Generation and Optimisation of Code using Coxeter Lattice Paths

Thomas J. Ashby
Institute for Computer Systems Architecture,
University of Edinburgh,
Scotland, UK
Email: T.Ashby@ed.ac.uk

Anthony D. Kennedy
SUPA, School of Physics,
University of Edinburgh,
Scotland, UK
Email: adk@ph.ed.ac.uk

Stephen M. Watt
Computer Science Department,
The University of Western Ontario,
London, Canada
Email: Stephen.Watt@uwo.ca

Abstract

Supercomputing applications usually involve the repeated parallel application of discretized differential operators. Difficulties arise with higher-order discretizations their communications can overlap processors in complex ways. Their correct and efficient implementation requires careful choreography of computation and communication, taking into account the symmetries of the problem and of the computer's communication network. This paper shows how these symmetries can be used to automate the construction of the code for optimized operator computation. This is done with considerable generality by making the symmetries both of the problem and the computer explicit using the language of finitely presented reflection (Coxeter) groups, and using coset enumeration to generate and optimize the required code.

I. Introduction

Many scientific computations require the frequent application of a discretised differential operator, such as a Laplacian or Dirac operator, and there is a trade-off between the order of the discretization scheme and the magnitude of the grid spacing. Naïvely it is advantageous to use higher-order discretization schemes because the discretization errors fall as a higher power of the grid spacing, but sometimes the cost of doing so outweighs the benefits. This may be because the system becomes less stable, or because the errors are limited by the availability of initial or boundary-value data. There are also situations, however, where higher order schemes are considered *too hard to program correctly and efficiently*. This is the problem we address.

Continuous translational and rotational symmetries of computational problems are necessarily broken by discretization, and this breaking is characterised by the difference between the true differential operator and its discretised form. This symmetry breaking operator is "irrelevant" in the sense that it vanishes as a suitably high power of the grid spacing, and becomes negligible for sufficiently small values thereof. Significant efforts are often made in the choice of discretization scheme and grid symmetries so as to minimise the number of operators that have to be adjusted in order to remove the grid artefacts up to a given power of the grid spacing so the efficient and portable implementation of such complicated schemes is of practical importance. In addition, the spacing between neighbouring grid points is not a fixed physical distance in some systems, but is implemented in terms of some *gauge field*, allowing finer grids to be used where important without destroying the homogeneity of the formulation. This further complicates the programming problem.

In this article, we consider *stencils* as a generalised abstraction for discretised operators. A set of values are associated with sites on a regular grid or lattice, and a new value for each site is calculated based on the old values in a neighbourhood of that site, described by a *stencil* that is the same shape for each site. All calculations are independent and so all sites can be

updated in parallel. We term each such update a *global stencil application*. The use of gauge fields means that the path linking the neighbour to the centre, or *seed* of the stencil (i.e., the site to be updated) must be taken into account. More concretely, we consider stencils whose associated expression can be treated as a (generalised weighted) sum centred on the seed, with the stencil being a collection of paths connecting the seed to various neighbours. A path consists of a sequence of links between nearest neighbours on the lattice, and when a value traverses a link, it is transformed by the associated *link transformation*. The value at the end of each path must be transported along it to the seed, being transformed along the way, before it can be incorporated into the sum.

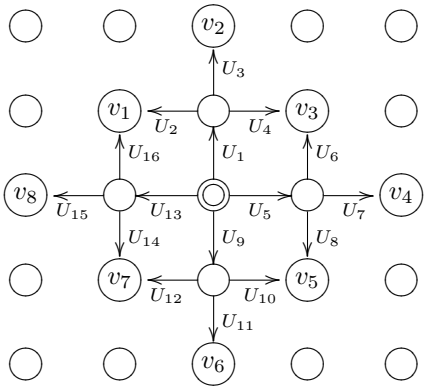
In this paper we describe a code generation framework to automate the generation and optimisation of numerical kernels, thereby eliminating a labourious part of writing and optimising scientific simulations. The foundation of our approach is group-theoretic, and has its roots in the explicit symmetries of the problem domain and the inter-processor communication network. Our formalism uses the correspondence between discrete reflections on Euclidean space and Coxeter groups to express elements of an arbitrary lattice and individual steps in paths on that lattice, and exploits the group structure to reason about sets of paths. We use the Todd–Coxeter algorithm to enumerate cosets of equivalent paths, giving a powerful yet simple approach to optimisation. Our protoype system can generate large complex stencils for our example problem domain and the optimisations give speedups of up to 2.7 over unoptimised code.

The rest of the paper proceeds as follows: Section II discusses related work, Section III gives the running example, Section IV introduces Coxeter groups and their relation to lattices, Section V describes how to recover a lattice from an affine Coxeter group and defines paths on the lattice, Section VI discusses group computations with finite lattices, and defines sublattices and how to generate and manipulate sets of paths, Section VII describes optimisation of paths, Section VIII covers some aspects of code generation, Section IX describes experimental results, and finally Section X concludes.

