Algorithm XXX: Efficient Algorithm for Representations of U(3) in U(N)

DANIEL LANGR, Czech Technical University in Prague, Czech Republic
TOMÁŠ DYTRYCH, Czech Academy of Sciences, Czech Republic and Louisiana State University, USA
JERRY P. DRAAYER, Louisiana State University, USA
KRISTINA D. LAUNEY, Louisiana State University, USA
PAVEL TVRDÍK, Czech Technical University in Prague, Czech Republic

An efficient algorithm for enumerating representations of $U(3)$ that occur in a representation of the unitary group $U(N)$ is introduced. The algorithm is applicable to $U(N)$ representations associated with a system of identical fermions (protons, neutrons, electrons, etc.) distributed among the $N = (\eta + 1)(\eta + 2)/2$ degenerate eigenstates of the ηth level of the three-dimensional harmonic oscillator. A C++ implementation of the algorithm is provided and its performance evaluated. The implementation can employ OpenMP threading for use in parallel applications.

CCS Concepts: • **Mathematics of computing** → *Combinatorial algorithms*; • **Theory of computation** → *Shared memory algorithms*;

Additional Key Words and Phrases: C++, group representation, OpenMP, parallel algorithm, unitary group

ACM Reference Format:
Daniel Langr, Tomáš Dytrych, Jerry P. Draayer, Kristina D. Launey, and Pavel Tvrdík. 2010. Algorithm XXX: Efficient Algorithm for Representations of U(3) in U(N). *ACM Trans. Math. Softw.* 9, 4, Article 39 (March 2010), 10 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

We present an algorithm for a fast reduction of irreducible representations (irreps) of the unitary group $U(N)$ into a complete set of irreps of $U(3)$, the symmetry group of the three-dimensional harmonic oscillator (HO). We start with a brief description of the underpinning mathematical procedure [Draayer et al. 1989]. Let $N = (\eta + 1)(\eta + 2)/2$ denotes the number of degenerate eigenstates of the ηth HO level, $\eta = 0, 1, 2 \ldots$. The $U(N)$ irreps associated with a distribution of fermions among these eigenstates is labeled by a Young tableaux

$$[f] = [f_1, \ldots, f_N], \quad f_i \in \{0, 1, 2\}, \quad f_i \geq f_{i+1},$$

where $\sum_i f_i = A$ is equal to the number of fermions. The condition

$$f_i \in \{0, 1, 2\} \tag{1}$$

Authors' addresses: Daniel Langr, Czech Technical University in Prague, Department of Computer Systems, Faculty of Information Technology, Thákurova 9, Praha, 16000, Czech Republic, daniel.langr@fit.cvut.cz; Tomáš Dytrych, Czech Academy of Sciences, Nuclear Physics Institute, Czech Republic, Louisiana State University, Department of Physics and Astronomy, USA; Jerry P. Draayer, Louisiana State University, Department of Physics and Astronomy, USA; Kristina D. Launey, Louisiana State University, Department of Physics and Astronomy, USA; Pavel Tvrdík, Czech Technical University in Prague, Department of Computer Systems, Faculty of Information Technology, Czech Republic.

Manuscript submitted to ACM

follows from the Pauli exclusion principle, which allows each eigenstate to be occupied by either no particles, a single particle, or two particles with different spin projections.

The reduction of a $U(N)$ irrep into $U(3)$ irreps is based on a generation of Gelfand patterns

$$\begin{array}{ccccccccc} g_{1,N} & & g_{2,N} & & g_{3,N} & & g_{4,N} & \cdots & g_{N,N} \\ & g_{1,N-1} & & g_{2,N-1} & & g_{3,N-1} & \cdots & g_{N-1,N-1} & \\ & & g_{1,N-2} & & g_{2,N-2} & \cdots & g_{N-2,N-2} & & \\ & & & \cdots\cdots\cdots\cdots & & & & & \\ & & & g_{1,2} & & g_{2,2} & & & \\ & & & & g_{1,1} & & & & \end{array}$$

where $g_{i,N} = f_i$ and $g_{i,j} \in \{0, 1, 2\}$ satisfies betweenness condition [Gelfand and Tsetlin 1950]

$$g_{i,j} \geq g_{i,j-1} \geq g_{i+1,j}. \tag{2}$$

Each Gelfand pattern labels a unique basis state of the $U(N)$ irrep $[f]$. The number of such patterns is therefore equal to the dimension of $[f]$,

$$\dim[f] = \prod_{l=2}^{N} \prod_{k=1}^{l-1} \frac{f_k - f_l + l - k}{l - k}. \tag{3}$$

