.e. slots). Since a slot can only hold one vertex at a time this is rather straightforward – Two intervals may use the same slot during the iteration if and only if they are disjunct. The start time of an interval may match the end time of the previous interval that used the slot without problems because the abstract data access model allows us to abstract from the intra-cell access order.

5.4.2 Assigning slots

Now that we have a defined which intervals can share a slot we can look at assigning slot numbers to them. When an interval is assigned a slot number its associated vertex will occupy the slot in cache memory for the duration of the interval and hence will not incur a cache miss when it is accessed at the end time of the interval. Obviously, we want to assign as many intervals as possible a slot number as to minimize the number of cache misses.

This minimization problem can be solved by a standard graph coloring algorithm after casting our intervals to a graph as follows. Let graph $G = (V, E)$ be a graph consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. Vertex set V contains a vertex v for every interval in $\mathcal{I}$, and E contains an edge for every pair of intervals $(v, v') \subseteq V \times V$ that partially overlap. Thus no two vertices connected by an edge can be assigned to the same slot. It follows that a set of intervals that can be assigned to the same slot corresponds to an independent set in G.

To assign the maximum number of intervals a stack number we need to find the maximum number of vertices that can be covered by k independent sets, where k is the number of slots available in the cache memory. This matches the k-colorability problem of G, i.e. assign the maximum number of vertices one color out of k such that no two vertices who are connected by an edge have the same color. Coloring two vertices the same color thus corresponds to assigning two intervals to the same slot.

The resulting graph is again a special class of graphs known as *interval graphs* [13]. For interval graphs the k-colorability problem of the corresponding set of intervals $\mathcal{I}$ can be solved in $O(\mathcal{I})$ time after sorting [14] [3]. The algorithm works directly on the intervals thereby avoiding the factor $|E|$ associated with the graph's construction and achieving linearity in $|V|$, i.e. $|\mathcal{I}|$. Achieving optimality of the interval coloring in linear time is a very strong result indeed as all problems so far have been of a combinatorial nature. By choosing k equal to the number of available slots in cache-memory we maximize the number of cache hits.

5.4.3 Data structure for uncolored intervals

By definition of the external-memory model of computation the whole data set does not fit into the cache memory. Although the access times of the vertices are spread over the traversal and they are not all needed at once we still expect that we cannot color 100% of the intervals (and otherwise this definitely holds for bigger input sizes). The results presented in Section 6.4 show this assumption is correct. We have an optimal interval coloring algorithm, so when an interval remains uncolored it means that it is inefficient to store its corresponding vertex in cache memory during this time (i.e. the slots can be put to better use). We therefore need to design an additional data structure in external memory to store vertices at for the durations of the uncolored intervals. By design this means that the next access of this vertex will incur a cache miss. We have completely knowledge of the next access times of these vertices (i.e. the vertex end points) and can take this information into account when designing the data structure. We will discuss two candidates for this data structure: A priority queue and an array.

In the priority queue approach the vertices will be tagged with the end time of their corresponding uncolored interval. They are then inserted into the queue and at the time they are required during the traversal they will be the minimum element in the queue and can be quickly retrieved. Priority queues are closely related to sorting and the sorting lower bound of $\Omega(n \log n)$ comparisons between n input elements applies to them. However, they are a heavily studied subject and many variants have been designed for many different purposes [17]. This includes a priority queue in the

[3] We note that a very straightforward implementation may be achieved by replacing Carlisle et al.'s use of a special-case disjoint set union data structure [15] by the general-case variant by Tarjan [16] since the asymptotic complexity will still be dominated by the sorting step.

external-memory model, for which an optimal cache-oblivious priority queue has been designed by Arge et al. [18]. In the RAM model queues have been developed with interesting distribution-sensitive properties. The Fishspear priority queue [19] has the property that the amortized cost of handling an element x is logarithmic its maximum position before extraction, i.e. over time the largest number of elements less than x simultaneously in the priority queue. Furthermore, Elmasry et al. present a priority queue with the time-finger property [20]: It supports insertions in worst-case constant time, and delete-min, access-min, delete, and decrease-key of an element x in worst-case $O(\log(\min\{w_x, q_x\}))$ time, where w_x (respectively, q_x) is the number of elements that were accessed after (respectively, before) the last access to x (i.e. insertion in our case) and are still in the priority queue at the time when the corresponding operation is performed.

The array approach does exactly what is says on the box – we index everything in an array. We reserve an array in external memory whose length is big enough to accommodate a number of vertices equal to the number of *uncolored* intervals. Since a vertex can have multiple uncolored intervals this structure may be bigger than actual number of vertices, but because it is stored in external memory (which may be viewed as infinitely large) this is not a problem.

The intervals directly correspond to the access times during the traversal so, if we assign an interval the array index that matches its position in the ascending-on-end-time sorted set of uncolored intervals, then they are loaded from the array sequentially, i.e. with spatial locality. This essentially makes it a stream for read operations and makes the design of such a big array worthwhile over just using an array of size $|V|$. We note that during execution most of the array slots will be empty. The indices before the read position are empty by construction, and the indices beyond it are sporadically written to to 'fill in the gaps' such that by the time the read position arrives there all vertices can be found in contiguous memory.

In general the space requirement of the array approach is a bit ridiculous, but for our application this is irrelevant. The array also requires a write per interval due to the irregular pattern in which it writes to memory slots, whereas priority queues can do buffered batch writes (i.e. writes with spatial locality). Conversely, the structure that the priority queue provides has overhead associated with it. Any priority queue has to make element comparisons at some point to sort its content. The priority queues with distribution-sensitive properties are not known to have cache-efficient operations, and the cache-oblivious queues which have optimal block operations lack the distribution-sensitive properties. This makes it hard to support insert and delete-min operations which together, amortized per vertex, stay below the time of the array's write penalty and efficient loading. Furthermore, all of them have pointer memory overhead and partially reside inside cache memory, occupying precious space which could otherwise be used for more slots (i.e. more colors).

In the end the exact values of the write time, the block size and the interval length/distribution are decisive in determining which approach is the most efficient. We will come back to this in the discussion of the results in Section 6.

5.5 Choosing data access times

We have seen that a vertex's access times depend on when we visit its adjacent cells during the traversal. If they are visited once, the access time is set, but if they are visited multiple times we can choose during which visit we will make the computation. So far we have assumed that this choice has been pre-established and that we are left with the resulting interval set. We will now present a heuristic solution for creating 'good' intervals.

5.5.1 Selection heuristic

Although it is possible to make an optimal choice for the 2D subdivision with use of the geometric information, making an optimal choice for the abstract problem is hard. It requires an algorithm

to have the foresight to know how good the solution of the resulting graph coloring problem is. We therefore use a heuristic solution.

After the choices have been made each vertex has a set of consecutive intervals $[t_1, t_2], [t_2, t_3], \ldots, [t_{k-1}, t_k]$ associated with it. The number of such intervals is equal to the number of adjacent cells minus one. Consider two random intervals in the domain of 1 to the length of the cell traversal. We know nothing about the intervals, but we can say is that the probability of them being disjunct increases as their lengths approach 1. Disjunct intervals lead to a sparser graph both for the interval and overlap graph and hence improve the interval set input for the coloring algorithms. We therefore aim to to minimize the average interval length. Furthermore, smaller intervals also improve temporal locality and, for the slots approach, maximize the time during which a slot can hold another vertex. Since the intervals of a vertex are consecutive it follows that to minimize the average length we need to minimize $t_k - t_1$ (for each vertex separately).