II. Related Work

Domain specific libraries (for example, the venerable BLAS) are a long-standing approach to optimisation for performance-critical numerical applications; highly optimised routines can be supplied by third parties and greatly benefit programs that map naturally on to the abstractions they provide. More recently, there has been considerable interest from the systems community in library generators. The first major advantage they offer is the means to cope with the difficulty in generating highly optimised code for a plethora of complex modern architectures [1]–[3]. Secondly they offer wider functionality, thereby blurring or crossing the line between libraries and domain specific languages (DSLs) [4]–[8]. Thirdly they can exploit domain specific knowledge using search over algorithmic degrees of freedom to find better performing (or even simply feasible) code [4], [5], [9]. In this paper, the primary issues are providing the foundation for the restricted DSL in question and establishing algorithmic optimisations, rather than machine tuning.

Stencils are a common idiom in scientific simulation and image processing. Consequently, various authors have considered how to express stencils cleanly using the functions of a domain specific library/language [6], [7]. Approaches to optimisation have included tiling data access to improve cache behaviour [10], removing redundant communication on parallel message



$$U_1U_2v_1 + U_1U_3v_2 + U_1U_4v_3 + U_5U_6v_3 + U_5U_7v_4 + U_5U_8v_5 + U_9U_{10}v_5 + U_9U_{11}v_6 + U_9U_{12}v_7 + U_{13}U_{14}v_7 + U_{13}U_{15}v_8 + U_{13}U_{16}v_1$$

Fig. 1. On the left is a simple stencil on a two-dimensional square lattice, with the seed point marked as a double ring. The stencil consists of twelve paths all of length two reaching eight different endpoints. The corresponding expression (a sum) is given on the right, and consists of the eight values at the end points $v_1 \dots v_8$ and sixteen link transformations $U_1 \dots U_{16}$. For a global stencil application, such an expression would be computed for the stencil centred on each point on the lattice.

passing systems [11], and fusing together the stencil operation with other subsequent operations [6], [7], or with subsequent applications of the same stencil using skewing to reorder loop iterations. Some combine several of these [12], [13] or target algorithms with specific uses of stencils [14]. These previous works have been restricted to scalar values at lattice sites, do not accommodate link transformations and are restricted to square 2D lattices (with the exception of [10], which considers a 3D case) rather than the general n dimensional case with different lattice types. Furthermore, very few of the previous works explicitly deal with communication between compute nodes.

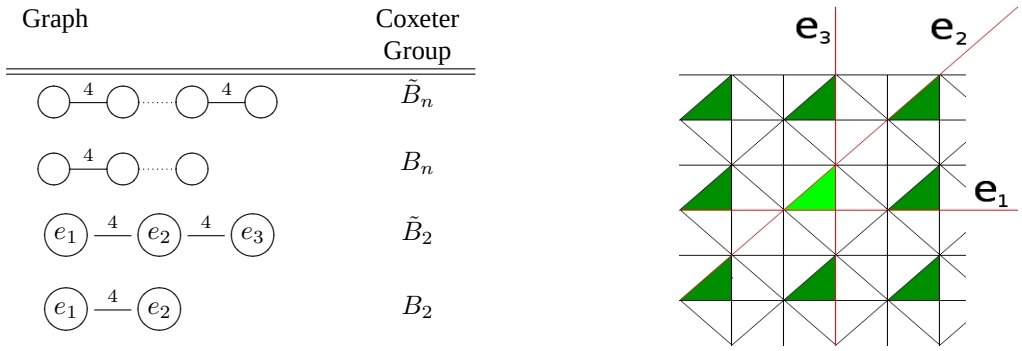
Our prototype is probably most similar to SPIRAL [5] in that it constitutes a small DSL to generate a family of numerical kernels, uses algorithmic knowledge to optimise, and is also based on a symbolic computer algebra system (SPIRAL is based on GAP [15], our work is based on Aldor [16]). There are various computer algebra packages related to Coxeter groups (e.g., [17], or functionality in [15], [18]), but none have been used for code generation. The SPIRAL group is also currently working on group symmetries, although with application to optimisation rather than as a foundation for their DSL [19].

III. Example

The running example used in this paper associates 3×2 matrices to lattice sites and 3×3 matrices to links. The link transformation is matrix-matrix multiplication. The stencil expression is addition of all the values once they have been transported to the seed. The link transformations are linear, and the addition of site values at the seed[1] distributes over the link transformations. This example is a simplified version of the structure found in an important class of lattice QCD computations.

