Adventures in the Quantum Polynomial Ring: Linear Algebra Computations in C

Nathan Nieman, Justin Lambright*, Courtney Taylor*, and Robert Green†

Department of Mathematics Anderson University and Department of Computer Science Bowling Green State University†

Introduction

This presentation examines methods utilized to improve the speed at which we could observe how elements of the symmetric group form p-Polynomials.

A recursive definition of the p-Polynomials follows:

1. $p_{u,v,w}(q) = 0$ if $w \not\geq u^{-1}v$
2. $p_{u,v,w}(q) = 1$ if $w = u^{-1}v$
3. For every s such that $us < u$,

$$p_{u,v,w}(q) = \begin{cases} p_{us,v,sw}(q) & \text{if } sw > w \\ p_{us,v,sw}(q) + qp_{us,v,w}(q) & \text{otherwise} \end{cases}$$

Where u, v, and w are elements of the n^{th} symmetric group (denoted S_n).

Results

Program Version	Speed (polys/sec)	Time for S_7
Original	9.8	414 years
Improved	238	17 years
C	419,000	84 hours
Bit Arrays	992,000	35 hours
Parallel	9,750,000	3.6 hours

Original Solution

Our research began with an existing software solution using Python and a combinatorial library from Sage, an open source mathematical toolkit to manipulate the symmetric group elements. Although this solution could perform computations much faster than a human, the need for a faster method becomes apparent when one looks at the sheer size of the calculations which this program is to perform.

Number of polynomials in group $S_n = n!$[3]
S_7 = more than **128,000,000,000 polynomials**

Each polynomial may take many recursions:
S_7 = more than **24,000 recursions**

Calculating all of S_7 at **9.2 polys/sec** would take over **441 years**.

Improved Solution

Simply by cleaning up the old Python code and removing some unnecessary operations being performed, we were able to improve the speed of the calculations by a great deal. Below is a line of code from the original version vs. the improved version.

```
str(eval("(%s)*(%s)*(%s)" % (u, v, w)))
                    ↓
              u * v * w
```

New Way of Doing Things

By representing the symmetric group elements as permutation matrices instead of in the traditional reflection notation we were able to find more computation-friendly way to calculate the p-Polynomials.

$$s_1s_2s_1 \rightarrow \begin{bmatrix} 0&0&1&0 \\ 0&1&0&0 \\ 1&0&0&0 \\ 0&0&0&1 \end{bmatrix}$$

There were, however, several operations which were easy to perform in the reflection notation, but become more difficult in the permutation matrix notation.

$s_2s_1 \leq s_1s_2s_1$ because s_2s_1 appears as a subword of $s_1\underline{s_2s_1}$

$$\begin{bmatrix} 0&1&0&0 \\ 0&0&1&0 \\ 1&0&0&0 \\ 0&0&0&1 \end{bmatrix} \overset{?}{\leq} \begin{bmatrix} 0&0&1&0 \\ 0&1&0&0 \\ 1&0&0&0 \\ 0&0&0&1 \end{bmatrix}$$

In order to move over to this new notation we would have to find a way to preform Bruhat comparisons and other operations on our permutation matrices. The following pseudocode is an adaptation of work by Anders Björner and Francesco Brenti[1]:

Naive Algorithm

For each possible box:
  $count_{A,B} \leftarrow 0$
  For each element inside the box:
    If $A_i = 1$:
      Add one to $count_A$
    If $B_i = 1$:
      Add one to $count_B$
  If $count_A > count_B$:
    return A $\not\leq$ B
return A $\leq$ B

Optimized Algorithm

$accumulator_{A,B} \leftarrow [0 \ldots 0]$
For each row:
  Add row of A to $accumulator_A$
  Add row of B to $accumulator_B$
  $sum_{A,B} \leftarrow 0$
  For each column:
    Add column of $accumulator_A$ to sum_A
    Add column of $accumulator_B$ to sum_B
    If $sum_A > sum_B$:
      return $A \not\leq B$
return $A \leq B$

$\boldsymbol{O(n^4)}$ vs. $\boldsymbol{O(n^2)}$

C Implementation

Using our new algorithms based upon the matrix representation, we moved our code over to C.

1. NxN float arrays to define our permutation matrices
2. Basic Linear Algebra Subprograms (BLAS) implementation for matrix multiplication and transposition

References

[1] A. Björner and F. Brenti. Combinatorics of Coxeter groups. *Graduate Texts in Mathematics*. Springer, New York, 231, 2005.

Bit Arrays

One may notice some unique properties about the permutation matrices used to represent symmetric group elements:

1. Each element is only either a one or zero
2. Only a single one is present in any given row or column

Because of this first property of the permutation matrices, **each row can be viewed as a single binary number**.

$$\begin{bmatrix} 0&1&0&0 \\ 1&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{bmatrix} \rightarrow \begin{bmatrix} 0100 \\ 1000 \\ 0001 \\ 0010 \end{bmatrix} \rightarrow \begin{bmatrix} 2 \\ 1 \\ 8 \\ 4 \end{bmatrix}$$

By representing the permutation matrices in this form, certain operations become significantly less costly, effectively using **bit-level parallelism**.

```
float rowA[N] = {0, 1, 0}
float rowB[N] = {0, 0, 1}
float rowC[N] = {0, 0, 0}

for (int i = 0; i < N; i++):
  rowC[i] = rowA[i] + rowB[i]
                ↓
int rowA = 2 // 01000...
int rowB = 4 // 00100...

int rowC = rowA + rowB // 011000...
```

$\boldsymbol{O(n)}$ vs. $\boldsymbol{O(1)}$

Parallelization

We next used **task-level parallelism** to speed up our computation by splitting our problem into separate tasks and spreading the tasks across multiple processor cores.
We are able to utilize the full processing power of a CPU by splitting up the computation into pieces and using multiple threads.