Each Gelfand pattern determines a weight vector $\vec{w} = [w_1, \ldots, w_N]$ of a basis state, where

$$w_i = \sum_j g_{j,i} - \sum_j g_{j,i-1} \quad \text{for} \quad 1 \leq i \leq N, \tag{4}$$

and $\sum_j g_{j,0} = 0$ by definition. We assign the N levels of $U(N)$ to degenerate HO eigenstates expressed in Cartesian coordinates, where the ith eigenstate is labeled by non-negative integers $(\eta_{z,i}, \eta_{x,i}, \eta_{y,i})$ that satisfy $\eta = \eta_{z,i} + \eta_{x,i} + \eta_{y,i}$. We arrange the eigenstates in standard speedometer order:

```
i ← 1
for k ← 0, 1, ..., n do
    for l ← k, k − 1, ..., 0 do
        η_{z,i} ← n − k
        η_{x,i} ← l
        η_{y,i} ← k − l
        i ← i + 1
    end
end
```

For each direction, we can form a vector of HO quanta as follows:

$$\vec{\eta}_z = [\eta_{z,1}, \ldots, \eta_{z,N}], \quad \vec{\eta}_x = [\eta_{x,1}, \ldots, \eta_{x,N}], \quad \text{and} \quad \vec{\eta}_y = [\eta_{y,1}, \ldots, \eta_{y,N}]. \tag{5}$$

The ith element of the weight vector $\vec{w}$ specifies the number of fermions occupying the ith eigenstate. As a result, the inner products

$$e_1 = \vec{w} \cdot \vec{\eta}_z, \quad e_2 = \vec{w} \cdot \vec{\eta}_x, \quad e_3 = \vec{w} \cdot \vec{\eta}_y \tag{6}$$

yield a $U(3)$ weight $[e_1, e_2, e_3]$, which corresponds to the total number of HO quanta along z, x, and y directions. Let $M[e_1, e_2, e_3]$ denotes the *multiplicity* of the $U(3)$ weight $[e_1, e_2, e_3]$, i.e., how many times this weight is generated across all basis states of $[f]$. If a $U(3)$ weight satisfies $e_1 \geq e_2 \geq e_3$, then it forms the $U(3)$ irrep $[e] = [e_1, e_2, e_3]$ in the $U(N)$ irrep $[f]$. The dimensionality of $[e]$ in $[f]$, i.e., the number of times a given $U(3)$ irrep $[e]$ occurs in $[f]$, can be obtained

via a simple difference relation:

$$D[e] = M[e_1, e_2, e_3] + M[e_1 + 1, e_2 + 1, e_3 - 2] + M[e_1 + 2, e_2 - 1, e_3 - 1] \\ - M[e_1 + 2, e_2, e_3 - 2] - M[e_1 + 1, e_2 - 1, e_3] - M[e_1, e_2 + 1, e_3 - 1]. \quad (7)$$

The purpose of our algorithm is to enable a fast and efficient, on-the-fly generation of a complete set of $U(3)$ irreps in a $U(N)$ irrep $[f]$ together with their dimensionalities.

2 MOTIVATION AND RELATED WORK

The three-dimensional harmonic oscillator potential is a good starting approximation to the potential that binds nucleons together, and hence the $U(N) \rightarrow U(3)$ group chain plays a central role in nuclear physics [Elliott 1958]. A detailed description of the mathematical procedure for $U(N) \rightarrow U(3)$ reduction was originally presented by Draayer et al. [1989], who also provided a relatively simple FORTRAN code called `UNTOU3` for its implementation. Unfortunately, while technically correct, the `UNTOU3` implementation cannot be applied efficiently to $U(N)$ irreps of very large dimensions, nor is it suitable for on-the-fly applications, especially when the latter is further enabled by parallel processing. The solution presented below circumvents all of these issues, and therefore is a major enabling feature for modern so-called algebraic no-core nuclear structure calculations [Launey 2017].