We use a two dimensional example (Fig. 1) as the formalism is clear, but benefits are less obvious; out technique is general and applies directly to higher dimension and arbitrary regular lattice shapes where the complexity of constructing operators by hand is much more daunting.

[1]Site values are taken from a vector space that provides an addition on its elements.



(a) Graphs of Coxeter Groups

(b) Reflections corresponding to group elements

Fig. 2. On the left is a table of graphs for Coxeter groups, showing the general form of the hypercubic group and its finite subgroup ($\tilde{B}_n, B_n$), and the graphs that correspond to the two-dimensional case used as the running example ($\tilde{B}_2, B_2$). In the latter two graphs the generators have been labelled to correspond to the graphic on the right, which shows each group element of $\tilde{B}_2$ (and B_2) as a reflection in the plane. The fundamental region (bounded by the generating reflections e_1, e_2 and e_3) and its translations have also been highlighted to show the correspondence between the translations and the points on a two-dimensional lattice. The fundamental region is the smallest region of Euclidean space such that it, together with all of its images under the action of the full Coxeter group, covers the space.

IV. Coxeter Groups and Lattices

Although in the context of lattice QCD we are chiefly interested in hypercubic lattices, using Coxeter groups automatically provides the generality to cope with arbitrary lattice types in any number of dimensions. Coxeter groups are finitely presented groups that can be represented by graphs showing the relations between generators [20]. Families of groups with common structure are denoted by a letter, a subscript giving the dimension of the space, and the addition of a tilde for an affine family related to the finite family with the same letter. For example, $\tilde{B}_2$ is the two-dimensional hypercubic (affine) group, B_2 is the associated finite group and $\tilde{B}_n$ is the general hypercubic group.

The sites of a discrete regular lattice in Euclidean space of dimension n (where the lattice and space have equal dimension) can be treated as translations of a single point at the origin. Any translation is the product of two reflections in parallel planes, so the translation group for the lattice is a subgroup of a finitely generated reflection group. As all rotational symmetries are also the product of two (non-parallel) reflections, such a reflection group is the whole symmetry group of the lattice. The Coxeter groups, which have been completely classified, are in one-to-one correspondence with the reflection groups on n dimensional Euclidean space. Consequently, any discrete regular lattice can be built from the corresponding affine Coxeter group. By showing how to build lattices and paths from these groups we therefore cover all possible lattice types in any number of dimensions.

V. Lattices and Lattice Paths

A. Lattices

As part of the classification process for Coxeter groups [21], Coxeter showed that removing a generator from a graph for any affine group must reduce the group to being finite, and that this can be done by removing any generator. In addition, it is always possible to remove a generator and keep the resulting graph connected – i.e., the corresponding finite subgroup is irreducible.

TABLE I

ACTING ON THE FIRST TRANSLATION WITH THE ELEMENTS OF THE EVEN SUBGROUP OF B_2 TO GIVE THE GENERATORS OF THE TRANSLATION SUBGROUP OF $\tilde{B}_2$ AND THEIR INVERSES. EACH RESULTING ELEMENT IS LABELLED WITH THE CORRESPONDING TRANSLATION DIRECTION AS IT OCCURS IN FIG. 2. NOTE THAT THE LAST TWO GROUP ELEMENTS ARE NOT THOSE THAT RESULT FROM SIMPLE APPLICATION OF THE AUTOMORPHISMS, BUT SHORTER WORDS THAT GIVE AN EQUIVALENT TRANSFORMATION. THE EQUIVALENCE OF THESE WORDS IS DISCUSSED IN SECTION VI-A.

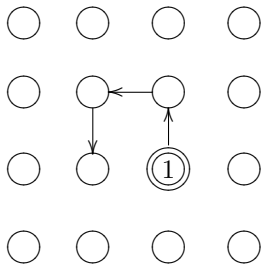
Transformation	Result	Direction in fig. 2
$1(e_2e_1e_2e_3)$	$e_2e_1e_2e_3$	$\leftarrow$
$e_1e_2(e_2e_1e_2e_3)$	$e_2e_3e_2e_1$	$\uparrow$
$e_2e_1(e_2e_1e_2e_3)$	$e_1e_2e_3e_2$	$\downarrow$
$e_1e_2e_1e_2(e_2e_1e_2e_3)$	$e_3e_2e_1e_2$	$\rightarrow$