The heuristic implicitly assumes that the distribution of the intervals over the traversal domain is uniform. This may look farfetched at first, but for tetrahedralizations it makes sense because each cell is associated with a small constant number of vertices. Our heuristic is applied to each vertex independently and does not consider the choice of other vertices. We therefore argue that it does not significantly favour any part of the traversal sequence more than others and that the result should be approximately uniform.

5.5.2 Implementing the heuristic

In this section the interval (length) of a vertex should be interpreted as the sum of (the lengths of) all its intervals, i.e. we are only interested in $[1, t_k]$ and not in the actual intervals $[t_1, t_2], [t_2, t_3], \ldots, [t_{k-1}, t_k]$.

Like the definition of the intervals in the abstract data access model we can use the same enumeration scheme to express at what time a cell can be accessed. For example a cell A has the access time set $\{2, 6, 18\}$ associated with it if it is the 2nd, 6th and 18th cell in the traversal sequence. Note that each access time can only occur once in the set a of cell – we do not have to deal with duplicates. The final interval of a vertex should thus span at least one value from the access time set of each of its adjacent cells. To find the optimal interval we will first define the notions of a minimal and minimum interval.

- An interval is considered a *minimal interval* if it contains at least one value from the access time set of each of its adjacent cells and every sub-interval of it does not, i.e. it cannot be shrunk.
- The *minimum interval* is the shortest interval of all minimal intervals.

The definition of a minimal interval implies that the cells at the endpoints of the interval only occur once. This will be the crux of our algorithm. We start with the interval $(0, 0)$. Whilst the interval does not span at least one access time of all m_v adjacent cells we increment the interval's right endpoint (rep) until it does. Note that the cell at the right endpoint now only occurs once in the interval. If at this point the left endpoint (lep) has become non-unique, i.e. more than once of its access times are contained by (lep, rep), we increment the left endpoint until it is. At the end of this procedure we have found a minimal interval which is then reported. The left endpoint is then incremented by one and the procedure is repeated to find the next minimal interval.

During execution we need to keep track of the number of occurrences of each adjacent cell's access times in (lep, rep), the number of distinct cells covered by the interval and the best minimal interval so far. Let m_v denote the number of adjacent cells of vertex v, S_v be the union of the m_v access time sets, and function $cell(s)$ return the associated cell id for $s \in S_v$. A minimum interval can then be found by Algorithm *MinimizeIntervalLength*.

Algorithm *MinimizeIntervalLength*(v)
$(*$ Finds a minimum interval of vertex v $*)$
1. Sort S_v on access time and store it in an array
2. $occurrences \leftarrow$ size m_v array initialized to zero
3. $lep \leftarrow 0$, $rep \leftarrow$ -1, $\#distinct \leftarrow 0$, $bestSoFar \leftarrow (0, \infty)$
4. **while** $rep < |S| - 1$
5. **while** $\#distinct \neq m_v$ **and** $rep < |S| - 1$
6. $rep \leftarrow rep + 1$
7. $occurences[cell(S[rep])] \leftarrow occurences[cell(S[rep])] + 1$
8. **if** $occurences[cell(S[rep])] = 1$ (* Up from zero *)
9. $\#distinct \leftarrow \#distinct + 1$
10. **while** $occurences[cell(S[lep])] > 1$
11. $lep \leftarrow lep + 1$
12. $occurences(cell(S[lep])) \leftarrow occurences(cell(S[lep])) - 1$
13. **if** $\#distinct = m_v$ **and** $S[rep] - S[lep] < lengthOf(bestSoFar)$
14. $bestSoFar \leftarrow (lep, rep)$
15. $lep \leftarrow lep + 1$
16. $occurences(cell(S[lep])) \leftarrow occurences(cell(S[lep])) - 1$
17. $\#distinct \leftarrow \#distinct - 1$
18. **for all** m_v access time sets
19. Make an arbitrary access time choice within $bestSoFar$.
20. **return** The resulting intervals of the access time choices

During execution Algorithm *MinimizeIntervalLength* considers all possible values for lep at some point, except those it skips over while shrinking the interval at lines 11-13. For each lep value the while loop at lines 6-10 guarantees by construction that we find the smallest possible value for rep such that interval (lep, rep) contains all m_v adjacent cells of v, and only then the shrinking step happens. Thus the only lep values we skip over are those who can never be the left end point of a minimum interval because, by construction, they form a non-unique left endpoint at that point and there exists a smaller minimal interval. Therefore *MinimizeIntervalLength* iterates over all minimal intervals and is able to find a minimum interval.

Every access time in S is accessed a constant number of times in between being included in and removed from $bestSoFar$. This means the running time is dominated by the sorting step and *MinimizeIntervalLength* runs in $O(m_v \log m_v)$ time with $O(m_v)$ storage. Note that Algorithm *MinimizeIntervalLength* is run for each vertex separately and that the expected average for m_v, as we argued in Section 5.3.4, is ~ 26.

5.6 Traversal order alternatives

At the abstract data access model we noted that the traversal order directly influences the possibilities for the resulting intervals. For the 2D variant and the running example for 3D we have thus far always used a spanning treewalk order, but since we do not use any assumptions or geometrical information in the abstract model it is not a requirement. Although the model can be applied to any order we want to find ourselves in one of two practical situations where the cell order is either: (1) implementable with one intermediate cell stack, or (2) the cell order is streamable (i.e. it is a permutation of C).

The first access of a cell is always from the input stack (stream) and any subsequent accesses to it use the intermediate stack. Therefore an order can be implemented with one intermediate stack if it follows a most recently used order for any subsequent accesses after the first access, i.e. an order for which it can always be found on top of the intermediate stack. We have proven in Section 4.2 that a spanning treewalk follows the most recently used order and thus commits to the

stack property, but there are infinitely more orders which also have this property. We will now specify this set of orders.

Let S be the string of cells denoting the (partial) cell traversal order, $S[i:j]$ $(i \leq j)$ be the substring starting from index i and ending at index j, $+\!\!+$ be the string concatenation operator, and c^n be the string of n consecutive c's (i.e. $c^1 = c, c^2 = cc$, etc). Every element in the set of all cell traversal orders which can be implemented with a single intermediate stack for a set of cells C can then be returned by the following recursive specification with initial call $Order(C, \emptyset)$. Since all occurrences of a cell $c \in C$ are inserted consecutively c does not block the transition from $S[i]$ to $S[i+1]$ since it will have been moved to the output stack before then. This reasoning can be applied inductively to argue the correctness of the returned string. One can easily verify that a depth-first traversal of any tree adheres to this definition. Permutation are also included and can be generated by choosing $n = 1$ for all $c \in C$.

$$Order(C, S) = \begin{cases} \text{if } C \neq \emptyset & Order(C \setminus \{c\},\ S[1:i] +\!\!+ c^n +\!\!+ S[i+1, |S|]), \\ & \quad \text{for } c \in C,\ i \in [1, |S|], \text{ and } n \in \mathbb{N}^+. \\ \text{Otherwise} & \text{Return } S \end{cases}$$

We know that traversal orders matching the specification can be efficiently implemented with the familiar system of stacks, but we still need to identify which orders lead to a better input for the graph coloring. It is not surprising that this leads us back to the locality principles. We observe that two adjacent cells of a tetrahedralization have three out of their four adjacent vertices in common. When they are traversed consecutively this, in the best case, results in three intervals of length one. Furthermore, any other cells adjacent to these vertices lie nearby in the subdivision and are likely to be accessed soon if we keep following the neighbour-to-neighbour principle. Note that the data access time heuristic of Section 5.5 (minimize the average interval length) also supplements this principle perfectly and significantly improves the probability of actually ending up with the intervals of length one. This directly leads us back to the spanning treewalk approach defined on the dual of the subdivision. More specifically, since different spanning trees have different properties, we will be looking at depth-first and breadth-first spanning trees.