Specifically, a more recent $U(N) \rightarrow U(3)$ code was developed in C++ as a part of `LSU3shell`—a high performance computing enabled implementation of the *symmetry-adapted no-core shell model* (SA-NCSM) [Dytrych et al. 2013, 2016; Langr et al. 2018]. `LSU3shell` has been up to now applied to study structure of light nuclei, where only relatively small $U(N)$ irreps arise. For example, in one of the largest nuclear structure calculation ever performed with `LSU3shell` (on the Blue Waters supercomputer), the most demanding $U(N) \rightarrow U(3)$ problem involved the following $U(21)$ irrep ($\eta = 5, A = 3$):

$$[f] = [2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$

which is spanned by 3080 Gelfand patterns (3). The solution took 0.0145 seconds on our testbed system.

The recent advances in algorithms and many-body techniques underpinning `LSU3shell` allows us to extend its reach towards heavier nuclei, where much larger irreps are needed and the $U(N) \rightarrow U(3)$ reduction becomes one of the main program bottlenecks. An illustrative example is the following $U(21)$ irrep ($\eta = 5, A = 13$):

$$[f] = [2, 2, 2, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], \quad (8)$$

which is spanned by 2 168 999 910 Gelfand patterns. The corresponding $U(N) \rightarrow U(3)$ reduction took 986.9 seconds on our testbed system. This huge growth in compute time motivated us to develop and implement a new $U(N) \rightarrow U(3)$ algorithm, which is introduced in Section 4.

3 ORIGINAL ALGORITHM

The original algorithm generates Gelfand patterns recursively as illustrated in ALGORITHM 1. The input are the labels of a $U(m)$ irrep $[g_{1,m}, \ldots, g_{m,m}]$, from which it generates all possible $U(m-1)$ irrep labels $[g_{1,m-1}, \ldots, g_{m-1,m-1}]$ that satisfy betweenness condition (2). Before each recursive call, the mth element of the weight vector $\vec{w}$ is updated according to (4). At the end of recursion, the construction of $\vec{w}$ is finalized, a corresponding $U(3)$ weight is evaluated (6), and its multiplicity $M[e_1, e_2, e_3]$ increased accordingly. In `LSU3shell` code, M is represented by a data structure of C++ type `std::map`, which is an associative container typically build up using some form of a binary search

ALGORITHM 1: `ProcessGelfandPatternRow`$([g_{1,m}, \ldots, g_{m,m}], \vec{w})$

Input: $[g_{1,m}, \ldots, g_{m,m}]$: mth row of a Gelfand pattern
Input: $\vec{w}$: constructed weight vector
Output: M: table of multiplicities of resulting $U(3)$ weights; initially, $M[e_1, e_2, e_3] = 0$ for any $[e_1, e_2, e_3]$
Data: $\vec{\eta}_z$, $\vec{\eta}_x$, $\vec{\eta}_y$: HO quanta vectors (5)

```
if m = 1 then
    w_1 ← g_{1,1}
    [e_1, e_2, e_3] ← [w · η_z, w · η_x, w · η_y]
    M[e_1, e_2, e_3] ← M[e_1, e_2, e_3] + 1
else
    for i ← 1, ..., m − 1 do
        G_i ← {g_{i,m−1} ∈ ℕ^0 : g_{i,m} ≥ g_{i,m−1} ≥ g_{i+1,m}}
    end
    for all possible combinations [g_{1,m−1}, ..., g_{m−1,m−1}] ∈ G_1 × ··· × G_{m−1} do
        w_m ← Σ_j g_{j,m} − Σ_j g_{j,m−1}
        ProcessGelfandPatternRow([g_{1,m−1}, ..., g_{m−1,m−1}], w)
    end
end
```

tree. And further, to reduce a $U(N)$ irrep $[f]$ into $U(3)$ irreps, one first needs to run ALGORITHM 1 by calling `ProcessGelfandPatternRow`$([f], \vec{0})$, i.e., with $[f]$ and an empty vector as arguments. The output of the algorithm is a table of multiplicities M, which enables one to obtain resulting $U(3)$ irreps and their dimensionalities (7).

