Accelerating many-nucleon basis generation for high performance computing enabled ab initio nuclear structure studies

Daniel Langr  
*Czech Technical University in Prague*

Tomáš Dytrych  
*Nuclear Physics Institute of the Academy of Sciences of the Czech Republic v. v. i.*

Kristina D. Launey  
*Louisiana State University*

Jerry P. Draayer  
*Louisiana State University*

Follow this and additional works at: https://digitalcommons.lsu.edu/physics_astronomy_pubs

**Recommended Citation**  
https://doi.org/10.1177/1094342019838314

This Article is brought to you for free and open access by the Department of Physics & Astronomy at LSU Digital Commons. It has been accepted for inclusion in Faculty Publications by an authorized administrator of LSU Digital Commons. For more information, please contact ir@lsu.edu.
Accelerating many-nucleon basis generation for high performance computing enabled ab initio nuclear structure studies

Daniel Langr¹, Tomáš Dytrych²,³, Kristina D Launey³ and Jerry P Draayer³

Abstract
We present the problem of generating a many-nucleon basis in SU(3)-scheme for ab initio nuclear structure calculations in a symmetry-adapted no-core shell model framework. We first discuss and analyze the basis construction algorithm whose baseline implementation quickly becomes a significant bottleneck for large model spaces and heavier nuclei. The outcomes of this analysis are utilized to propose a new scalable version of the algorithm. Its performance is consequently studied empirically using the Blue Waters supercomputer. The measurements show significant acceleration achieved with over two orders of magnitude speedups realized for larger model spaces.

Keywords
Ab initio, high performance computing, nuclear structure, many-nucleon basis generation, parallel algorithm, symmetry-adapted no-core shell model

1. Introduction
Understanding the origin, structure, and phases of hadronic matter is key to comprehending the evolution of the universe. To fully achieve this, we need to model the complex dynamics of atomic nuclei that control a vast array of astrophysical phenomena and are often found key to understanding processes in extreme environments, from stellar explosions to the interior of nuclear reactors.

A dramatic increase in performance of modern high performance computing architectures advanced capability of achieving first-principles descriptions of light nuclei and their nuclear reactions (Epelbaum et al., 2011; Hagen et al., 2010; Navrátil et al., 2000; Quaglioni and Navrátil, 2008). Over the past two decades, first-principles numerical studies of nuclei and nuclear reactions have been routinely facilitating leadership class of high performance computing systems. To extend the reach of ab initio methods toward heavier nuclei, we have developed a novel method dubbed symmetry-adapted no-core shell model (SA-NCSM) (Dytrych et al., 2013) and implemented it as a suite of highly scalable computer codes¹ (Dytrych et al., 2016). Our approach is to solve the Schrödinger equation for a many-nucleon quantum system interacting via realistic interactions that are tied to the underlying quark/gluon considerations. The solution to this problem is achieved by finding the lowest lying eigenstates and eigenvalues of the nuclear Hamiltonian matrix. Matrix elements are computed in a many-nucleon basis that spans a physically relevant subspace of the nuclear Hilbert space, the so-called model space. Eigenvalues of the nuclear Hamiltonian matrix correspond to the nuclear binding energies. The associated eigenvectors are subsequently utilized for computing nuclear observables to compare with experiment and make predictions. This step involves vector–matrix–vector multiplications, where large sparse matrices are associated with physical operators.

The SA-NCSM framework utilizes many-nucleon basis states of a fixed parity, consistent with the Pauli principle, and limited by a many-body basis cutoff $N_{\text{max}}$. This cutoff

¹ Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
² Nuclear Physics Institute, Czech Academy of Sciences, Prague, Czech Republic
³ Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA, USA

Corresponding author:
Daniel Langr, Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Prague 6, Czech Republic.
Email: daniel.langr@fit.cvut.cz
is defined as the maximum number of harmonic oscillator (HO) quanta allowed in a many-nucleon basis state above the minimum for a given nucleus. The dimension of the basis increases combinatorially as a function of $N_{\text{max}}$ and the number of nucleons, which is illustrated in Figure 1.

The SA-NCSM approach adopts powerful mathematical algorithms of group theory (Draayer et al., 1989) to organize model space according to SU(3) and SU(2) symmetry groups that dominate the low-energy nuclear dynamics (Elliott, 1958a, b; Elliott and Harvey, 1963; Rosensteel and Rowe, 1977). This, in turn, allows us to reduce computationally intractable model spaces to a smaller number of physically relevant configurations based on their deformations and proton, neutron, and total intrinsic spins.