Depth-first treewalks maximize the string of traversing through unvisited cells before returning to parent cells and thus, intuitively, open up the door for many length-one intervals. Conversely, breadth-first trees are much more balanced and have shorter string of unvisited cells, but cells adjacent to the same vertex are likely to have approximately the same depth in the tree. Therefore the traversal length to go up in the tree and down into another branch to reach the adjacent cell (assuming there is no parent-child relation) should be of order $O(\log n)$ instead of $O(n)$. This should result in smaller intervals overall (i.e., following the notation of the selection heuristic, $t_k - t_1$ is minimized).

It should be obvious that for streamable traversal orders a random permutation of C will not result in the best set of intervals since it has very poor locality. At first glance a Hamiltonian path through the subdivision seems to have good locality, but for arbitrary subdivisions finding one is a NP-complete problem [9] (if it exists at all) and different Hamiltonian paths may also have very different locality properties. We therefore extract a permutation from the spanning tree traversal by means of pruning. The pruned version of a spanning treewalk traversal order is the original depth- or breadth- first spanning tree traversal order in which of each cell only its first access is preserved, resulting in a permutation with, hopefully, a high degree of locality.

If we allow for two intermediate cell stacks instead of one we note that it is possible to treat them as the input/output stack of the original 2D Stack & Stream algorithm and transfer all $|C|$ cells from one stack to the other as many times as we want before moving the cells to the actual output stack, i.e. one iteration consists of many subdivision traversals in which every cell is visited once. If we transfer all cells back and forth over the two intermediate cell stacks $|V|$ times during

one iteration then all vertices can trivially be stored on one intermediate vertex stack/slot by only accessing one vertex per full transfer (i.e. subdivision traversal). Needless to say this is highly inefficient, but it illustrates the point that there exists a trade-off between the traversal length and the optimal solution in terms of the graph coloring. Even with one intermediate cell stack visiting a selection of cells multiple times in a specific order may improve the graph coloring, as illustrated by the generalized 2D approach.

This concludes the discussion of the last variable under our control. The global problem can thus be formulated as:

> Minimize the number of colors of the overlap graph coloring,
> **or** maximize the number of colored intervals of the interval graph k-coloring (for k equals the number of slots in cache memory)
>
> - over all possible traversal orders matching the specification of $Order(C, S)$ (of practical length),
> - over all possible access time choices.

We had assumed that the subdivision of the input space is pre-determined, otherwise this would be another variable.

5.7 Data streams and structures overview

In the previous sections we discussed the theory behind the optimization problem and as a result we did not focus on the implementation details. We will now fill in the blanks and provide an overview of the proposed data streams and structures.

Most of the data structures are analogous to the 2D variant as discussed in Section 4. However, we make a distinction based on whether the cell traversal order is a permutation or an order in which cells are visited multiple times according to the specification of the previous section (this includes spanning treewalks).

Analogously to the generalized 2D Stack & Stream algorithm we cannot deduce from which stack a vertex should be popped, and to which stack it should be pushed, locally and thus need to store it explicitly. However, in contrast to the 2D traversal the number of stacks is not limited to a left and right hand stack, thus we require integers instead of a few bits of additional information (i.e. the assigned colors) [4]. Since this information is not of negligible size and is only used once we no longer wish to store it with the cell and instead stream it. This forms the main data stream that drives the traversal.

At each step of the cell traversal we access the main data stream and retrieve the load/store locations of all vertices that need to be accessed. An equal number of cell matrix columns will then be loaded from the cell matrix stream. What happens next depends on the traversal order. If it is a permutation order then all contributions can immediately be stored at the right vertices, we store the vertices to the right stack/slot, and we can move on to the next cell. If it is not a permutation order then a system of stacks (i.e. an input, one intermediate, and output stack) is also involved for passing on partial contributions. To distinguish between loading from the input or intermediate stack, resp. storing to the intermediate or output stack, an additional two bits of information are required in the main stream. The partial contributions are stored to this structure, the vertices to their store stack/slot, and we move on to the next cell in the traversal. To avoid write latency all streams can be implemented with an array instead of two stacks, since their values will not be altered.

[4] The size of the integer depends on the number of colors used.

Iterations are once again possible by inverting the traversal order and a backwards interpretation of all data structures.

5.8 Incorporating edge and facet variables

The abstract data access model describes the accesses to data element at points in time. So far we have only used it to design an efficient data structure for intermediate vertex storage, but analogously we can also employ it to describe edges and facets (or cells for that matter, but their data accesses are already efficient by design).

If edge or facet variables are added to the model this therefore simply results in more intervals that need to be colored. In the case of the overlap coloring they can simply be stored among the vertices and share stacks with them. In the case of the interval coloring the slot size should be adjusted to the maximum storage capacity required among the vertex, edge and/or facet variable(s).

Loading these variables from the correct stack or slot also requires explicit references, which should be included into the main data stream discussed in the previous section.

6 Experimental Results

In this section we present the results of our approach to the 3D finite element method. We do not compare our approach to other implementations directly because industrial implementations also employ other techniques to improve performance (examples include multi-core-aware parallelization, GPU accelerated computing, cache tuning, compiler optimization and more). We therefore measure the effectiveness in terms of the optimization problem instead, i.e. the characteristics of the graph colorings. Section 6.1 describes the setup of our approach, Sections 6.3 and 6.4 present the results of the overlap resp. interval coloring, and in Section 6.6 we present a discussion of the obtained results and their implications.

6.1 Setup

The input subdivision is a Delaunay tetrahedralization of n 3D points randomly picked from a uniform distribution inside the positive unit cube. The implementation was made in *MATLAB R2014b* and uses the twister random number generator with the seed 49874574. The full computational code is given below, after which the resulting structures P, DT, and the neighbour relations between the cells of DT (which can be queried from DT with function $neighbors(DT)$) are written to a text file. During experimentations we noticed that the exceptions thrown by *MATLAB* indicate that behind the scenes the implementation is based on *qhull* (http://www.qhull.org/). The documentation of the delaunayTriangulation class, including all methods on the resulting triangulation object, can be found at *MATLAB*'s website [21].

```
rng(49874574,'twister');          //Set rng seed and rng generator type
P = rand(n,3);                    //Generate a size n uniform random 3D pointset
DT = delaunayTriangulation(P);    //Compute the Delaunay triangulation of P
```

All spanning trees are rooted at the first cell that is written to file by *MATLAB*, and the children of a cell are visited in the order they are listed by $neighbors(DT)$. Finally, we note that inside the reported minimum interval of Algorithm *MinimizeIntervalLength* we choose the earliest access time for each cell (i.e. our choice at lines 18-19 of *MinimizeIntervalLength* is not random). The correctness of the implementation was verified by a brute-force comparison of the color assignment of all $O(|\mathcal{I}|)$ interval pairs for small input sizes ($n = 1,000$), as well as by a checksum approach: Using a uniform weighting of one and vertex-to-vertex contribution equal to their vertex ID's we verified that at the end of a traversal the values of all accumulating variables are equal to the known sum of adjacent vertex ID's in the subdivision.