ALGORITHM 1 is generic. It can be applied for $U(N)$ irreps with unconstrained labels $f_i \in \mathbb{N}^0$, thereby enabling classification of particles with additional quantum properties such as, e.g., isospin. A major disadvantage of this procedure is low performance due to frequent dynamic memory allocations. These are required primarily for construction of lower Gelfand pattern rows, which are represented by dynamic arrays (C++ `std::vector` containers) whose sizes are not known until runtime. Note that additional dynamic memory allocations are required when new $U(3)$ weights are added into M. However, this only happens occasionally in practice since, generally, $\left|\left\{[e_1, e_2, e_3] : M[e_1, e_2, e_3] > 0\right\}\right| \ll \dim[f]$. For instance, in the $U(21)$ irrep (8), there are only 1365 distinct $U(3)$ weights with nonzero multiplicities. Consequently, on average, only every 1.6 millionth access to M requires adding a new entry. A second cause of low performance of ALGORITHM 1 is that its implementation in `LSU3shell` is not parallelized.

4 NEW ACCELERATED ALGORITHM

Taking condition (1) into account, we can represent each row in a Gelfand pattern by three integers n_2, n_1, and n_0, corresponding to the number of labels that are equal to 2, 1, and 0, respectively. For instance, $n_2 = 6$, $n_1 = 1$, and $n_0 = 14$ specify the $U(21)$ irrep (8). Such an approach allows us to avoid using dynamic arrays and therefore renders the vast majority of dynamic memory allocations unnecessary. However, we also need to find rules for the construction of all possible $(m-1)$th pattern rows from an mth pattern row represented by $[n_2, n_1, n_0]$ (i.e., $m = n_2 + n_1 + n_0$). Such rules can be derived from the betweenness condition (2) as follows:

(1) If $n_2 + n_1 + n_0 = 1$, the recursion can be ended with $w_1 = 2n_2 + n_1$. Otherwise:
(2) If $n_2 > 1$, $n_1 = 0$, and $n_0 = 0$, the only possible $(m-1)$the row is represented by $[n_2 - 1, 0, 0]$ $(w_m = 2)$.
(3) If $n_2 = 0$, $n_1 > 1$, and $n_0 = 0$, the only possible $(m-1)$the row is represented by $[0, n_1 - 1, 0]$ $(w_m = 1)$.
(4) If $n_2 = 0$, $n_1 = 0$, and $n_0 > 1$, the only possible $(m-1)$the row is represented by $[0, 0, n_0 - 1]$ $(w_m = 0)$.

ALGORITHM 2: `AcceleratedProcessGelfandPatternRow`$([n_2, n_1, n_0], [\delta_1, \delta_2, \delta_3])$

Input: $[n_2, n_1, n_0]$: representation of mth row of a Gelfand pattern, where $m = \sum_i n_i$
Input: $[\delta_1, \delta_2, \delta_3]$: partial contributions to resulting $U(3)$ weights $[e_1, e_2, e_3]$
Output: M: table of multiplicities of resulting $U(3)$ weights; initially, $M[e_1, e_2, e_3] = 0$ for any $[e_1, e_2, e_3]$
Data: $\vec{\eta}_z$, $\vec{\eta}_x$, $\vec{\eta}_y$: HO quanta vectors (5)

```
m = n2 + n1 + n0
if m > 1 then
    if n2 > 0 then
        AcceleratedProcessGelfandPatternRow([n2 − 1, n1, n0], [δ1 + 2η_{z,m}, δ2 + 2η_{x,m}, δ3 + 2η_{y,m}])
        if n0 > 0 then
            AcceleratedProcessGelfandPatternRow([n2 − 1, n1 + 1, n0 − 1], [δ1 + η_{z,m}, δ2 + η_{x,m}, δ3 + η_{y,m}])
        end
    end
    if n1 > 0 then
        AcceleratedProcessGelfandPatternRow([n2, n1 − 1, n0], [δ1 + η_{z,m}, δ2 + η_{x,m}, δ3 + η_{y,m}])
    end
    if n0 > 0 then
        AcceleratedProcessGelfandPatternRow([n2, n1, n0 − 1], [δ1, δ2, δ3])
    end
else
    tmp ← 2n2 + n1
    [e1, e2, e3] ← [δ1 + tmp · η_{z,1}, δ2 + tmp · η_{x,1}, δ3 + tmp · η_{y,1}]
    M[e1, e2, e3] ← M[e1, e2, e3] + 1
end
```

