Coxeter Lattice Paths

T. J. Ashby[1], A. D. Kennedy[2], and S. M. Watt[3]

[1] Institute for Computer Systems Architecture,
University of Edinburgh, Scotland, UK
T.Ashby@ed.ac.uk
[2] Particle Physics Theory,
University of Edinburgh, Scotland, UK
adk@ph.ed.ac.uk
[3] Computer Science Department,
The University of Western Ontario, London, Canada
Stephen.Watt@uwo.ca

Abstract. An important step in lattice QCD, and various supercomputing applications in general, consists of updating the values associated with sites on a regular lattice based on a neighbourhood of each site described by a *stencil* centred on it. The path from a neighbour to the centre of the stencil may also play a part in the update by the action of link transformations on values that traverse a given link. This paper describes a method to automatically generate code and carry out an important class of optimisations for such problems on large parallel computers, applicable to stencils expressed as paths on any given regular lattice type and dimension. Our technique is based on the theory of affine Coxeter groups and revolves around recovering and manipulating the generators of the translation subgroup corresponding to the desired lattice. The method has been prototyped in the Aldor language.

1 Introduction

Physical systems are often homogenous or isotropic. Such symmetries may be discrete, as for a crystal lattice, or continuous as in the space-time vacuum of quantum field theories. In the latter case numerical computations make use of a discrete approximation to render the problem finite. It is necessary that the physics of these systems, which is often encoded in an action, respect these symmetries. While the simplest form of the action may be easy to write by hand, it is often desirable to to use higher-order discretization schemes ("improved actions") to reduce the discretization errors, and these can become extremely complicated. The numerical simulation of such physical systems is one of the principal consumers of time on large-scale parallel computers, and the intrinsic parallelism of the underlying physics often leads to the most natural and effective way of parallelizing the computations.

This leads to a significant programming problem, as the mapping of the symmetries of the physical system to those of a parallel computer often depends on the specific machine topology used, and on the detailed parameters of the

Dagstuhl Seminar Proceedings 06271
Challenges in Symbolic Computation Software
http://drops.dagstuhl.de/opus/volltexte/2006/769

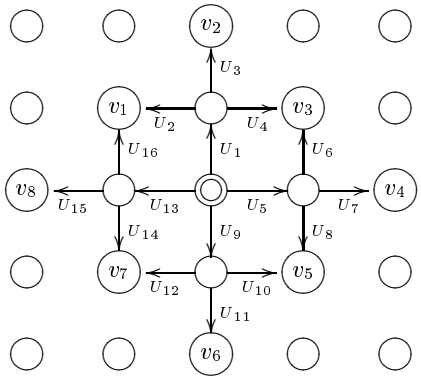
problem. For example, the amount of computation allocated to a machine node depends on both the size of the physical system being simulated and on the size of the computer used to simulate it. The goal of this paper is to automate the generation and certain optimisations of programs to implement such computations by taking a group-theoretic approach that makes both the symmetries of the problem and the machine explicit. Although our original motivation comes from problems in lattice QCD, we aim to keep the treatment as general as possible.

The action is usually of the form of a sum over all lattice sites of a discrete differential operator applied to some field, for example a Laplacian or Dirac operator. In discrete form this corresponds to calculating an expression for each lattice site the sum over some number of neighbouring sites of the field values with various weights. The pattern of neighbours is the same for each site, and we call such a pattern a *stencil*, the point to which an instance of the expression corresponds the *central point* (or simply centre) of the stencil, and the computation of a given stencil expression for each site on the lattice a *global stencil application*.

Often the field variables on the sites are connected to each other with connections or gauge fields living on the links of the lattice. These gauge fields may represent electromagnetic interactions, the gluonic interactions of the strong nuclear force, the distortion of spacetime in general relativity or inhomogeneous grid spacings in more general problems. More concretely, there are a set of transformations associated with the links on the lattice, with one transformation per link. Before contributing to the stencil expression, the value at a given site in the neighbourhood must be transported to the centre of the stencil, being transformed by the *link transformation* of every link it traverses along its path. The values involved in a stencil depend upon the end points of the paths, and we define a stencil as being a set of paths *from* a central point *to* these end points (a value is transported backward along a path).

There are many benefits to having some formal way of representing and reasoning about stencils as we have defined them above. The first advantage is that we may define a rigorous way of enumerating all possible stencils of a certain class (i.e., involving paths of a certain length) for a particular lattice. The second is that we may reason about stencils; the ability to check for certain properties, such as rotational symmetry, may be useful. Although doing this by hand may be simple for low dimensional lattices, such manipulations rapidly become unintuitive as the number of dimensions and path length increases. The third is that we can automate the generation of code to compute expressions from the automatically generated stencils, including the case where a lattice is distributed over the nodes of a parallel machine. Finally, naively generating code from stencils and global stencil applications can result in redundant computations; the ability to reason about stencils as collections of paths gives us the opportunity to develop automatic optimisations.