6.2 Subdivision and interval characteristics

The Delaunay tetrahedralization gives us a subdivision of which the element sizes are shown in Table 2. The average number of cells and intervals per vertex lie in the range $6.671 - 6.719$ and $25.68 - 25.88$ respectively, which matches our earlier theoretical prediction. Due to the high number of intervals, or objects in general, our Java implementation is not able to handle the construction of the interval set for point sets of $n \geq 90,000$, for any traversal order and consequently throws an

	Point set size (n)							
	10,000	**20,000**	**30,000**	**40,000**	**50,000**	**60,000**	**70,000**	**80,000**
Cells	66,356	133,413	200515	267,651	335,449	402,930	470,240	537,503
Intervals	22,5424	513,652	772060	1,030,604	1,291,796	1,551,720	1,810,960	2,070,012

Table 2: The resulting number of cells and intervals from the Delaunay tetrahedralization of n uniformly distributed points in the 3D unit cube.

out-of-memory exception. The additional overhead of the interval and overlap coloring algorithm causes it to break down for even smaller input sizes. As a result we are only able to obtain the interval coloring of subdivisions of $\leq 80,000$ vertices, and the overlap coloring of subdivisions of $\leq 50,000$ vertices. To emphasize the extend of the memory requirement: The number of executions of line 11 of Algorithm *OverlapColoring* during the coloring of the $n = 50,000$ DF overlap graph (which equals the number of edges) is 1,308,211,509. This corresponds to an average of 1013 edges per vertex in the graph. Storing this graph in full and/or using any non-linear coloring algorithm is inadvisable.

The characteristics of the interval length distribution for our four traversal orders Depth First (DF) and Breadth First (BF) spanning tree treewalks, and their pruned variants (DF_P and BF_P)) are presented in Figure 9. To provide some context the results of a random permutation are also included. These may be viewed as an absolute lowerbound – any structured approach should outperform it. We will therefore not discuss or compare its results and only include it in figures where it does not affect the interpretability of other results (e.g. by completely distorting the plot due to extreme values relative to the rest). We used the largest possible input size for which just the interval set could be constructed for all traversal orders, which was for $n = 75,000$. We note that 57-73% of all intervals have length one, and 76-81% have a length of ten or smaller. Furthermore, the different properties of breadth and depth first traversals and their pruned variants result in different length distributions with BF_P coming out on top from length eight onward. This is consistent with the intuition discussed in Section 5.6 – depth-first traversals have more length-one intervals whereas breadth-first traversals have smaller intervals overall. It is perhaps surprising that the tipping point between the two approaching lies so low.

The improvement of pruning on the interval length distribution of breadth-first traversals follows from the removal of unnecessary traversal steps. Due to the breadth first approach any two cells adjacent to the same vertex will have approximately be the same depth in the tree. If they have a parent-child relation then the intervals will be small, but if they lie on different branches then they will be longer because we need to traverse up the tree and then down again. The pruned variant only preserves the first visit of a cell and therefore does not travel back up the tree, but instead immediately moves on to the child of the last cell of which not all children have been visited. This shortens the traversal length between two cells adjacent to the same vertex, and hence the interval length.

For depth-first traversals the adjacent cells of a vertex are not expected to approximately have the same depth, but they should still benefit from the pruning effect. Somehow this is not reflected in the length distribution. This may be explained as follows: From the access time selection heuristic of Section 5.5 it follows that sometimes it may be more favourable to access a cell at a later access time. Because depth-first trees are very unbalanced (the size of the subtree at the first child is expected to be much bigger than the size of the subtree of the last child) the traversal distance between the first two visits is bigger than for breadth-first trees, and the heuristic has a bigger impact on the interval length distribution (i.e. choosing the second access time option instead of the first makes a bigger difference compared to the breadth-first tree). The pruned variant forgoes the improvements gained by the selection heuristic by always preserving the first visit of a cell and, despite the expected improvements of pruning, results in a less favourable interval length distribution.

In addition to the effect of the traversal order we also examine the effect of the point set size on the interval length distribution for the same traversal order. Figure 10 shows the distribution for BF_P traversals for different point set sizes. In the sampled range there are only minuscule differences – the maximum length interval becomes bigger, but percentage wise the distribution does not change. Although only the plot for DF_P is shown, but we have verified this holds for all four traversal orders. We cannot measure the distribution for even bigger subdivisions, but at over two million intervals these results are highly unlikely to be a fluke, and instead represent a steady

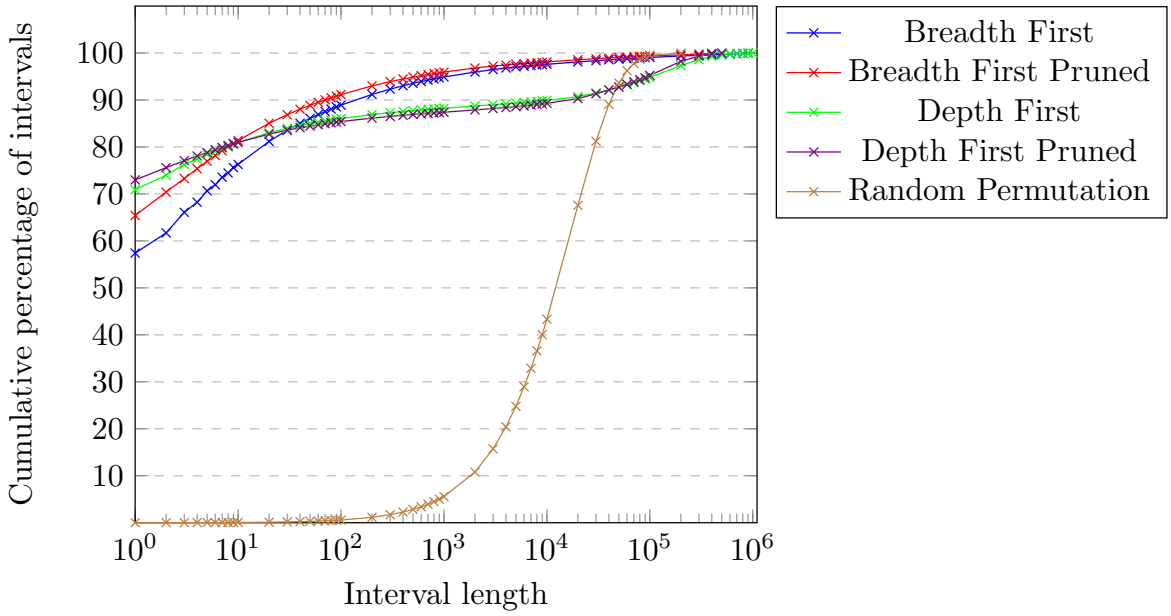


Figure 9: Interval length distribution resulting from various traversal orders for $n = 75,000$.

average. Hence we conclude that the distributions we found are a property of the traversal order, and are not dependent on the size of the subdivision. Therefore the traversal orders maintain their locality properties and scale well to larger subdivisions.

Clearly the spanning tree based traversals result in good locality. However, the target is to obtain a good graph coloring and the distribution of the intervals over the traversal length (i.e. $|C|$ or $2|C| - 1$) also plays a role. The graph coloring results will be presented in Sections 6.3 and 6.4.

6.3 Overlap coloring

The overlap graph of the interval set is colored by the heuristic coloring algorithm of Section 5.3.4. In Figure 11 the resulting number of stacks is shown for various input sizes. The results are consistent with our presumption that smaller intervals lead to less stacks (i.e. colors). Judging from the experimental results the relation between the input size of the point set and the total number of stacks looks to be linear for DF and DF_P and sublinear for BF and BF_P, although measurements of bigger input sizes are needed to confirm this conjecture about the asymptotic relation.

The BF and BF_P traversal order clearly result in the least amount of stacks. Since the difference in the number of stacks between BF and BF_P is negligible the BF_P traversal order has the best results because it does not require the additional data structure for passing on partial contribution (since it is a permutation).