The first step in generating a lattice from an affine Coxeter group is to choose one such generator. Figure 2(a) shows the Coxeter graph for the two dimensional hypercubic group and the graph of the finite subgroup that results from removing one generator (i.e., e_3). When starting from the finite group, the removed generator is an "extra" generator that makes the group infinite. Given that the finite group contains no translations but the infinite group does, the extra generator must be parallel[2] to one element of the finite group, i.e., the angle between them must be zero. All the other elements that are not parallel to the extra generator form a pair with it such that repeated products of the pair have some finite periodicity specified by the angle between the elements, so the parallel element can always be found by brute force. However, in certain cases there may be formulae to find it directly. For the general hypercubic group, the relationship defining the word in B_n parallel to the extra generator can be stated as $\langle R_n(R_{n-1}(\ldots R_3(R_2(e_1))\ldots)), e_{n+1}\rangle = 1$. The notation $\langle , \rangle$ indicates Euclidean inner product of the normals to the reflections, and $R_i(x)$ corresponds to the inner automorphism $e_i x e_i^{-1}$ and is used to make clear that group element x is being transformed by e_i. In the two-dimensional hypercubic case, the parallel element is $R_2(e_1) = e_2e_1e_2^{-1}$.

The extra generator and its parallel from the finite group give us a translation, one of the generators of the translation group corresponding to the lattice; call this the *first translation generator*, being $e_2e_1e_2e_3$ in the example. We derive the other translation generators by appealing to the correspondence between group elements and affine transformations. The inner automorphisms defined by the elements of the finite group correspond to orthogonal linear similarity transformations and so map translations to translations with the same length. Each inner automorphism thus maps the first translation generator to one of the generators of the translation group or its inverse.

Given that the $2n$ elements of the finite subgroup map the first translation generator to one of the n translation group generators (and their inverses), it is only necessary to deal with the even subgroup of the finite subgroup (i.e., the rotations) whose elements are easy to construct. By applying the automorphism corresponding to each element (except the identity) in the even subgroup of the finite subgroup of the affine Coxeter group to the first translation generator, we get the generators (and their inverses) of the translation subgroup and therefore the lattice. This is shown for the two-dimensional hypercubic case in Table I. Figure 2(b) shows the translations of the fundamental region that correspond to the translation subgroup. Each translated region corresponds to a lattice point.

[2]In the following discussion we identify group elements with their corresponding Euclidean reflection, thus giving sense to phrases such as "parallel group elements"



$$(1, [e_2e_3e_2e_1, e_2e_1e_2e_3, e_1e_2e_3e_2]) = (1, \uparrow \leftarrow \downarrow)$$

Fig. 3. Below, a lattice path, with above a diagram representing it. The path starts from the origin on the lattice, represented by the identity element. The step sequence is a word in the free group generated by the translation subgroup generators. The directions correspond to the generators given on the right of fig. 2

B. Lattice Paths

A *lattice path* consists of a starting point on the lattice s, which is an element of the translation subgroup T of the affine Coxeter group, and a finite sequence of steps (translations) p, each of which is represented by a generator (or inverse of a generator) of the translation subgroup. Each step in the sequence represents a translation from the current lattice point to the next point in the path, starting from s. The translations in a sequence are applied from left to right; an example of a lattice path is given in Figure 3. The set of such step sequences P can be treated as the free group generated by the translation subgroup generators $g_i \in T$, with lattice paths being members of the direct product $T \times P$. Any point reached along a lattice path can be found by taking the required number of steps, treating them as elements of the translation subgroup and multiplying them with the start point using the group operation of the affine Coxeter group.

For lattice QCD calculations we are only interested in paths that never double back – i.e., take a step in a given direction and immediately reverse that step. This is achieved automatically by treating step sequences as elements of a free group, as $gg^{-1} = 1$ for any free group generator g thus making paths with redundant steps equal to the same paths with the redundancies removed.

Paths that return to where they started (i.e., loops) are important in lattice QCD. A path can be checked to see if it is a loop in a similar way to finding points along the path. Elements of the step sequence are treated as elements of the affine Coxeter group and multiplied together; if the result is the identity then the path is a loop.

VI. Finite Lattices, Sublattices and Path Generation

A. Finite Lattices and the Word Problem

The *uniform word problem* is the name given to the task of determining whether two words a and b in the generators of a finitely presented group are in fact the same element under the group presentation. This problem has already been mentioned when discussing the shortest words representing the translation subgroup generators in Table I; to pick the shortest representation of a group element requires knowing when two words are equivalent. For a finite group this can be solved in theory by enumerating the group as cosets of the identity using the Todd–Coxeter algorithm [22], although in practice a

naïve approach may run out of machine resources or take too long; here we assume that computing with the finite groups in question is always feasible. The same question is known to be undecidable for some infinite groups. This makes doing practical computations with a group corresponding to an infinite lattice potentially difficult. The simplest solution to this problem is to restrict the affine Coxeter group to a finite group.