The rest of the paper proceeds as follows: Section 2 gives the running example, Section 3 introduces Coxeter groups and their relation to lattices, Section 4



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

Fig. 1. On the left is a simple stencil on a two-dimensional square lattice, with the central 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 $v_1 \ldots v_8$ and sixteen link transformations $U_1 \ldots U_{16}$. For a global stencil application, such an expression would be computed for the stencil centred on each point on the lattice.

describes how to recover a lattice from an affine Coxeter group and defines paths on the lattice, Section 5 discusses group computations with finite lattices, and defines sublattices and how to generate and manipulate sets of paths, Section 6 describes optimisation of paths, Section 7 covers some aspects of code generation and finally Section 8 concludes.

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 as elements of a group, and exploits the group structure to reason about sets of paths.

2 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 centre. The link transformations are linear, and the (linear space) addition of site values at the centre distributes over the link transformations. This example is a simplified version of the structure found in some lattice QCD computations. A simple stencil on a two dimensional lattice and the associated expression is shown in Figure 1.

3 Coxeter Groups and Lattices

Although in the context of lattice QCD we are chiefly interested in four dimensional hypercubic lattices, using Coxeter groups automatically provides the generality to cope with arbitrary lattice types in any number of dimensions.

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. As the lattice is regular and n is finite, any one translation can be built up from a finite set consisting of the linearly independent translations to the immediate neighbours of the origin and their inverses. Hence the translations form a finitely generated (infinite) group under composition, with the identity element representing the origin. Alternatively, the translations can be viewed as symmetry transformations of the infinite lattice, again forming an infinite group under composition generated by the n smallest linearly independent translations. 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 faithfully represented by (i.e., 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.

4 Lattices and Lattice Paths

4.1 Lattices

As part of the classification process for Coxeter groups [1], 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. The first step in generating a lattice from an affine Coxeter group is to choose one such generator. Figure 2 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 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

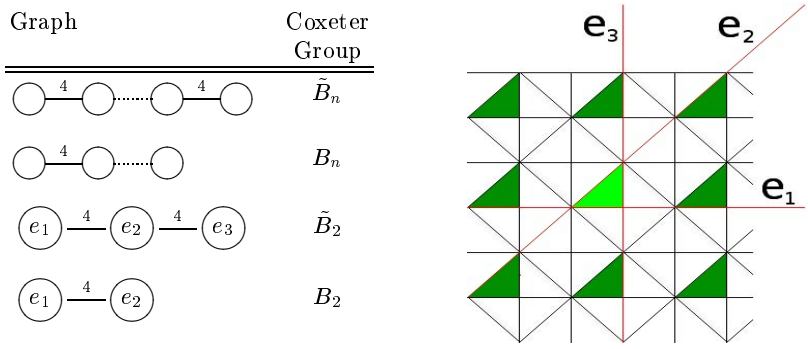


Fig. 2. On the left is a table of graphs for Coxeter groups, showing the general form of the hypercubic group and its finite subgroup $(\tilde{B}_n, B_n)$, and the graphs that correspond to the two-dimensional case used as the running example $(\tilde{B}_2, B_2)$. In the latter two graphs the generators 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.

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, 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 $R_2(e_1) = e_2 e_1 e_2^{-1}$. Note that the inverse is not strictly necessary here as the generators are reflections and therefore their own inverses.

The extra generator and its parallel from the finite group give us one of the generators of the translation group corresponding to the lattice; call this the *first translation generator*, being $e_2 e_1 e_2 e_3$ 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 generator to one of the generators of the translation group or its inverse. The result of such a transform is clearly in the infinite group, and is also still a translation; it must therefore be a translation group generator (or its inverse).

Given that the elements of the finite subgroup provide a double cover of the translation group generators (and their inverses), it is only necessary to

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

Table 1. 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 5.1.

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 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 1. Figure 2 (on the right) shows the translations of the fundamental region that correspond to the translation subgroup. Each translated region corresponds to a lattice point. A further link between the finite subgroup and the translation subgroup is that the former corresponds to the cosets of the latter (in the affine Coxeter group).

4.2 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 sequences P can be treated as the free group generated by the translation subgroup generators $g \in T$. Although it may be possible to define some operations on lattice paths and treat them as elements of an algebraic structure, this is unnecessary for our purposes so we treat lattice paths as a direct product $T \times P$ with no structure other than simple equality between elements (i.e., when two lattice paths are identical). 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. Any path that contains such subsequences can be reduced to a shorter path without them by checking all adjacent pairs of steps in the step sequence and

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

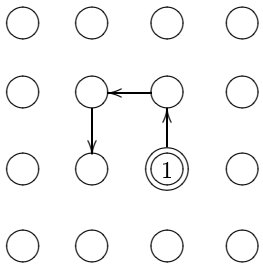


Fig. 3. Above, a lattice path, with below a diagram representing it. The path starts from the origin on the lattice, represented by the identity element. The step sequence is a word in the free group generated by the translation subgroup generators. The directions correspond to the generators given on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

removing any pair that multiply to the identity. 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 as they correspond to the pure gauge field part of the action. 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.