The effectiveness of the system of stacks approach also depends on the distribution of the intervals over the stacks. Based on the coloring heuristic, which is biased towards assigning low color number, we expect that every additional stack accommodates less and less intervals. This is reflected in the plot of Figure 12, which shows the cumulative percentage of interval accommodated as a function of the number of stacks. For $n = 50,000$ the first stack of BF_P accommodates $76,8\%$ of all intervals. The remaining stacks accommodate $> 1\%$ each up to stack 6 (inclusive), $> 0.1\%$

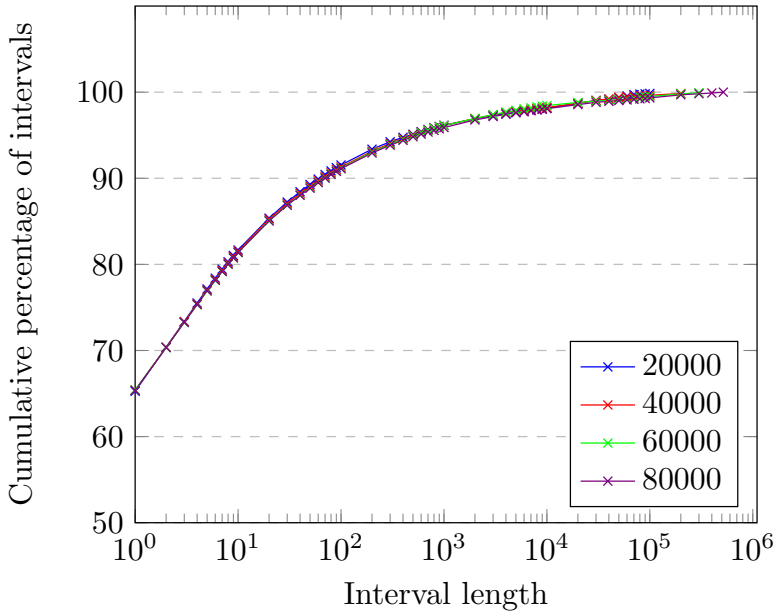


Figure 10: Interval length distribution of the BF_P traversal order for various point set sizes.

each up to stack 27 (inclusive), and from stack 85 onwards each stack accommodates less than 0.01% of all intervals.

In addition to the raw number of intervals per stacks the maximum load of the stacks (i.e. the maximum number of elements that simultaneously reside on it during execution) is also of importance. If they only hold a few elements their locality properties barely play a role. Even though for $n = 50,000$ the first stack of BF_P accommodates almost one million intervals ($76,8\%$ of all intervals) its maximum load is only 53. This is because all length-one intervals can trivially be assigned to stack one (of which there are a lot), and most other small intervals will be assigned to it as well due to the bias in the heuristic. The biggest maximum load among all 554 stacks is 77, followed by 69. The average is 26.6.

We noticed that the difference between the number of intervals accommodated and the maximum load is smaller for higher stack numbers. This makes sense because following from the heuristic's bias it is expected that stacks with a lower (resp. higher) stack number serve more small (resp. large) intervals and the length distribution is not uniform. However, for stacks 325 through 554 (41% of all stacks) this difference is zero. This means there is no oscillation in their load – it is increasing until the maximum load is reached, after which it is decreasing. We note that they together accommodate only 4212 intervals ($3.26 \cdot 10^{-3}\%$).

Our interpretation is that these are the longest intervals, i.e. those whose lengths are a big fraction of the total traversal length. For all stacks their top interval is the smallest of those that reside on it and, due to the coloring bias, this top interval is also continuously kept small for lower stack numbers. It is therefore unlikely that a long interval can be assigned a low stack number since it will overlap with the top intervals of most stacks, maybe even all of them in which case it would require the creation of a new stack. This causes an effect where intervals up to a certain size can be accommodated by the first so many stacks and, conversely, longer intervals most likely are assigned a stack number beyond that. Apparently this results in an implicit threshold whereby, from a certain stack number onwards, all intervals which reach this stack number are so long that it never occurs that two disjunct ones are contained by the top interval of those stacks. Hence their load never fluctuates. In our experiment this threshold effect kicks in from stack number 325

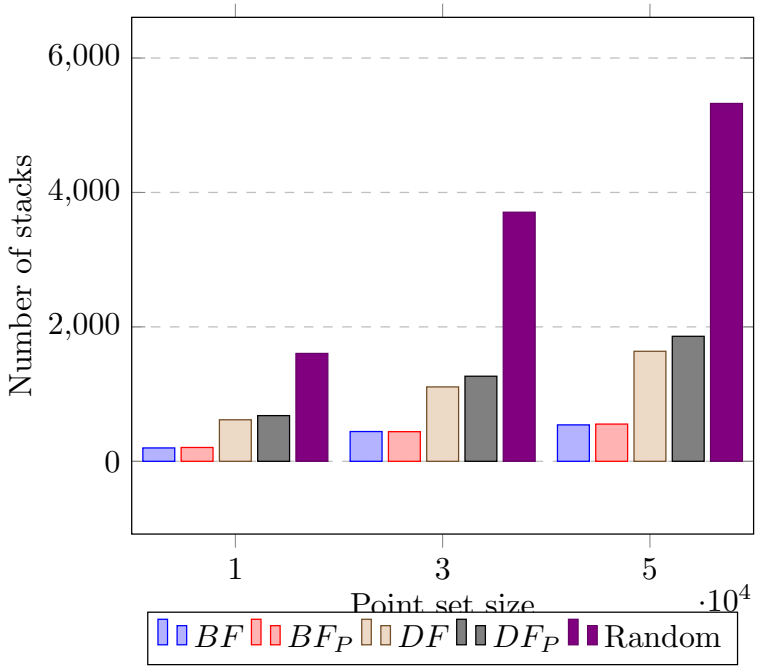


Figure 11: The number of colors used by the heuristic coloring algorithm for overlap graphs of various point set sizes.

onward.

6.4 Interval coloring

The interval graph of the interval set is colored by the optimal $k-$coloring algorithm by Carlisle et al. as discussed in Section 5.4.2. The results of the interval graph coloring are easier to interpret since the only output is the percentage of intervals colored. In Table 3 this percentage is shown for the different traversal orders and various k for a subdivision of a point set of $n = 75,000$. Table 4 shows the scalability of the approach by presenting the percentages for BF_P traversal orders for various point set sizes.

Analogously to the results of the interval distributions and the overlap coloring the breadth first orders outperform the depth first orders, and the difference between BF and BF_P is negligible for realistic k-values – the small k-values are included to show the relation between k and the percentage for the whole range. Therefore the BF_P traversal order has the best results because it does not require the additional data structure for passing on partial contribution (since it is a permutation).