Construction of sparse matrices of physical operators involves two sets of many-nucleon basis states—one associated with matrix rows and the second one with matrix columns. Let us denote their dimensions by $m$ and $n$, respectively. The number of nonzero matrix elements is then $O(m \times n)$, where both $m$ and $n$ increase in a combinatorial manner with the number of nucleons and the model space cutoff parameter $N_{\text{max}}$. This increase implies enormous requirements for memory space and computational power to carry out calculations beyond light nuclei and low values of $N_{\text{max}}$. Such resources are provided by, for example, the Blue Waters system, operated by the National Center for Supercomputing Applications. Its main non-GPU part contains 362,240 CPU cores and 1.382 PB of memory in total, which we utilize to a full extent in our large-scale SA-NCSM calculations.

The SA-NCSM workflow consists of the following three major steps:

1. generation of a many-nucleon basis for a given model space,
2. assembly of a matrix operator, and
3. calculation of its eigenvalues and eigenvectors.

For large $N_{\text{max}}$ cutoffs and medium mass nuclei, the first step became a major bottleneck. For example, it would take hours on the Blue Waters system to generate the $N_{\text{max}} = 12$ basis of $^{20}\text{Ne}$ nucleus. In this article, we first present the SU(3) classification scheme, describe many-nucleon basis construction process and model space selection rules adopted by the SA-NCSM framework (Section 3). Consequently, we proceed to analyze the baseline basis generation algorithm (Section 4), and, based on these analyses, we present a new accelerated basis generation algorithm and prove its superiority in the experimental study conducted on the Blue Waters system (Section 5).

This article is an extended version of our conference paper (Langr et al., 2018), where we introduced the accelerated basis generation algorithm and presented its performance measurements carried out using mostly our test-bed system. Here, we additionally:

- describe the SU(3)-scheme many-nucleon basis and associated symmetry-based selection rules that specify an SA-NCSM model space (Section 3);
- provide a description of many-nucleon basis construction utilized by the SA-NCSM framework (Section 3);
- present both the baseline and the accelerated basis construction algorithms in agreement with notation and terminology introduced by Section 3; and
- report and discuss more thorough experimental results obtained on one of the most powerful supercomputers in the world.

2. Related work

The choice of the many-nucleon basis is inherently coupled with the physics of low-energy nuclear dynamics. The traditional approach is to adopt $m$-scheme basis, that is, to construct the many-nucleon basis as antisymmetrized products of HO wave functions. This approach underpins, for example, two highly scalable NCSM codes MFDn (Maris et al., 2010) and BigStick (Johnson et al., 2013). A detailed description of the parallel generation of $m$-scheme basis is provided by Sternberg et al. (2008).

Since Hamiltonian is a rotationally invariant operator, its eigenstates are characterized by the total angular momentum $J$. Naturally, as an alternative choice to the $m$-scheme, one can construct a basis composed of states carrying $J$ as a good quantum number, the so-called $J$-coupled basis (Aktulga et al., 2011).

In contrast to all other research groups, our approach is to adopt the SU(3)-scheme basis; see Section 3 for details.

3. Method

3.1. Construction of many-nucleon basis states

The SA-NCSM adopts powerful mathematical algorithms of group theory to decompose a given model space in terms of SU(3) × SU(2) irreps, where the spatial part of basis is classified by the SU(3) × SO(3) group chain. Thereby,
The dashed line divides single-shell (left) and inter-shell (right) quantum numbers.

For example, \( \Omega \), \( \lambda \), and \( (\eta_1, \eta_2) \) describe spherical, prolate, and oblate deformation, respectively. The quantum number \( S \) denotes the total intrinsic spin. It is coupled with the total orbital angular momentum \( L \) to the total angular momentum \( J \) and its projection \( M \). The multiplicity label \( \kappa \) reflects the fact that multiple occurrences of \( L \) are possible within a generic irrep \( (\lambda_0 \mu_0) \) of \( SU(3) \).

The symbols \( i_p \) and \( i_n \) schematically denote a set of additional quantum numbers needed to uniquely determine proton and neutron \( SU(3) \times SU(2) \) irreps, that is, distribution of protons and neutrons over the major HO shells and their single-shell and inter-shell labels as illustrated in Figure 2. Namely, in each HO shell \( \eta \) \( (\eta = 0, 1, 2, \ldots) \) with \( A_\eta > 0 \) nucleons, we construct antisymmetric \( U(\Omega_\eta) \times SU(2) \) irreps, where \( \Omega_\eta = (\eta + 1)(\eta + 2)/2 \) is degeneracy of the major HO shell \( \eta \). Next, we reduce \( U(\Omega_\eta) \) with respect to \( SU(3) \) to obtain the single-shell labels \( \alpha_\eta(\lambda_0 \mu_0)S_{\alpha_\eta} \), where index \( \alpha_\eta \) is required to distinguish multiple occurrences of \( SU(3) \) irrep \( (\lambda_0 \mu_0) \) in a given \( U(\Omega_\eta) \) irrep. The bracket \( (i_p, i_n) \) denotes coupling of proton and neutron irreps into the final proton–neutron irrep \( (\lambda_\mu)S \). The symbol \( \rho \) denotes a multiplicity label which is needed to distinguish between multiple occurrences of the \( (\lambda_\mu) \) irrep within the product of coupling.