(5) If $n_2 > 0$, $n_1 > 0$, and $n_0 = 0$, there are 2 possible $(m-1)$th rows represented by $[n_2, n_1 - 1, 0]$ $(w_m = 1)$ and $[n_2 - 1, n_1, 0]$ $(w_m = 2)$.
(6) If $n_2 = 0$, $n_1 > 0$, and $n_0 > 0$, there are 2 possible $(m-1)$th rows represented by $[0, n_1, n_0 - 1]$ $(w_m = 0)$ and $[0, n_1 - 1, n_0]$ $(w_m = 1)$.
(7) If $n_2 > 0$, $n_1 = 0$, and $n_0 > 0$, there are 3 possible $(m-1)$th rows represented by $[n_2, 0, n_0 - 1]$ $(w_m = 0)$, $[n_2 - 1, 1, n_0 - 1]$ $(w_m = 1)$, and $[n_2 - 1, 0, n_0]$ $(w_m = 2)$.
(8) If $n_2 > 0$, $n_1 > 0$, and $n_0 > 0$, there are 4 possible $(m-1)$the rows represented by $[n_2, n_1, n_0 - 1]$ $(w_m = 0)$, $[n_2, n_1 - 1, n_0]$ $(w_m = 1)$, $[n_2 - 1, n_1, n_0]$ $(w_m = 2)$, and $[n_2 - 1, n_1 + 1, n_0 - 1]$ $(w_m = 1)$.

These rules can be further simplified into a form used in our new proposed algorithm for the recursive Gelfand pattern generation presented as ALGORITHM 2. In contrast to ALGORITHM 1, partial inner products

$$[\delta_1, \delta_2, \delta_3] = \left[\sum_{i=m+1}^{N} w_i \eta_{z,i}, \sum_{i=m+1}^{N} w_i \eta_{x,i}, \sum_{i=m+1}^{N} w_i \eta_{y,i} \right]$$

represent the second parameter thereby avoiding the use of dynamic arrays for storing weight vectors $\vec{w}$. The resulting inner products (6) are therefore calculated stepwise during the recursive process.

ALGORITHM 2 takes the same first argument as ALGORITHM 1 except that $U(N)$ irrep $[f]$ is now specified in terms of $[n_2, n_1, n_0]$, and hence it is called as `AcceleratedProcessGelfandPatternRow`$([n_2, n_1, n_0], [0, 0, 0])$. The basic implementation of ALGORITHM 2—without additional optimizations and parallelization (see Section 5 for details)—achieved on our testbed system for the irrep (8) a factor of 6 speedup with respect to ALGORITHM 1; namely, it reduced the runtime from 986.9 to 158.4 seconds.

5 IMPLEMENTATION

We have implemented ALGORITHM 2 in a form of a standalone C++ header file `UNtoU3.h` that comprises the supplemental part of this article. The usage of `UNtoU3.h` requires a C++ compiler compliant with the C++11 Standard (supported nowadays by all mainstream compiler vendors). Within `UNtoU3.h`, we define a class template `UNtoU3<T,U>` that has the following application programming interface (API):