5 Finite Lattices, Sublattices and Path Generation

5.1 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 (or more neatly whether a certain word equals the identity, in this case $ab^{-1} = 1$). This problem has already been mentioned when discussing the shortest words representing the translation subgroup generators in Table 1; 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, although in practice a naive approach may run out of machine resources or take too long; here we assume that computing with finite groups 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 lattice QCD 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 [4]. To make group computations practical the infinite lattice can be limited in the same way by adding relations to the presentation of the affine Coxeter group. More specifically, the relation $g^L = 1$ is added 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 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 as 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 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 due 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.

5.2 Finite Sublattices on Parallel Machines

A global stencil application operation 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.

[4] 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

5.3 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 $p_1 \ldots p_n \in P$), 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 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 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.

6 Path Optimisation

Finding and removing redundant computation relies on exploiting further structure that stencils possess. 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.

6.1 Common Path Segments

When transporting a value to the centre 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 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 first, 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

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

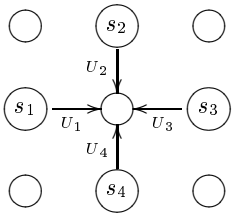
Fig. 4. Common prefix elimination applied to the stencil expression from fig. 1

been transported to the same site (other than the centre) they can be combined and a single value can then be transported to the centre, 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 to the centre (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. 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 on actual hardware 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 analogous *scatter* optimisation is just the dual of the *gather* optimisation discussed above, with factorisation of a complimentary stencil defining the destinations 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 could 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 optimisation is not further investigated in this paper however, as it is more complex to implement than the

single stencil case and may suffer from very rapid increases in required storage for intermediate results.



$$
\begin{aligned}
t_1 &:= U_1 s_1 \\
t_2 &:= U_2 s_2 \\
t_3 &:= U_1 s_3 \\
t_4 &:= U_2 s_4 \\
s_1^{new} &:= \ldots + U_1^{-1}(t_2 + t_3 + t_4) \\
s_2^{new} &:= \ldots + U_2^{-1}(t_1 + t_3 + t_4) \\
s_3^{new} &:= \ldots + U_3^{-1}(t_1 + t_2 + t_4) \\
s_4^{new} &:= \ldots + U_4^{-1}(t_1 + t_2 + t_3)
\end{aligned}
$$

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 from a number of stencils (hence the ellipsis for s_1^{new} etc).

6.2 Path Equivalence

For some computations, sets of paths can be grouped into equivalence classes due 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 to another equivalent lattice path such that starting from any element 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

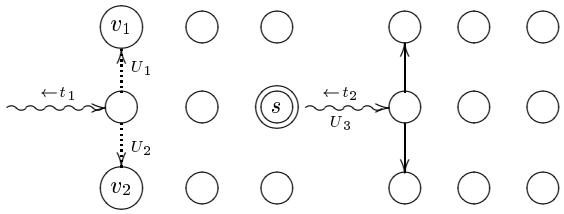


Fig. 6. This figure shows two paths that start at point s and cross the boundary of a 3×3 sublattice, and how the simultaneous computation of these paths for each sublattice is arranged. The dotted arrows represent the computation on the local node required by the neighbour on the left, corresponding 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.

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.

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

7 Code Generation

Naive code generation for a single node is straightforward. It can proceed point by point visiting each site in the finite lattice, generating the code for the expression trees that constitute the stencil at each site. The centre of the stencil and the stencil paths are used to calculate which site values and link transformations are used for an expression. Generating code for the nodes of a parallel machine is more interesting. Code for expressions based on paths contained 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. An example

of this is shown in Figure 6. 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. Thus, if all nodes compute the part of the stencil on behalf of the neighbour for its equivalent stencil, followed by the necessary computation local to the home node, then all neighbouring results can be communicated at the same time as a global shift in one direction. If all nodes compute all their stencils in the same order then global communication is automatically arranged, and this can be done by generating the same code for each node. Our approach thus exploits the underlying symmetry of the problem and its expression using Coxeter groups to get parallel code generation for free.

Code for a path that crosses a boundary is generated as follows. 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 wraparound 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 rather than using the value produced from the part of the path that has wrapped around, that 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 on the right of Figure 6.

8 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 sets of paths, including checking for properties such as rotational invariance etc. It can also be used to manipulate the corresponding expressions to enable such optimisations as common path segment elimination and removal of redundant paths based on equivalence transformations, and automate code generation for the nodes of a parallel machine. Although developed for lattice QCD, the same techniques could be applied to generating code for any problem that requires

such stencil computations. Our approach has been implemented in a prototype system written using the Aldor [2] programming language.

There are several possible extensions to this work. They include generating code for a real machine and examining the performance trade-off between extra space and computation, implementing a more aggressive version of common path segment elimination exploiting common elements across stencils rather than simply elements within a stencil (both mentioned in Section 6.1), and extending the system to handle the expressions that arise from the force term in HMC for QCD when using more complex stencils. An important addition for code generation 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.

References

1. H. S. M. Coxeter. *Regular Polytopes.* Dover, New York, 1973.
2. Stephen Watt et al. Aldor. `http://www.aldor.org`.