The algorithm of basis construction starts by generating all possible distributions of nucleons over the major HO shells, limited by the number of HO quantum excitations up through \( N_{\text{ex}} \). Next, for each major HO shell \( \eta \) with \( A_\eta \) nucleons, we utilize the powerful mathematical algorithm for enumerating single-shell \( SU(3) \times SU(2) \) irreps (Drayer et al., 1989). We store resulting single-shell irreps in a lookup data structure indexed by \( \eta \) and the number of particles \( A_\eta \). Consequently, this table along with the list of proton and neutron distributions is being used for the construction of proton and neutron \( SU(3) \times SU(2) \) irreps by coupling the single-shell irreps as illustrated in Figure 2. The resulting quantum labels of proton and neutron irreps are stored in data structures that are mapped to \( i_p \) and \( i_n \) indices. The final step is to couple proton and neutron irreps into many-nucleon irreps while applying selection rules to determine whether they belong to given model space. This last step is the most time-consuming phase of the basis construction and Section 4 focuses on its acceleration.

### 3.2. Selection rules

Conventionally, the model space is defined by the \( N_{\text{max}} \) cutoff parameter, that is, one typically restricts many-nucleon basis by considering all many-nucleon configurations that carry up to \( N_{\text{max}} \) total number of HO quanta. As can be seen from Figure 1, the dimensionality of the nuclear model space becomes rapidly computationally prohibitive as the number of active nucleons and the cutoff parameter \( N_{\text{max}} \) increases. Symmetries underpinning many-nucleon basis states of the SA-NCSM approach play a crucial role in the process of selecting the physically relevant portion of the model space. In particular, for each space of nuclear configurations with \( N_{\text{ex}} \) total HO quanta, we provide a list of selected \( SU(3) \times SU(2) \) irreps. That is, we specify \( SU(3) \) quantum numbers along with proton, neutron, and total intrinsic spins \( S_{p}, S_{n}, \) and \( S \)

\[
N_{\text{ex}} \to \{ (\lambda_1 \mu_1)S_{p}, S_{n}, S_{1}; (\lambda_2 \mu_2)S_{p}, S_{n}, S_{2}, \ldots \}
\]

An empty list signifies that we include all configurations with \( N_{\text{ex}} \) total HO excitation quanta. A set of parameters for each \( N_{\text{ex}} \leq N_{\text{max}} \) considered is provided in the input file and constitutes the model space definition of physically relevant model space for target nuclei.

---

**Figure 2.** Example of proton and neutron \( SU(3) \times SU(2) \) irreps and all additional quantum numbers needed for their full specification. The dashed line divides single-shell (left) and inter-shell (right) quantum numbers.
4. Algorithms

Originally, our implementation of the SA-NCSM was built upon distributed memory parallelism, namely upon the Message Passing Interface (MPI) library interface (Gropp et al., 1998; Snir et al., 1998). The disadvantage of this approach is that all MPI processes running on the same computational node keep a large amount of redundant auxiliary data in their memory spaces during the matrix construction, which is the most time-demanding phase of the calculation. We therefore transformed the matrix construction code to exploit the hybrid distributed-/shared-memory parallelism, namely, the combination of MPI processes and OpenMP threading (Chapman et al., 2007), which allowed us to share large portions of data among threads running on the same node. Consequently, this integration of multi-threading resulted in a significant reduction of the memory requirements for auxiliary data structures as well as the runtime of the matrix construction phase.

These memory and runtime savings along with the computational power of leadership HPC systems, represented by the Blue Waters supercomputer, opened up new possibilities to study light and medium mass nuclei using the SA-NCSM approach. At the same time, the original (baseline) single-threaded basis generation, which stemmed directly from the methodology introduced in Section 3, quickly became the main application bottleneck.