Although in theory the lattices used for many numerical calculations are infinite, to make computations feasible they are restricted to a finite number of points by imposing some kind of boundary condition. In this work we make the simplifying assumption that the boundary condition is always periodic in all directions [3]. To make group computations practical the infinite lattice can be limited by adding the relation $g^L = 1$ for g a generator of the translation subgroup, where $L - 1$ is the side length of the lattice in all dimensions. For example, the infinite two dimensional hypercubic lattice can be restricted to the finite lattice with side length 4 in Figure 1 by adding the relation $g^5 = 1$ to the group presentation. It is not possible to specify different side lengths using multiple relations because for any generator a, there exists a rotation r such that $a^L = 1$ is equivalent to $b^L = (rar^{-1})^L = ra^L r^{-1} = 1$ where b is the generator of the translation subgroup that points along the shortest side length L. As such it does not matter which translation subgroup generator is used for the restriction.

Once this restriction has been made it is possible to enumerate the elements of the lattice as a finite group and obtain the group multiplication table, making group calculations straightforward. This applies to all steps, including calculating the translation subgroup generators (and their inverses).

Applying the restriction in this way introduces some problems. For example, it is no longer enough to check for a loop by seeing if the step sequence multiplies to the identity, as this can happen when a path in one direction is as long as the side length of the lattice corresponding to the restricted affine group, owing to the extra relation. In the cases we are interested in (for lattice QCD calculations) this situation would not arise as path lengths are restricted to prevent this from happening; such paths can be of interest (e.g., Polyakov loops) but we do not deal with them directly in this work.

B. Finite Sublattices on Parallel Machines

A global stencil application is data-parallel and this is often exploited to distribute the work over the processing nodes of a parallel machine. Assuming the lattice dimensions are such that it is possible to distribute the work fairly, each node is assigned a contiguous sublattice of the same size. The points on such a sublattice can be represented by adding a relation to the group corresponding to the infinite lattice as described above, but with a smaller side length that subdivides the side length of the main lattice.

When dealing with computations for per-node sublattices it becomes possible for paths to be long enough to equal the side length of the sublattice, thus raising the possibility of false positives when looking for loops. Consequently certain group computations (such as detecting loops) need to be performed using elements of the parent finite lattice before mapping the result down to the sublattice representing an individual node.

[3]Boundary conditions are almost always periodic or antiperiodic in lattice QCD, and the latter can be treated exactly the same way as the periodic case

C. Manipulating Stencils and Generating Lattice Paths

By taking a single stencil to be a collection of elements of the free group (i.e., step sequences $[p_1 \dots p_n] \in P$), the lattice paths representing the computations for a global stencil application on a finite lattice can be enumerated by generating the cross product of the set of free group elements constituting the stencil with the set of lattice points. A stencil with certain symmetries can be generated from an initial set of free group elements by transforming them using elements of the relevant finite Coxeter group as inner automorphisms. A subsequence is transformed by treating each generator in the word as a member of the main (finite) lattice group and transforming it individually. For example, the pair of sequences $(\uparrow \leftarrow)$ and $(\uparrow \uparrow)$ generate the full stencil shown in Figure 1 when acted on by the corresponding finite Coxeter group (i.e., by reflection and rotation). Similarly, a stencil can be checked for symmetry by ensuring that the transform of each path is already in the stencil. Stencils that consist of all paths of a given length can be generated by enumerating the words in the free group up to some length.

VII. Path Optimisation

We examine two types of redundancy; that which arises from common path segments and that which arises from the equivalence of paths for some computations. Both types of redundancy can be further split into cases that are local to an individual stencil and those that result from considering the collection of stencils that make up a global stencil application.

A. Common Path Segments

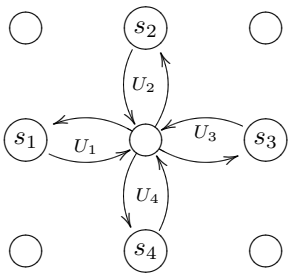
When transporting a value to the seed of a stencil, a path is applied in reverse. The corresponding transformation can be represented as an ordered sequence of the links that are traversed, with the final link on the left of the sequence. For example, the first path in the stencil in Figure 1 is represented as U_1U_2. This is consistent with notation used for the sequence of transformations that the path constitutes, i.e., given some operand to the right of the (expression representing the) path, the transformations are applied from right to left, i.e., $U_1U_2v_1$. It is also consistent with treating paths as a sequence of translations from the origin to the final point (i.e., from left to right). The extra structure that allows optimisation is the fact that a stencil expression consists of a combination of the transported values where the operation used to combine them distributes over the link transformations. Combining the expressions corresponding to two paths that have steps (and therefore transformations) in common and performing such a distribution amounts to factorisation of the expressions involved.

Taking a single stencil in isolation, common path segment elimination equates to removing common path prefixes. If the representations of any two paths have a common prefix, then it can be factored out. In other words, once two values have been transported to the same site (other than the seed) they can be combined and a single value can then be transported thereafter, thereby avoiding the need to apply the sequence of link transformations corresponding to the common prefix twice (i.e., to two different values). An example of this as applied to Figure 1 is given in Figure 4. While this optimisation saves computation, it requires extra space to hold the intermediate result that is being accumulated before being propagated over the common prefix (i.e., the subterms in parentheses).