- `T` template parameter—type of the computer representation of $U(3)$ weight labels e_1, e_2, and e_3; by default, it is 32 bit unsigned integer.
- `U` template parameter—type used for storing multiplicities $M[e_1, e_2, e_3]$; by default, it is 32 bit unsigned integer.
- `U3Weight` public member type—an array of 3 integers of type `T` that represents a $U(3)$ weight $[e_1, e_2, e_3]$.
- `U3MultMap` public member type—type of the computer representation of the resulting table M. If optimizations are disabled, it is a type alias for `std::map<U3Labels,U>` (see Section 5.1 for explanation).
- `generateXYZ(`n`)` public member function—generates vectors $\vec{\eta}_z$, $\vec{\eta}_x$, and $\vec{\eta}_z$ (5) for a given η and stores them internally. These vectors can be used repeatedly for solutions of multiple $U(N) \rightarrow U(3)$ problems (`generateU3Weights` calls) as long as η is maintained.
- `generateU3Weights(`n_2, n_1, n_0`)` public member function—implements ALGORITHM 2 itself. The function parameters define an input $U(N)$ irrep $[f]$ represented by $[n_2, n_1, n_0]$.
- `multMap()` constant public member function—returns a constant reference to M of type `U3MultMap` generated by the `generateU3Weights` function. M contains only $U(3)$ weights $[e_1, e_2, e_3]$ with nonzero multiplicities.
- `getDimensionality(`$[e_1, e_2, e_3]$`)` constant public member function—if $[e_1, e_2, e_3]$ (parameter of type `U3Weight`) is a $U(3)$ irrep ($e_1 \geq e_2 \geq e_3$), returns its dimensionality $D[e]$; otherwise returns 0.

An exemplary usage of the `UNtoU3` class for generating $U(3)$ irreps in the $U(6)$ irrep

$$[f] = [2, 1, 1, 1, 1, 0] \tag{9}$$

can be performed by a C++ program shown in Figure 1. The program output reads

```
[5,4,3] : 1
[6,4,2] : 1
```

since the U(6) irrep (9) contains only two $U(3)$ irreps, [5, 4, 3] and [6, 4, 2], both with dimensionality equal to one (for detailed elaboration of this case, see [Draayer et al. 1989]).

5.1 Optimizations

Additionally, we implemented the following performance optimization techniques:

Opt. I. The first optimization reduces the number of recursive calls by applying the *tail call elimination* (TCE). Namely, within ALGORITHM 2, the outermost **if** statement is replaced by a **while** loop that preserves the evaluated condition. In the loop, the last (tail) recursive call is eliminated and the corresponding $(m-1)$th row is resolved in the next iteration instead. The outermost **else** statement is removed and the underlying commands are performed unconditionally after the loop ends. Consequently, within each call of `AcceleratedProcessGelfandPatternRow`, one recursive call is eliminated.

Opt. II. The second optimization is to store multiplicities of $U(3)$ weights in a hash table instead of a binary search tree. This is accomplished by redefinition of `U3UNMultMap` from `std::map` into `std::unordered_map`. A suitable hash function needs to be provided for map keys of type `U3Weight`, i.e., for an array of 3 integers. Based on

```
#include <iostream>
#include "UNtoU3.h"

int main() {
   UNtoU3<> gen;
   gen.generateXYZ(2);
   gen.generateU3Weights(1,4,1);
   for (const auto & pair : gen.multMap()) {
      const auto & irrep = pair.first;
      if (auto D = gen.getDimensionality(irrep))
         std::cout << "[" << irrep[0] << "," << irrep[1] << "," << irrep[2] << "] : " << D << "\n";
   }
}
```

Fig. 1. Sample usage of UNtoU3 class for the $U(6)$ irrep (9).

microbenchmarking, we ended up with a hash function that places the least significant bytes of e_1, e_2, and e_3 next to each other as a single 64 bit value.

Opt. III. The third optimization precalculates contributions of all possible mth Gelfand pattern rows to $U(3)$ weights for $m \leq 3$. It also reduces the number of recursive calls, since the recursion can be ended when $m \leq 3$ instead of when $m \leq 1$. The precalculation can be done as soon as $\vec{\eta}_z$, $\vec{\eta}_x$, and $\vec{\eta}_y$ vectors are known, therefore, it is performed automatically within the `generateXYZ` function.

All the above described optimizations are applied by default, however, they can be optionally disabled by defining `UNTOU3_DISABLE_TCE`, `UNTOU3_DISABLE_UNORDERED`, and/or `UNTOU3_DISABLE_PRECALC` preprocessor symbols before inclusion of the `UNtoU3.h` header file. Enabling/disabling particular optimizations does not change the usage of the `UNtoU3` class from the programmer's perspective.