4.1. Mapping of matrices and basis states to MPI processes

The SA-NCSM approach maps matrices to MPI processes in a two-dimensional checkerboard fashion, where a matrix is split into $N \times N$ blocks. Each block is assigned to a single MPI process. In case of symmetric matrices, such as Hamiltonian operators, only triangular part of a matrix is stored in computer memory and mapped to MPI processes. The total number of processes is then equal to $P = N(N + 1)/2$; this concept is illustrated in Figure 3 for $N = 5$ and therefore 15 MPI processes in total with ranks 0 to 14. Each row/column of processes can be identified by $I$ and $J$ zero-based coordinates, respectively. Let $p_{I,J}$ denotes the MPI process with the row coordinate $I$ and the column coordinate $J$. In each row/column of processes, there is a single process with a special role; we call this process principal. For symmetric matrix operators, the principal processes are mapped to diagonal matrix blocks ($I = J$).

There are two sets of many-nucleon basis states—the “vertical” one that spans matrix rows and the “horizontal” one that spans matrix columns. According to the rules of matrix-process mapping, the vertical and horizontal basis sets are split into $N$ parts. Each MPI process then needs to construct its local $I$ th part of the vertical basis and $J$ th part of the horizontal basis.

$N$ is therefore an important parameter for SA-NCSM runs. It is derived from the total number of MPI processes and it determines the number of distinct sets of states the bases are split into. Each MPI process is then assigned one such set from the vertical basis and one from the horizontal basis.

Simple splitting of basis states into $N$ parts and their mapping to MPI processes would lead to poor load balancing due to irregular structures of nonzero matrix elements observed, for example, in Figure 3. We therefore assign small segments of basis states to MPI processes in a round-robin fashion (see, e.g. Kleinrock, 1976), where $k$ th segment is mapped to an MPI process with row/column coordinate $k \mod N$. Each segment is defined by all many-nucleon SU(3) × SU(2) irreps obtained by coupling a particular pair of proton and neutron irreps associated with $i_p$ and $i_n$ indices. This implies that matrix operators are handled with permuted rows and columns with respect to $(i_p, i_n)$ pairs. We further say that the $(i_p, i_n)$ combinations and corresponding basis states mapped to a particular MPI process are process-local from its perspective.
Algorithm 1. GenerateBasis(modelSpace, N, K).

Input: modelSpace: given model space
Input: N: number of basis blocks, i.e., principal MPI processes
Input: K: index of basis block computed by actual MPI process
Output: dims[], pBasisIpIn[], wpn[], blockEnd[], firstStateInBlock[]: arrays with calculated basis data

1. \( i_p, i_n \leftarrow \text{function-of-modelSpace} \)
2. \( \text{numberOfStates} \leftarrow 0 \)
3. \( \text{ipInPair} \leftarrow 0 \)
4. \( \text{dims}[0] \leftarrow N - 1 \leftarrow 0 \)
5. for \( i_p \leftarrow 0 \) to \( i_p^{\text{max}} \) do
   6. for \( i_n \leftarrow 0 \) to \( i_n^{\text{max}} \) do
      7. \( N_{ee} \leftarrow \text{function-of-(}i_p, i_n)\)
      8. if \( N_{ee} \) does not belong to modelSpace then continue
      9. \( \text{SelectionRules} \leftarrow \text{function-of-(modelSpace, } N_{ee}) \)
     10. \( K' \leftarrow \text{function-of-} (\text{ipInPair}, N) \)
     11. if \( \text{SelectionRules} \) contains allowed spins then
         // current \( (i_p, i_n) \) pair is valid but "process-global"*/
         12. calculate data for current \( (i_p, i_n, \text{SelectionRules}) \)
         13. \( \text{dims}[K'] \leftarrow \text{dims}[K'] + \text{function-of-data} \)
         14. if \( K' = K \) then
             // current \( (i_p, i_n) \) pair is valid and "process-local"*/
             15. append \( (i_p, i_n) \) at the end of \( pBasisIpIn[] \)
             16. append \( \text{function-of-data} \) at the end of \( wpn[] \)
             17. append \( \text{function-of-data} \) at the end of \( \text{blockEnd[]} \)
             18. append \( \text{numberOfStates} \) at the end of \( \text{firstStateInBlock[]} \)
             19. \( \text{numberOfStates} \leftarrow \text{numberOfStates} + \text{function-of-data} \)
     20. \( \text{ipInPair} \leftarrow \text{ipInPair} + 1 \)

4.2. Baseline algorithm

The pseudocode of the baseline basis generation algorithm is given in Algorithm 1. This pseudocode does not cover the preparatory steps but merely the last step (see Section 3), which represents by far the most time-demanding part of the whole basis generation process. Algorithm 1 is run twice by each MPI process—once for the vertical basis and once for the horizontal basis—while process coordinates \( I \) and \( J \) are used as arguments for the algorithm parameter \( K \).

There is no MPI communication involved, since each process has a complete list of proton and neutron SU(3) \( \times \) SU(2) irreps allowing it to generate complete many-nucleon basis of a given model space.