It is remarkable that with only ten slots of cache memory over 76% of all vertex accesses can be accommodated in a subdivision of $80,000$ points, and with 100 slots this increases to over 95%. The results in Table 4 indicate that these percentages are only marginally influenced by the size of the subdivision. This implies that, up to at least input sizes of $n = 80,000$, over 95% of all vertex accesses through the intermediate data structure result in a cache-hit if the cache memory only has room for 100 slots. For $k \geq n/10$ over 99% of all intervals can be colored. The vertices corresponding to uncolored intervals incur writing latency when being written to the external memory data structure, and a cache miss penalty when they are loaded again. Depending on the implementation of this data structure these write and load operations may or may not have spatial

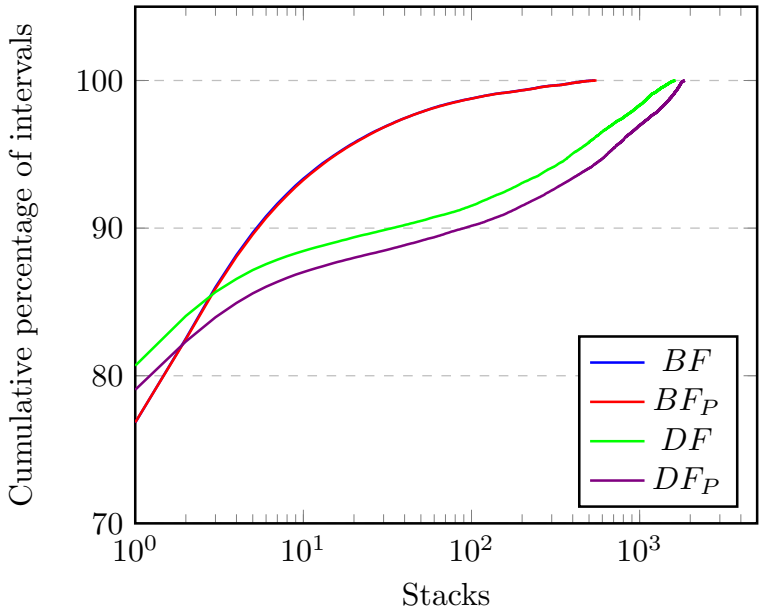


Figure 12: The cumulative percentage of intervals assigned to the first x stacks of all traversal orders for $n = 50,000$. Note that the graphs for BF and BF_P are almost identical.

locality.

Based on the high percentages achievable with only a couple of dozen slots an obvious optimization is to color the graph in two stages: We first color with k_1 colors where k_1 matches the L1 cache size, and then color the remainder with k_2 colors where k_2 matches the L2 cache size. The overall bottleneck will still be the slowest memory involved (i.e. RAM), but this strategy will still increase the number of L1 cache hits. It is unknown if the k_1-coloring followed by the k_2 coloring yields the same results as the $(k_1 + k_2)-$coloring, i.e. if optimality is retained, but since ten slots already color $> 70\%$ the difference in L1 hits is definitely worth the relatively negligible decrease in L2 hits.

6.5 Hardware specifications

We have now progressed to the point where the hardware specifications come in to play to make a comparison between the cache efficiency of the two approaches, i.e. stacks versus slots. We will limit the discussion to the BF_P traversal order since it has proven to yield the best results for both approaches. The typical hardware specifications as of 2012 are given by Patterson et al. [22, Figure 5.35] and are shown in Table 5. The access latency of a L1 cache hit is 1 or 2 cycles, i.e. instant.

Following the example in the model of computation a vertex can have a vector field with velocity in d dimensions and pressure as associated variables. This means that it requires four double precision variables worth of memory, i.e. 64 bytes. Since we only consider the DF_P traversal order we do not have to to pass on partial contribution, but we do need the accumulating variables which sum up the received contributions at each cell. There is one for each vertex variable so this would make the total memory footprint of a vertex 128 bytes or, in general, the memory requirement is two times the number of variables times eight bytes (double precision is always required). A total of four variables is not unusual for finite element methods. This means that the number of vertices per block (i.e. B) is one or two and that the gains by spatial locality are almost or completely nullified. The cache's vertex capacity ranges thus are $2,500 - 50,000$ for L2

Traversal	Number of colors (k)										
order	**10**	**25**	**50**	**100**	**250**	**500**	**1,000**	**2,000**	**4,000**	**8,000**	**16,000**
BF	78,33%	89,35%	93,08%	95,29%	97,06%	97,88%	98,48%	99,04%	99,43%	99,71%	100,00%
BF_P	76,54%	88,68%	92,86%	95,22%	97,04%	97,87%	98,48%	99,04%	99,43%	99,71%	100,00%
DF	78,93%	86,20%	87,90%	88,80%	89,73%	90,50%	91,41%	92,57%	93,99%	95,80%	97,78%
DF_P	75,42%	83,65%	86,15%	87,26%	88,17%	88,82%	89,61%	90,58%	91,87%	93,48%	95,36%
Random	1,39%	2,39%	3,50%	5,09%	8,19%	11,65%	16,45%	23,06%	32,05%	44,08%	59,65%

Table 3: Percentage of intervals colored by the k-coloring algorithm for various k and traversal orders for the subdivision of a point set of $n = 75,000$.

Point set	Number of colors (k)										
size (n)	**10**	**25**	**50**	**100**	**250**	**500**	**1,000**	**2,000**	**4,000**	**8,000**	**16,000**
10,000	76,77%	89,33%	93,57%	95,71%	97,45%	98,37%	99,04%	99,56%	100,00%	100,00%	100,00%
20,000	76,70%	89,06%	93,27%	95,51%	97,19%	97,94%	98,65%	99,23%	99,72%	100,00%	100,00%
30,000	76,68%	88,94%	93,10%	95,35%	97,18%	97,99%	98,56%	99,15%	99,75%	100,00%	100,00%
40,000	76,66%	88,87%	93,03%	95,36%	97,20%	98,09%	98,70%	99,25%	99,68%	100,00%	100,00%
50,000	76,56%	88,74%	92,92%	95,26%	97,14%	98,07%	98,59%	99,07%	99,53%	99,90%	100,00%
60,000	76,48%	88,67%	92,86%	95,27%	97,19%	98,17%	98,80%	99,22%	99,64%	99,94%	100,00%
70,000	76,56%	88,67%	92,84%	95,13%	96,90%	97,76%	98,40%	98,90%	99,37%	99,84%	100,00%
80,000	76,53%	88,67%	92,79%	95,15%	96,98%	97,84%	98,46%	99,02%	99,45%	99,79%	100,00%

Table 4: Percentage of intervals colored by the k-coloring algorithm for various k and input sizes for the BF_P traversal order.

	Hardware level		
Feature	**L1 cache**	**L2 cache**	**Paged memory**
Total size in blocks	$250 - 2000$	$2,500 - 25,000$	$16,000 - 250,000$
Total size in kilobytes	$16 - 64$	$125 - 2000$	$1,000,000 - 1,000,000,000$
Block size in bytes	$16 - 64$	$64 - 128$	$4000 - 64,000$
Miss penalty in CPU clocks	$10 - 25$	$100 - 1000$	$10,000,000 - 100,000,000$

Table 5: Typical cache features as of 2012 [22, Figure 5.35].

and $250 - 4,000$ for L1 respectively, assuming that the full cache is available. In practice there is also some object overhead (e.g. pointer data) and the CPU also serves other processes which also require cache capacity (e.g. the operating system).

This has strong implications for the stacks approach because we intended to make the most of their cache-oblivious nature – the excess data elements loaded by a block I/O-operation always loads exactly those data elements which are needed next. But since $B = 1$ or 2 this has little to no advantages over the naive approach of storing them in an array and using random access, i.e. by design the stack data structure does not improve the I/O-efficiency and which vertices remain cached is solely determined by the cache's replacement policy (i.e. least recently used). Therefore the interval coloring, which is provable optimal in its selection of which vertices to store in cache memory and exercises complete control over the replacement policy, can only improve on it. The small number of elements per block also immediately resolves the discussion of the uncolored interval data structure: The write penalty of the array approach is not excessive since a vertex fills a block almost completely, and (cache-efficient) priority queues provide no spatial locality and only introduce overhead.

In practice things are slightly different due to prefetching techniques which can be employed to speed up load operations. We distinguish two types of prefetches: In software and in hardware [23, Section 3.7].