5.2 Parallelization

To leverage modern multi-core hardware architectures, we also provide a parallelized implementation of ALGORITHM 2 based on OpenMP threading. Above a certain threshold for m, separate OpenMP tasks are created for subsequent recursive calls. To reduce the synchronization overhead, all threads maintain their local tables M and these are finally reduced inside a critical section.

Since OpenMP is a non-standard extension of the C++ programming language, the parallelization of our implementation is disabled by default. Optionally, it can be enabled by defining the `UNTOU3_ENABLE_OPENMP` preprocessor symbol before the inclusion of the `UNtoU3.h` header file. An OpenMP enabled C++ compiler with the support of OpenMP tasks has to be used then; otherwise, no change is required in the source code that uses the `UNtoU3` class. A number of threads can be controlled in a standard way, e.g., by defining the `OMP_NUM_THREADS` environment variable before running a program.

A sample usage of the `UNtoU3` class with all the optimizations and the parallelization enabled is shown in Figure 2. This program prints out `2168999910`, which is equal to the dimension of the irrep (8). The function `dim` calculates dim[e] for resulting $U(3)$ irreps according to (3).

```
#include <iostream>

// #define UNTOU3_DISABLE_TCE
// #define UNTOU3_DISABLE_UNORDERED
// #define UNTOU3_DISABLE_PRECALC
#define UNTOU3_ENABLE_OPENMP
#include "UNtoU3.h"

unsigned long dim(const UNtoU3<>::U3Weight & irrep) {
   return (irrep[0] - irrep[1] + 1) * (irrep[0] - irrep[2] + 2) * (irrep[1] - irrep[2] + 1) / 2;
}

int main() {
   UNtoU3<> gen;
   gen.generateXYZ(5);
   gen.generateU3Weights(6,1,14);
   unsigned long sum = 0;
   for (const auto & pair : gen.multMap())
      if (auto D = gen.getDimensionality(pair.first))
         sum += D * dim(pair.first);
   std::cout << sum << std::endl;
}
```

Fig. 2. Sample usage of `UNtoU3` class for the $U(21)$ irrep (8) with optimizations and OpenMP parallelization enabled.

6 PERFORMANCE

To evaluate the performance of the above introduced implementations of ALGORITHM 1 and ALGORITHM 2, we conducted experiments on our dual-socket testbed system with two 10-core Intel Xeon E5-2630 v4 Broadwell CPUs. For compilation, we used the GCC C++ compiler (g++) version 5.4.0 wiht `-O2`, `-std=c++11`, and `-DNDEBUG` compilation flags. For experiments with enabled parallelization, we further appended `-fopenmp`.

Within the experiments with ALGORITHM 2, we built programs from the source code shown in Figure 2. We therefore measured the run time of $U(N) \rightarrow U(3)$ reduction for the $U(21)$ irrep (8) followed by the evaluation of resulting $U(3)$ irreps dimensionalities. Commenting out / uncommenting the `UNTOU3_XXX` preprocessor symbol definitions allowed us to focus on particular combinations of enabled/disabled optimizations/parallelization introduced in the text above. In case of ALGORITHM 1, we updated the source code accordingly such that the implementation from `LSU3shell` has been used.

As has been mentioned previously, the measured run times for ALGORITHM 1 and ALGORITHM 2 with both the optimizations and the parallelization disabled were 986.9 and 158.4 seconds, respectively. Next, we measured the impact of particular optimizations as well as their combinations; the results are shown in Table 1. Note that the optimizations alone reduced the run time by a relatively small factors. However, their combinations had in some cases a cumulative effect. For instance, TCE alone reduced the run time from 158.4 to 151.2 seconds, therefore by 4.55 percents only. However, if hash table (Opt. II.) was additionally enabled, then TCE reduced the run time much more significantly from 110.0 to 70.2 seconds, therefore by 36.2 percents. All the 3 optimizations together reduced the run time more than by a factor of 3.

Table 1. Running times of ALGORITHM 2 in seconds for the input $U(21)$ irrep (8) in dependence on enabled (Y) or disabled (N) optimizations measured on the testbed computer system.