In the pseudocode, we omit computational details which are irrelevant to the problem described in this article. To denote functional dependencies between particular algorithm parts, we use the simplified syntax with the "function-of-" prefix, which generally means that some data are calculated in dependence on another data. We also use the square brackets suffix "[]" to denote that some data are stored as an array. Such arrays are generally considered to be dynamic, that is, they are initially empty and can grow by appending additional elements.

Algorithm 1 iterates over all possible pairs of proton and neutron SU(3) \( \times \) SU(2) irreps represented by indices \( i_p \) and \( i_n \). Some of these pairs are filtered out by the selection rules; see lines 8 and 11 of Algorithm 1. We call the pairs that are not filtered-out valid. After applying the selection rules, the algorithm decides which valid pairs belong to which MPI processes (line 10). If the pair belongs to the current MPI process (line 14), then it is process-local. Valid process-local \( (i_p, i_n) \) pairs are stored into \( pBasisIpIn[] \) array and the corresponding calculated basis data are appended at the end of arrays \( wpn[], \text{blockEnd[]}, \) and \( \text{firstStateInBlock[]} \). Here, the array \( wpn[] \) holds many-nucleon SU(3) \( \times \) SU(2) irreps \( (\lambda \mu)S \) that are the product of \( (i_p, i_n) \) coupling and satisfy the applied selection rules (line 12). The array \( \text{blockEnd[]} \) stores the total number of irreps generated by \( (i_p, i_n) \) coupling. Finally, the array \( \text{firstStateInBlock[]} \) holds the location of the first many-nucleon basis state from a segment given by \( (i_p, i_n) \) coupling. Moreover, there is an additional array \( \text{dims[]} \) with \( N \) elements, which is calculated by all processes; this array contains the dimension of each block of basis states. The total dimension of the entire model space is a sum of all elements in this array.

Two main causes of inefficiency may be observed in Algorithm 1. First, the array \( \text{dims[]} \) is redundantly calculated by all MPI processes. To evaluate elements of \( \text{dims[]} \), some data need to be calculated first for each valid \( (i_p, i_n) \) pair (line 12), which is a costly operation. Second, there is no threading in Algorithm 1. One might observe that the whole iterative process (lines 5 and 6) is inherently sequential. Namely, the development of each iteration depends on the outcome of all the previous iterations for the following reasons:

1. The distribution of valid \( (i_p, i_n) \) pairs to processes is a function of \( \text{ipInPair} \) (line 10), which represents its
order among all valid pairs (line 20). There is no way how to find the value of $ipInPair$ for particular $(i_p, i_n)$ directly.

2. The calculated basis data are appended into corresponding arrays (lines 15–18). With direct loop OpenMP parallelization, the order of insertions of elements into these arrays would therefore change, which is not acceptable.

4.3. Accelerated algorithm

We proposed and introduce here the accelerated version of Algorithm 1; its pseudocode is presented by Algorithm 2 and 3. The first optimization represents the elimination of redundant calculation of the $dims[]$ array by all MPI processes. In the accelerated algorithm, only the principal processes evaluate their contributions to the $dims[]$ array, that is, $dims[K]$. The whole array is finally reduced and distributed to all processes by MPI_Allreduce communication operation. Such an approach significantly reduces the amount of costly data calculations, which now needs to be performed only for $(i_p, i_n)$ pairs that are local to each MPI process.

The second optimization incorporates efficient parallelization in the form of multithreading into Algorithm 1. The accelerated algorithm iterates over $(i_p, i_n)$ pairs three times; however, all three outer loops over $i_p$ indices are now performed in parallel by all OpenMP threads (lines 6, 17, and 35). Our solution is generic such that it allows to use OpenMP dynamic loop scheduling, which was found necessary for balanced computational load among threads. The parallelization in the accelerated algorithm works as follows:

---

**Algorithm 2.** GenerateBasisParallel(modelSpace, N, K)—Part I.

---

**Input:** modelSpace: given model space
**Input:** $N$: number of basis blocks, i.e., principal MPI processes
**Input:** $K$: index of basis block computed by actual MPI process
**Output:** $dims[]$, $pnBasisIn[]$, $wpn[]$, $blockEnd[]$, $firstStateInBlock[]$: arrays with calculated basis data

```c
1 $i_p^\max$, $i_n^\max$ ← function-of-modelSpace
2 numberOfStates ← 0
3 ipInPair ← 0
4 $dims[0 : N - 1]$ ← 0
5 $ipPairsForIp[0 : i_p^\max]$ ← 0
6 for $i_p$ ← 0 to $i_p^\max$ do in parallel /* first loop */
7 for $i_n$ ← 0 to $i_n^\max$ do
8     $N_{ex}$ ← function-of-($i_p$, $i_n$)
9     if $N_{ex}$ does not belong to modelSpace then continue
10    SelectionRules ← function-of-($modelSpace$, $N_{ex}$)
11    if SelectionRules does not contain allowed spins then continue
12     $ipPairsForIp[i_p] ← ipPairsForIp[i_p] + 1$ /* current $(i_p, i_n)$ is valid and “process-global” */
13 $firstIpInPairForIp ←$ parallel exclusive prefix sum over $ipPairsForIp$
14 localIpInPairsForIp[0 : $i_p^\max$] ← 0
15 localWpnsForIp[0 : $i_p^\max$] ← 0
16 locNumStatesForIp[0 : $i_p^\max$] ← 0
17 for $i_p$ ← 0 to $i_p^\max$ do in parallel /* second loop */
18 for $i_n$ ← 0 to $i_n^\max$ do
19     $N_{ex}$ ← function-of-($i_p$, $i_n$)
20     if $N_{ex}$ does not belong to modelSpace then continue
21    SelectionRules ← function-of-($modelSpace$, $N_{ex}$)
22    if SelectionRules does not contain allowed spins then continue
23     $K' ←$ function-of-($ipInPair$, $N$)
24     ipInPair ← ipInPair + 1
25     if $K'$ $\neq K$ then continue
26     /* current $(i_p, i_n)$ is valid and “process-local” */
27      calculate data for current $(i_p, i_n, SelectionRules)$
28      localIpInPairsForIp[i_p] ← localIpInPairsForIp[i_p] + 1
29      localWpnsForIp[i_p] ← localWpnsForIp[i_p] + function-of-data
30      locNumStatesForIp[i_p] ← locNumStatesForIp[i_p] + function-of-data
31 localFirstIpInPairForIp ← parallel excl. prefix sum over localIpInPairsForIp
32 localFirstWpnsForIp ← parallel exclusive prefix sum over localWpnsForIp
33 locNumStatesBeforeIp ← parallel exclusive prefix sum over locNumStatesForIp
34 resize $pnBasisIn$, $firstStateInBlock$, $wpn$, and $blockEnd$ properly
```

Langr et al. 527
1. Within the first loop, the number of valid \((i_p, i_n)\) pairs for each \(i_p\) is stored into the temporary \(ipInPair\) array (line 12). Next, the exclusive parallel prefix sum is run over this array resulting in a new temporary array \(firstIpInPair\). The element \(firstIpInPair[i_p]\) therefore equals the number of valid \((i_p, i_n)\) pairs for all \(0 \leq i_p < i_p^{\text{max}}\).

2. Within the second loop, we can now evaluate \(ipInPair\) for each \(i_p\) independently (and therefore concurrently; lines 18 and 25). In this loop, the number of valid process-local \((i_p, i_n)\) pairs for each \(i_p\) is stored into the temporary array \(localIpInPair\) (line 28). Similarly, the number of \(wpn\) elements and the number of states for each \(i_p\) are stored into temporary arrays \(localWpnsForIp\) and \(locNumStatesForIp\), respectively (lines 29 and 30). As a next step, parallel prefix sums over these arrays are performed (lines 31–33). The results of these prefix sums then allow us to properly resize the resulting basis arrays (line 34) and for each \(i_p\) find where to store generated basis data.

3. Within the third loop, the basis data for process-local \((i_p, i_n)\) pairs are finally stored into corresponding arrays (lines 48–57).

An additional advantage of the proposed solution is that data are not appended into arrays; instead, they are written at already-known positions. This avoids memory reallocations, which, when performed frequently, might considerably hinder scalability of multithreaded programs.

Finally, to prevent redundant calculations within loops, we integrated several (thread-local) software cache data structures into the code. These are not shown in the pseudocode of the accelerated algorithm; however, they contribute to its higher efficiency as well.

The first of these caches contains the selection rules for each allowed \(\text{Nex}\). This cache data structure is constructed in the first loop at line 11 and reused in the second and third loops at lines 23 and 44. In the code, it is implemented as an array (indexed by \(\text{Nex}\)) of maps (associative arrays), where keys are pairs of \(\text{Sp}\) and \(\text{Sn}\), proton and neutron spin quantum numbers, and values contain a vector of selected spins along with a vector of \(\text{SU}(3)\) quantum numbers selected for each spin. Note that maps from the C++ Standard Library are typically implemented as some form of a binary search tree (such as a red-black tree implemented in GNU libstdc++).