Software prefetch instructions can be implemented into the software and give hints to the cache which block of data should be loaded into cache memory, anticipating future use. This operation works just like a load from memory operation, but since the memory request can be send before the CPU realizes that the data is required the CPU does not have to wait for it when it does. It is the programmer's responsibility to issue the prefetch instruction in a timely manner. By prefetching data from potential search paths in comparison-based searching data structures Khuong et al. [24] show that data structures which better suit prefetch instructions may outperform data structures which are more cache-friendly in the external-memory model of computation when there is an excess of bandwidth between RAM and cache memory.

Hardware prefetch instruction are issued by the CPU, which is able to detect simple access patterns like sequential reads of an array and will load blocks into the cache hierarchy even before they are accessed. Due to hardware reasons[5], reading the next block(s) immediately afterwards is faster than accessing a single random block at only 2 clock cycles of 400 MHz DDR2 SDRAM per block [25, Table 3.1]. A modern CPU runs at 3 GHz [22, Figure 1.17], so this brings down the latency to 15 CPU cycles per subsequent block and effectively expands the effects of spatial locality from intra-block to inter-block (but only for consecutive operations on nearby memory locations).

By issuing prefetch instructions the CPU can be fed a continuous stream of non-cached data. This affects the main data stream, cell matrix stream, as well as the vertex input stream (i.e. they also affect the slots approach). Since stacks occupy contiguous memory prefetching is also applicable to them.

As a final step we need to assert that the latency of the computational work does not exceed the load latency (in cycles), otherwise faster loading begets no benefits. In a tetrahedralization where each cell has four adjacent vertices the matrix-vector multiplication $A_c\mathbf{v_c}$ consists of 12 additions and 16 multiplications. The latency of an addition resp. multiplication is 3 resp. 5 cycles , but due to pipelining[6] the throughput is one addition/multiplication per cycle after the

[5] As to not introduce the full hardware structure of RAM memory we refrain ourselves from the technical foundation of this effect – it is related to certain parts retaining their charge, resulting in non-uniform access latencies. We refer to Akesson et al. [25, Chapter 3] and Shao [26, Section 2.1] for an overview of the RAM memory structure and the timing constraints. In jargon terminology: it is related to the delay between Column Access Strobe (CAS) operations being smaller for accesses to the same row of memory).

[6] Instructions take several cycles to complete in which the instruction is fetched, decoded, an affective address

first one [27, Section 8.9]. Therefore the total computational latency per cell is at most 5+27 cycles. The data consists of four vertices of 8 double precision variables (4×64 bytes) each and a 4×4 cell matrix of 16 double precision variables (128 bytes). The cell matrix may be efficiently prefetched from the input stream in 15-30 cycles and does not form the bottleneck. However, the vertices are not guaranteed to cache-hit and if even one of them has a cache miss the $100 - 1000$ cycles miss penalty definitely exceeds the computation latency. Thus the cache-hit ratio of vertex variables forms the bottleneck of the algorithm and improving it directly affects the running time.

We we note that the maximum (theoretical) peak transfer rate from RAM to cache memory can also form a bottleneck. The machine used by Khuong et al. could transfer a maximum of 8 bytes/cycle. Therefore the cell matrix will take at least $128/8 = 16$ cycles to load, regardless of prefetching.

6.6 Discussion

Thus far we have refrained from making a comparison between the two approaches. This is also not directly possible because, in contrast to the slots approach, the cache-miss ratio of the stacks approach is not immediately apparent. However, by putting things into context we are able to make a decision.

The slots approach exercises explicit control over k slots of the cache memory and can easily be extended to also include L1 cache efficiency, but is still subject to the constant factors of overhead that result from the ideal cache assumption. Furthermore, the exact choice of k is hard to make because it requires an estimation of the CPU and cache usage of other processes. Only the upperbounds can be directly derived from the hardware specification. However, even for small k a high fraction of all intervals can be colored. It should also be noted that prefetching techniques make loading vertices from the uncolored interval array just as fast as loading data from any of the streams.

The stacks approach has cache-oblivious data structures (i.e. also L1 efficiency by design) and benefits from prefetching, but the replacement policy is non-optimal and decided by the cache. Our experiments indicate the number of stacks is not constant, but instead linear in the subdivision's size and already reaches into the hundreds for subdivisions of only $335,000$ cells. With such a large number of stacks they have lost their main asset: The cache's least recently used replacement policy matching our intended use of the data because elements which have not been recently used will also not be used in the near future (since they are buried deep in the stack). Moreover, the cache's blocks are spread thin over the stacks and due to $B = 1$ less consecutive loads are needed to trigger a cache-miss by hitting the uncached data below. When uncached data is hit it can be loaded faster with prefetch instructions, but since we have no guarantee that the next elements in line will be used in the near future this may be wasteful. Optimizing when (not) to prefetch directly brings us back to the slots approach. In summary the stacks approach stands or falls with a bounded number of stacks. Since this condition is not met the interval coloring approach makes better use of the cache and deals better with the inevitable misses (i.e., prefetching from the uncolored array is faster than unstructured single block loads from different stacks). If it would be met then stacks would have been the better choice due to their oblivious nature and ease of implementation.

The remainder of this section is dedicated towards estimating the effectiveness of the interval coloring approach in terms of the number of cache-misses by looking at the lower bound of all parameters.

Assuming $B = 1$ and a L1 cache of 250 blocks of which only 20% are available to us, i.e. $k = 50$,

is read (if necessary) and finally executed. During a cycle each step is executed for a different instruction, i.e. instruction i_4 is fetched while i_3 is decoded, while an address is read for i_2, while i_1 is executed. At the end of the cycle each instruction moves one step forward in the pipeline. Therefore the latency is the length of the pipeline, but the throughput is one operation per cycle.

the coloring of the $n = 80,000$ subdivision still results in a 92,79% L1 cache-hit rate for vertex accesses. Analogously we find that a L2 cache of 2,500 blocks of which only 20% are available, i.e. $k = 500$, results in a vertex cache-hit rate of 97,84%. We had already established that these percentages scale well with the subdivision size. Since the interval coloring percentage has a one-to-one correspondence to the vertex cache-hit rate we therefore conclude that the result are very strong indeed – the various heuristics have actively optimized the cache-ratio of the vertices. The asymptotic complexity of the pre-processing step is dominated by the sorting of the intervals, i.e. $O(|\mathcal{I}| \log |\mathcal{I}|)$ where in 3D $|\mathcal{I}|$ is approximately 26 times the size of the point set of the subdivision.

Due to the large number of open ends and optimization problems these topics have a separate discussion in Section 7.

7 Optimizations and future work

We have experimented with various algorithms and heuristics to find the data access order which is most favourable for obtaining cache efficiency. The experiments had an exploratory nature with the aim to identify the usefulness of simple heuristics for the global optimization problem presented at the end of Section 5.6. To improve the results we put forth the following ideas as (pointers to) areas of interest for further optimization.

Traversal orders

1. In this thesis we have explored breadth- and depth- first spanning trees, but there are many more. Are there other spanning trees with more favourable properties (pruned or not pruned)?
2. In our experimental setup the choice of the root and the order in which children of a cell are visited were kept constant at an arbitrary value/order. Could we benefit from choosing the starting cell in a particular way?
3. Our definition of a treewalk has been a depth-first traversal of the spanning tree. How good are (pruned) breadth-first treewalks of breadth- and depth-first spanning trees?
4. Can the use of a traversal orders which is not based on spanning tree heuristics be argued? A geometric separator recursively make balanced bisections of the subdivision. An in order traversal of the leafs of its recursion tree (i.e. single cell subdivisions) seems to have good locality properties.
5. Our pruning heuristic preserves the first occurrence of each cell in the treewalk. Could we benefit from choosing the which occurrence is preserved in a particular way?
   - For depth-first trees preserving the last occurrence looks promising. The argument for why DF outperformed DF_P may be completely inverted – the unbalanced tree property can work in our favor.
   - Before pruning the access time choice heuristic of Section 5.5 could be used to determine which option looks like the best choice to retain by means of majority vote. That is, if a cell is accessed more than once during the treewalk then, after applying the heuristic, the access time at which the most vertices are accessed is preserved during pruning (rather than always the first).