The factoring is performed in a straightforward manner by constructing trees representing sets of step sequences with common prefixes. The set of paths is first divided into subsets representing individual trees, based on the first step in the sequence.

$$\begin{array}{l} U_1U_2v_1 + U_1U_3v_2 + U_1U_4v_3 + \\ U_5U_6v_3 + U_5U_7v_4 + U_5U_8v_5 + \\ U_9U_{10}v_5 + U_9U_{11}v_6 + U_9U_{12}v_7 + \\ U_{13}U_{14}v_7 + U_{13}U_{15}v_8 + U_{13}U_{16}v_1 \end{array} \rightarrow \begin{array}{l} U_1(U_2v_1 + U_3v_2 + U_4v_3) + \\ U_5(U_6v_3 + U_7v_4 + U_8v_5) + \\ U_9(U_{10}v_5 + U_{11}v_6 + U_{12}v_7) + \\ U_{13}(U_{14}v_7 + U_{15}v_8 + U_{16}v_1) \end{array}$$

Fig. 4. Common prefix elimination applied to the stencil expression from fig. 1



$$\begin{array}{l} t_1 := U_1s_1 \\ t_2 := U_2s_2 \\ t_3 := U_1s_3 \\ t_4 := U_2s_4 \\ s_1^{new} := \ldots + U_1^{-1}(t_2 + t_3 + t_4) \\ s_2^{new} := \ldots + U_2^{-1}(t_1 + t_3 + t_4) \\ s_3^{new} := \ldots + U_3^{-1}(t_1 + t_2 + t_4) \\ s_4^{new} := \ldots + U_4^{-1}(t_1 + t_2 + t_3) \end{array}$$

Fig. 5. A diagram and the corresponding expression showing how common postfix elimination might be applied to paths from different scatter stencils (based on the example in fig. 1). The values at s_1, s_2, s_3 and s_4 are scattered to the point in the centre (to give $t_1, \ldots$ etc), combined in various ways and the results are then scattered back to contribute to $s_1^{new} \ldots$ etc. This postfix optimisation saves a total of 8 link traversals on top of prefix elimination for the paths shown. Note that these are only partial expressions taken from a number of stencils (hence the ellipsis for s_1^{new} etc).

Each set is then further divided based on the second step, and a branch is added to each tree for each step corresponding to a nonempty subset. This continues until all the necessary steps have been added to the tree. The amount of extra storage required for the factored computation is determined by the order in which parts of the expression are evaluated and the number of levels of branching that exist in the tree. This trade-off between total computation, expression ordering and space requirements implies a nontrivial trade-off affecting performance on a real machine (e.g. involving amount of arithmetic, cache locality and cache footprint respectively), but for the moment we make the simplification that prefix factoring is always beneficial and the order of path segment computation is irrelevant; a more detailed study is left to future work.

For a global stencil application there is a case analogous to common prefix elimination; the common section is at the beginning of the paths rather than the end — i.e., a single value is destined for several different stencil expressions. This *scatter* optimisation is just the dual of the *gather* optimisation discussed above, with factorisation of a complimentary stencil defining the different seeds that a value is to be sent to, rather than where values are coming from. As such it offers the same path prefix savings when treating individual stencils separately. Note that when scattering, a value is only combined with other values at the end of any given path. The two approaches can be synthesised by alternating the steps of a scatter with a step that combines values at intermediate nodes before pushing them further down a path resulting in common postfix elimination as well; see Figure 5. This form of cross-stencil optimisation is not further investigated in this paper however, as it is more complex to implement than the single stencil case and suffers from very rapid increases in required storage for intermediate results.

B. Path Equivalence

For some computations, sets of paths can be grouped into equivalence classes owing to some algebraic property of the associated expressions that result in them producing the same answer. Thus, it is only necessary to compute the expression

corresponding to one representative from each equivalence class. Note that these equivalence relations only apply for certain computations, as opposed to common path segments elimination which is always applicable (assuming distributivity). The two equivalence relations used as an example in this paper are reversal and cyclic permutation of loops, redundancies that arise from computing matrix traces of the values produced by paths.

Of the two equivalence relations, reversal is local to individual stencils as the path must start and end at the same point, whereas cyclic permutation exists between loops from different stencils. For the sake of uniformity, the same technique is used for both local and global equivalences. It requires that the equivalence relation is embodied by a function that takes a lattice path in a finite lattice to another equivalent lattice path such that starting from any element of an equivalence class (i.e., a path) it is possible to reach a given element by iterating the function – i.e., there is a least one element that serves as the root of the class. If the set of lattice paths being grouped into equivalence classes is finite and the function embodying the equivalence relation is closed under the set then the equivalence classes can be found as follows. By applying the function to each lattice path once we get a set of ordered pairs of elements. These pairs can be sorted into sets where no element of any pair is present in more than one set. These sets denote the transitive closure of the equivalence function starting from any element, and all individual elements present in a given set are equivalent.