Opt. I. (TCE)	Opt. II. (hash table)	Opt. III. (precalc.)	Run time [s]
N	N	N	158.4
Y	N	N	151.2
N	Y	N	110.0
N	N	Y	134.4
Y	Y	N	70.2
Y	N	Y	129.3
N	Y	Y	60.2
Y	Y	Y	50.1

As for the parallelization, the run time with all the optimizations enabled was 4.8 seconds with 20 OpenMP threads. The corresponding speedup was therefore 10.4, which did not imply linear scalability. We attribute this behavior primarily to the overhead of the OpenMP tasking mechanism. To support this conclusion, we also measured the case with parallelization enabled but using only a single OpenMP thread; it lasted 65.5 seconds in contrary to 50.1 seconds when parallelization was disabled.

The final highest speedup of ALGORITHM 2 achieved with all the optimizations without the parallelization was 3.16 with respect to the same algorithm without optimizations and 16.7 with respect to ALGORITHM 1. Additionally, the same speedups with enabled parallelization were 33.0 and 205.6, respectively, on our 20-core testbed machine.

Moreover, we did not focus on a single input irrep only. We also performed measurements for different $U(N)$ irreps—they produced similar speedups as those mentioned above.

7 CONCLUSIONS

The contribution of this article is a new fast algorithm for the calculation of $U(3)$ irreps in an $U(N)$ irrep and its C++ implementation that can run in parallel on multi-core hardware systems. Due to our focus on many-fermion problems (1), we were able to design the algorithm in a highly-efficient way that would not otherwise be possible. We showed that within experiments conducted on 20-core computer system, the new algorithm accelerated the solution of $U(N) \rightarrow U(3)$ problem 16.7 times when run sequentially and more than 200 times when run in parallel in comparison with the original algorithm implemented in the `LSU3shell` software. The new algorithm therefore enables much larger $U(N) \rightarrow U(3)$ problems in applications such as nuclear structure modelling.

ACKNOWLEDGMENTS

The work is supported by the Czech Science Foundation under Grant No. 16-16772S and by the Czech Ministry of Education, Youth and Sports under Grant No. CZ.02.1.01/0.0/0.0/16_019/0000765.

REFERENCES

J. P. Draayer, Y. Leschber, S. C. Park, and R. Lopez. 1989. Representations of U(3) in U(N). *Computer Physics Communications* 56 (1989), 279–290. https://doi.org/10.1016/0010-4655(89)90024-6

T. Dytrych, K.D. Launey, J.P. Draayer, P. Maris, J.P. Vary, E. Saule, U. Catalyurek, M. Sosonkina, D. Langr, and M.A. Caprio. 2013. Collective Modes in Light Nuclei from First Principles. *Physical Review Letters* 111, 25 (2013), 252501. https://doi.org/10.1103/PhysRevLett.111.252501

T. Dytrych, P. Maris, K.D. Launey, J.P. Draayer, J.P. Vary, D. Langr, E. Saule, M.A. Caprio, U. Catalyurek, and M. Sosonkina. 2016. Efficacy of the SU(3) scheme for ab initio large-scale calculations beyond the lightest nuclei. *Computer Physics Communications* 207 (2016), 202–210. https://doi.org/10.1016/j.cpc.2016.06.006

J. P. Elliott. 1958. Collective Motion in the Nuclear Shell Model. I. Classification Schemes for States of Mixed Configurations. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 245, 1240 (1958), 128–145. https://doi.org/10.1098/rspa.1958.0072

Israel M Gelfand and Michael Tsetlin. 1950. Finite-dimensional representations of the group of unimodular matrices. In *Dokl. Akad. Nauk Ser. Fiz.*, Vol. 71. 825–828. English translation in I. M. Gelfand. Collected Papers Vol. II, Springer-Verlag, 1988.

Daniel Langr, Tomáš Dytrych, Tomáš Oberhuber, and František Knapp. 2018. Efficient Parallel Generation of Many-Nucleon Basis for Large-Scale Ab Initio Nuclear Structure Calculations. In *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017)*. Lecture Notes in Computer Science, Vol. 10778. Springer Berlin Heidelberg, 341–350.

Kristina D. Launey. 2017. *Emergent Phenomena in Atomic Nuclei from Large-scale Modeling: A Symmetry-guided Perspective.* World Scientific Publishing Company.