The second cache is constructed in the second loop and contains the results of coupling of two \(\text{SU}(3)\) irreps. It is implemented as an array (indexed by \(\text{Nex}\)) of maps (associative arrays), where keys are pairs of \(\text{Sp}\) and \(\text{Sn}\), proton and neutron spin quantum numbers, and values contain a vector of selected spins along with a vector of \(\text{SU}(3)\) quantum numbers selected for each spin. Note that maps from the C++ Standard Library are typically implemented as some form of a binary search tree (such as a red-black tree implemented in GNU libstdc++).

The second cache is constructed in the second loop and contains the results of coupling of two \(\text{SU}(3)\) irreps. It is implemented as a map, where keys consist of a pair of \(\text{SU}(3)\) labels and values contain a dynamic array of resulting \(\text{SU}(3)\) irreps.

### 5. Experiments

To evaluate the proposed accelerated basis generation algorithm, we carried out experiments where we compared its performance with the algorithm baseline implementation.

```plaintext
for i_p ← 0 to i_p^{\text{max}} do in parallel
    ipInPair ← firstIpInPairForIp[i_p]
    localIpInPair ← localFirstIpInPairForIp[i_p]
    wpnIndex ← localFirstWpnForIp[i_p]
    numberOfStates ← locNumStatesBeforeIp[i_p]
for i_n ← 0 to i_n^{\text{max}} do
    if N_{ex} does not belong to modelSpace then continue
    SelectionRules ← function-of-(modelSpace, N_{ex})
    if SelectionRules does not contain allowed spins then continue
    K' ← function-of-(ipInPair, N)
    ipInPair ← ipInPair + 1
    if K' \neq K then continue
    pnBasisIpIn[localIpInPair] ← (ipInPair, i_n)
    firstStateInBlock[localIpInPair] ← numberOfStates
    calculate data for current (i_p, i_n, SelectionRules)
    if process is principal then dims[K] ← dims[K] + function-of-data
    wpnCount ← function-of-data
    write function-of-data to wpn[wpnIndex : wpnIndex + wpnCount - 1]
    wpnIndex ← wpnIndex + wpnCount
blockEnd[localIpInPair] ← function-of-data
numberOfStates ← numberOfStates + function-of-data
localIpInPair ← localIpInPair + 1
reduce dims across all processes (by using MPI_Allreduce)
```
For these experiments, we utilized the Blue Waters supercomputer, namely its main non-GPU part. It is based on 22,636 Cray XE6 nodes, each node containing 2 AMD 8-core Bulldozer CPUs and 64 GB of memory. Each Bulldozer physical core is further exposed as 2 virtual cores under certain conditions (so-called clustered multithreaded: CMT).

Blue Waters is a Cray supercomputer running the Cray Compiling Environment (CCE). Within this environment, we used the GNU programming environment module (PrgEnv-gnu) version 5.2.82 for compilation, which encapsulated the GNU C++ compiler (g++) version 4.9.3 through the cc compiler script. For program compilation and build, we enabled C++14 support (-std=c++14), OpenMP (-fopenmp), and a full optimization level with vectorization (-O3).

Within our calculations on Blue Waters, we typically set the scheduler to run 1 MPI process per computational node. Each MPI process is then split by the OpenMP runtime into 32 threads, since our SA-NCSM implementation partially satisfies the conditions of CMT. This MPI/OpenMP configuration turned out to provide the lowest overall application wall time. Consequently, calculations with Hamiltonian operator matrices imply the following facts:

1. Both vertical and horizontal basis span the same model space.
2. Only the upper triangular part of a symmetric Hamiltonian matrix operator is stored in memory and mapped to P MPI processes.
3. P and N must be chosen such that \( P = N(N+1)/2 \). Moreover, N has to be an odd number due to the restrictions imposed by the used eigensolver (Maris et al., 2010).
4. The maximum number of MPI processes used for a single calculation on Blue Waters can be therefore set to \( P = 22366 \), which corresponds to \( N = 211 \).

It would be absurdly wasteful to use almost all the nodes of Blue Waters just to measure and compare the performance of our basis generation algorithms. Within the presented experiments, we therefore exploited the so-called simulation mode provided by our SA-NCSM implementation. Simulation mode uses a setup for a real calculation (model space and \( P \)) but runs only a single MPI process \( p \). This process then generates its local bases parts the same way as if all \( P \) processes were running in parallel. Such an approach allowed us to evaluate the basis generation algorithms using only a single Blue Waters node.

The question is whether such single-node/single-process experiments reflect the algorithm performance that would be observed in normal (nonsimulated) SA-NCSM calculations. As for the basis generation itself, the simulation mode differs from the normal application runs in two aspects:

1. Within normal runs, the basis generation runtime is driven by the “slowest” MPI process due to load-balancing issues, whereas we report the runtime of a single \( p_{1,J} \) process only.
2. The final MPI all-reduction operation is omitted.