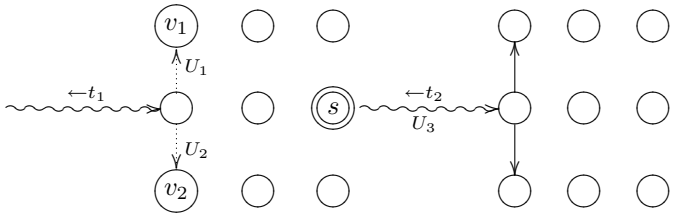
The function for path reversal is trivial to construct. Cyclic permutation requires multiplying the first step in the sequence with the lattice path start point to give a new start point, and a cyclic shift of the step sequence elements. The loops that are to be sorted into equivalence classes can be found using the methods described earlier in the paper. For these two examples the choice of representative for the equivalence class is unimportant, but in the general case there may be an advantage to applying some criterion. For example, where paths of different length can be equivalent it may be better to choose the representative as the shortest path to minimise the necessary amount of computation. Also, it is better to pick representatives such that the set of paths to be computed from each seed point is the same; this simplifies code generation.

Note that removing redundant equivalent paths ought to be done before common path segment elimination to save effort in the latter step.

VIII. Code Generation

Code generation for a single node machine is relatively straightforward. The first step is to group together contiguous sets of points in the lattice that have the same stencil structure. The second step is to generate loops over the contiguous sets of points, where the body of the loop is generated from the expression tree for the required stencil. The seed of the stencil and the stencil paths are used to calculate the offsets for site values and link transformations used in an expression. The typical case would be a single loop over all the points in the lattice applying the same stencil. This may be complicated slightly by boundary conditions requiring separate loops for boundary points, but this is not necessary in our example assuming that the indexing function used to retrieve values and link transformations takes the wrap-around of the boundary condition into account (this is a common technique based on pre-computing arrays of pointers to neighbours).

Generating code for the nodes of a parallel machine is more interesting. Code for expressions based on paths contained



(a) Two sublattices and paths that cross their boundaries

$t_1 := U_1 v_1 + U_2 v_2$
$send(t_1)$ whilst $receive(t_2)$
$s := U_3 t_2$

(b) Pseudo-code

Fig. 6. This figure shows two paths that start at point s and cross the boundary of a 3×3 sublattice, and how the simultaneous computation of these paths for each sublattice is arranged. The dotted arrows represent the computation on the local node required by the neighbour on the left, corresponding to the parts of the paths that cross onto the neighbour on the right. The local node first performs the computation corresponding to the dotted paths, and then passes that result to the left whilst receiving the corresponding result from the right. It then performs the remainder of the computation locally. Pseudo-code corresponding to this is given on the right.

entirely within a local sublattice is generated in the same way as before. In contrast, an expression involving a path that crosses a boundary between two sub volumes requires part of the expression for that path to be computed on one node, communication of an intermediate value from that node to its neighbour (where the centre of the stencil resides) and further computations on the "home" node. This can be viewed as a special type of boundary condition that applies for all sublattices. An example of this is shown in Figure 6(a). This can be extended to paths that cross multiple boundaries.

When paths cross boundaries, the symmetry of the computation across the machine can be exploited to automatically insert communications primitives in the correct place and arrange the global computation. Given that each node must compute the same set of stencils, when the home node requires a neighbour to perform some computation on its behalf for some stencil, the neighbour in the opposite direction will also require some computation from the home node for the stencil in the equivalent position on that neighbouring node. The points visited by the relevant path are generated by computing the destination of each step using the group corresponding to the lattice subvolume local to the node. This automatically causes the path to wrap around when it crosses the boundary of the subvolume as a result of the extra relation. Generating code for the expression proceeds as before, except that any step that causes the path to wraparound requires insertion of a *send-and-receive* communication primitive. This denotes that the value should be communicated to a neighbour and simultaneously the necessary value should be received from the other neighbour in the opposite direction. This approach can be extended in a straightforward manner to trees representing a combination of some number of paths. Note that prefix factoring for this type of path equates to a reduction in communication on a parallel machine, which is likely to be an important optimisation. An example of this using pseudo-code is given in Figure 6(b). Our approach thus exploits the underlying symmetry of the problem and its expression using Coxeter groups to get parallel code generation for free.