Intervals

6. The data access time selection heuristic only focusses on minimizing the average interval length by minimizing the interval $[t_1, t_k]$. The choice of $t_2, \ldots, t_{k-1}$ does not matter in this context, but it can make a difference for the interval length distribution. For example you can have two intervals of length 50, or one of 3 and one of 97. Could we benefit from choosing $t_2, \ldots, t_{k-1}$ in a particular way without using combinatorial time (i.e. considering all possibilities)?
7. Our heuristics are aimed at obtaining as many disjunct intervals as possible by minimizing their length. However, the stacks approach would also benefit from intervals which contain each other. Can different optimization heuristics be formulated which also take this secondary goal into account?

Implementation details

8. The data in the 'main data stream' as described in Section 5.7 only contains addresses (i.e. stack or slot ID's) of where variables should be loaded from and/or stored to. This is actually redundant – since they are static the compiled program could already

contain the stack/slot addresses in its load/store instruction, thus removing the need for the stream. The CPU has a dedicated independent cache for instructions so this causes no conflicts. This optimization is possible because, unlike the space-filling curve approach, we're dealing with a static subdivision.

9. Whenever a bit of information is required it might be faster to dedicate a full byte to that value instead. The time saved by loading less bits may be exceeded by the time it takes to filter the individual bits from a byte.
10. Due to the invariant that the number of vertices loaded from the input stack is always smaller than the number written to the output stack the combined size of the input and output stack is always smaller than $|V|$. They can therefore be implemented together in a size $|V|$ array.
11. (Knowledge of RAM design memory required:) By storing each stream and stack on a different bank of SDRAM memory read and write operations will never cause row conflicts and the memory latency will be smaller. If this optimization is incorporate it voids the suggestion in bullet 10.

Miscellaneous

12. The abstract data access model completely describes all data accesses during an iteration. Therefore every off-line optimization technique in the book for minimizing memory latency can be used as long as it does not make the pre-processing time excessive (e.g. RAM address mapping techniques [26]).
13. When running the algorithm in parallel, the abstract data access model may be used for weight balancing purposes by assigning a uniform fraction of the traversal length to each processor.
14. The 3D approach as such has no dependencies on the dimension of the subdivision – All we need are the vertex accesses times along a traversal order. Therefore it can be directly applied to subdivisions in other dimensions. How effective is the interval (overlap) graph coloring approach for other dimensions? Are the same heuristics still the best heuristics?
15. The 2D approach works efficiently with one intermediate cell/vertex stack, but still incurs misses when loading from these structures. It would be of interest to compare the actual number of misses with the 2D slots approach using the BF_P traversal order for the same subdivision.
16. How does the percentage of colored data elements change if we incorporate edge and/or facet variables into our subdivision?
17. We have not explored the variant of the finite element method with dynamic subdivisions. This poses additional constraints on the approach. In 2D dynamic operations seem to be possible as long as the spanning tree can be dynamically adjusted as well. In 3D things are more complicated because updates are not local. The load/store addresses depend on the interval graph coloring algorithm which works off-line on the set of all intervals, and the result is stored spread out over a stream. Moreover, even just maintaining the set of all intervals with acceptable overhead is challenging. A single insertion/deletion can trigger an update of the start/end times of $\Omega(|\mathcal{I}|)$ intervals.

8 Conclusion

In this thesis we set out to generalize the Stack & Stream algorithm for finite element method computations to arbitrary 2D and 3D subdivisions. The Stack & Stream algorithm is designed to minimize the overall memory latency that is incurred during executing, which is one of the most serious bottlenecks in high performance computing. In 2D we have been successful in this attempt and have developed an I/O-efficient algorithm for arbitrary subdivisions at the cost of doubling the traversal length.

The 3D variant posed a harder challenge and no optimal solution was found. We have introduced the abstract data access model which can be used to capture the data access pattern of any finite element method computation regardless of its traversal order, access pattern or the subdivision's dimension. In the context of this model two approaches for I/O-efficient computations were defined based on the overlap graph and interval graph coloring. Both approaches are subject to the effectiveness of heuristic solutions to combinatorial problems, e.g. the traversal order and access time choices.

We have implemented both approaches and the experimental results showed that the interval graph coloring for the BF_P traversal order yields the best results. Over 92% (resp. 97%) of all vertex data accesses are a cache-hit in L1 (resp. L2) cache memory for tetrahedral subdivisions of point sets up to $80,000$. All other data accesses may be efficiently prefetched from data streams. The experiments also indicate that these percentages are only marginally influenced by the size of the subdivision and scale well to bigger subdivisions. Considering that we do not exploit any particular property of the structure of such big input subdivisions, these high percentages are a powerful result indeed.

The many heuristics and variables underlying our approach provide many angles for further optimization. Before any of them are considered it should be established that the results presented in this thesis, which are conjectured to be almost independent of the subdivision's size, indeed scale well to even bigger subdivisions and predict actual performance on actual machines well. Thereafter the effectiveness of the many listed (traversal order) alternatives may be explored. The main open question is whether it is possible to bound the number of colors of the interval (overlap) coloring by a constant. Without making any assumptions about the subdivision this seems like an inherently hard problem due to the combinatorial nature of the many variables involved.

References

[1] Mark S Gockenbach. *Understanding and implementing the finite element method*. Siam, 2006.

[2] Marc Alexander Schweitzer. *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*, volume 29 of *Lecture Notes in Computational Science and Engineering*. Springer, 2003.

[3] Frank Günther, Miriam Mehl, Markus Pögl, and Christoph Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[5] Michael Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer, 2013.

[6] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002.

[7] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[8] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[9] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[10] Overlap Graphs. Http://graphclasses.org/classes/gc_913.html. Accessed September 2015.

[11] Joseph C Culberson and Feng Luo. Exploring the k-colorable landscape with iterated greedy. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge*, 26:245–284, 1996.

[12] Rex A Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6(1):343–367, 1991.

[13] Interval Overlap Graphs. Http://graphclasses.org/classes/gc_234.html. Accessed September 2015.

[14] Martin C Carlisle and Errol L Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59(3):225–235, 1995.

[15] Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing (STOC)*, pages 246–251. ACM, 1983.

[16] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[17] Gerth Stølting Brodal. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 150–163. Springer, 2013.

[18] Lars Arge, Michael A Bender, Erik D Demaine, Bryan Holland-Minkley, and J Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[19] Michael J Fischer and Michael S Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM (JACM)*, 41(1):3–30, 1994.

[20] Amr Elmasry, Arash Farzan, and John Iacono. A priority queue with the time-finger property. *Journal of Discrete Algorithms*, 16:206–212, 2012.

[21] http://nl.mathworks.com/help/matlab/ref/delaunaytriangulation-class.html. Accessed September 2015.

[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (5th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2013.

[23] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2015.

[24] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *CoRR*, abs/1509.05053, 2015.

[25] Benny Akesson and Kees Goossens. *Memory Controllers for Real-Time Embedded Systems*. Springer, 2011.

[26] Jun Shao. *Reducing main memory access latency through SDRAM address mapping techniques and access reordering mechanisms*. PhD thesis, Department of Electrical and Computer Engineering, Michigan Technological University, 2006.

[27] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs – An optimization guide for assembly programmers and compiler makers*. Technical University of Denmark, Augustus 2014.