IX. Results

We have prototyped the framework and optimisation techniques described in this paper using a system written in Aldor [16]. While experimental work is ongoing, we have initial single processor results demonstrating the effectiveness of common path segment elimination. The experiments were performed on a Pentium 4 workstation, using a two-dimensional lattice with 500^2 sites and a three-dimensional lattice with 50^3, and path lengths of 2, 3 and 4, where site values and link transforms are 2×3

TABLE II
EXPERIMENTAL RESULTS FOR LOW DIMENSION

Dimension	Path Length	Unfactored		Factored		Speedup
		L.O.C.	time (s)	L.O.C.	time (s)	
2D	2	37	4.00	37	2.98	1.34
	3	145	16.93	121	9.50	1.78
	4	541	67.04	373	28.90	2.31
3D	2	91	3.96	79	2.69	1.47
	3	601	25.32	409	12.12	2.08
	4	3751	144.40	2059	53.33	2.70

and 3×3 complex matrices respectively. Each experiment uses one single stencil with all paths of that length, generated automatically by enumeration. The results are given in Table II. The complexity of the resulting stencils can clearly be seen in the number of lines required to encode them; there is approximately one call to a function per line, where each function performs some matrix operation such as applying a link transformation or adding together site values. The performance improvements are signifcant, with speedups of 1.34 to 2.7. The results also show that the importance of optimisation increases rapidly as the dimension and path length of the stencil increases.

X. CONCLUSION

In this article we have shown how to generate any shape discrete regular lattice with an arbitrary number of dimensions and sets of paths on those lattices using the theory of Coxeter groups. This formalism can be used to reason about inter-processor communications and sets of paths and to manipulate the corresponding expressions. We use this to enable optimisations and automate code generation for a parallel machine. Although here applied to lattice QCD, the same techniques could be applied to generating code for any problem that requires such stencil computations. The optimisations in our prototype give significant speedups.

There are several possible extensions to this work, such as examining the performance trade-off between extra space and computation (Section VII-A) and implementing more traditional optimisations such as loop unrolling, tiling etc. and especially prefetching, which is known to be important for this type of code. An interesting addition to code generation for parallel machines would be message vectorisation. If generating large numbers of paths turns out to be computationally expensive, it may be necessary to investigate alternative representations of groups, such as using subgroups of permutation groups. More theoretical extensions include investigating different embeddings of higher dimensional lattices into lower dimensional machines, and handling cases where it is cheaper to communicate link transformations rather than the values themselves.

REFERENCES

[1] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[2] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.

[3] S.-C. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM Press, 2006, pp. 222–232.

[4] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[5] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[6] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn, "POOMA: A Framework for Scientific Simulations on Paralllel Architectures," in *Parallel Programming in C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 547–588. [Online]. Available: citeseer.ist.psu.edu/reynders96pooma.html

[7] T. L. Veldhuizen, "Arrays in blitz++," in *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1998.

[8] K. Osmond, O. Beckmann, and P. Kelly, "A Domain-Specific Interpreter for Parallelizing a Large Mixed-Language Visualisation Application," in *Proceedings of LCPC 2005*, October 2005. [Online]. Available: http://pubs.doc.ic.ac.uk/Domain-Specific-Interpreter/

[9] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[10] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 32.

[11] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling stencils in high performance fortran," in *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 1997, pp. 1–20.

[12] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, 2004.

[13] G. R. Pike, "Reordering and storage optimizations for scientific programs," Ph.D. dissertation, Computer Science Division, University of California, Berkley, 2002, chair-Paul N. Hilfinger.

[14] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 115–133, 2004.

[15] *GAP – Groups, Algorithms, and Programming*, The GAP Group, University of St. Andrews, UK, 2006.

[16] S. Watt, *Aldor Users Guide*, 2002. [Online]. Available: http://www.aldor.org/docs/aldorug.pdf

[17] J. Stembridge, "Coxeter/weyl for maple." [Online]. Available: http://www.math.lsa.umich.edu/~jrs/maple.html

[18] W. Bosma, J. Cannon, and G. Matthews, "Programming with algebraic structures: design of the magma language," in *Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation*, M. Giesbrecht, Ed. Oxford: ACM, 1994, pp. 52–57.

[19] J. Johnson, "Generating symmetric fft algorithms," Talk at Schloss Dagstuhl Seminar *Challenges in Symbolic Computation Software*, July 2006.

[20] N. Bourbaki, *Groupes et Algèbre de Lie*. Hermann, Paris, 1968, ch. IV–VI.

[21] H. S. M. Coxeter, *Regular Polytopes*. New York: Dover, 1973.

[22] J. A. Todd and H. S. M. Coxeter, "A practical method for enumerating the coset of a finite abstract group," in *Proceedings of the Edinburgh Mathematical Society*, ser. 2, 1937, vol. 5, pp. 26–34.