In what follows, we show that both these aspects have only a negligible impact on basis generation runtime and, consequently, that our measurements predict behavior of basis generation algorithms in large-scale SA-NCSM production runs. The basis generation runtime is very well balanced across all MPI processes and the presented experiments are therefore insensitive to the choice of \( p_{1,J} \) for the simulation mode (see Section 5.2 for details).

5.1. Results and discussion

In the reported runtimes, we always considered the overall generation of both sets of basis states (namely their process-local parts) including all their steps introduced in Section 3. The measurements were performed by instrumenting our SA-NCSM framework source code, namely by the inclusion of time measurements for program parts that were of our interest.

Recall that within production runs, we always set the number of threads per MPI process to 32. However, to measure threading scalability, the number of threads per process became a parameter in our experiments; we denote it by \( T \).

First, we simulated the utilization of the whole Blue Waters machine (\( N = 211 \)) and compared runtimes of the baseline and accelerated basis generation algorithms for different nuclei as a function of increasing \( N_{max} \), see Figure 4. These measurements themselves do not show how much was the accelerated algorithm faster compared to the baseline algorithm, therefore, we also show the speedup (ratio of runtimes) in Figure 5. It is obvious that the accelerated algorithm reduced the basis generation runtime dramatically, which was our initial goal. Moreover, the speedup was not constant; it grew with both the number of nucleons and \( N_{max} \), which implies that the speedup is
generally higher for large model spaces. This represents a very positive feature of the accelerated basis generation algorithm, since medium mass nuclei are nowadays the primary target of our studies.

Next, we measured the intra-node strong scalability (fixed problem size) of the accelerated basis generation algorithm as a function of the number of per-process OpenMP threads $T$; the results are shown in Figure 6. For sake of comparison, we also show the runtimes of the sequential baseline algorithm, which are naturally insensitive to the number of threads used. The measured speedup of the accelerated algorithm between 32 and 1 thread was around 8.

Figure 6 also shows which portion of the overall speedup of the accelerated algorithm with respect to the baseline algorithm was caused by parallelization (multithreading) and which portion was caused by other optimizations discussed in Section 4. Even when the accelerated algorithm was run with a single-thread only ($T = 1$), the basis generation runtime was reduced by a factor around 21. This acceleration was mainly caused by removing the need of calculating data for all the $(i_p, i_a)$ pairs and also by introducing software cache databases that prevented redundant calculations in subsequent algorithm loops (see the end of Section 4 for details).

In the next experiment, we measured the basis generation runtime as a function of increasing $N \in \{1, 5, 11, 33, 55, 111, 211\}$, which in the simulation mode reflects the basis generation runtime for the increasing number of MPI processes $P \in \{1, 15, 66, 561, 1540, 6216, 22366\}$ that would be used in a production run for modeling $^{12}$C nucleus and $N_{\text{max}} = 12$. The results are shown in Figure 7 for the baseline algorithm and the accelerated algorithm runs with 1 and 32 threads. The baseline version was almost insensitive to $N$, which, again, may be explained by redundant calculation of data for all the $(i_p, i_a)$ pairs because of the independent construction of the whole $\text{dims}[\cdot]$ array by each MPI process. On the contrary, the accelerated algorithm reduced the runtime significantly with the growing $N$, both in single-threaded and multithreaded algorithm runs.

Recall that in the baseline basis-generation algorithm, each MPI process iterates over all $(i_p, i_a)$ indices only once, while in the accelerated algorithm three times. This brings into the accelerated algorithm an additional overhead that does not depend on $N$. The effects of this overhead can be observed in Figure 7; in the corresponding measurements, the runtime of the accelerated algorithm did not decrease linearly with the growing $N$ (e.g. between $N = 55$ and $N = 211$, the runtime reduction in case of $T = 32$ was only 20%) and for $T = 1$ and $N = 1$, the accelerated algorithm was slower (of around 35%) than the baseline algorithm.

In the final experiment, we show the runtime of particular phases of the accelerated basis generation algorithm as a function of three different combinations of $N$ and the per-process number of threads $T$; see Figure 8. Recall that for production runs using the Blue Waters system, the maximum feasible value of $N$ is 211 and the maximum number of threads per MPI process $T$ is 32; both of these maximal values were taken into account in the measurements. We show runtimes of all three loops of the accelerated algorithm, where subsequent parallel prefix sums are also included. The results indicate that the first loop is very fast if multithreading is exploited and its runtime is practically insensitive to $N$, which was expected, since this loop iterates over all the $(i_p, i_a)$ combinations, not just those local to an actual MPI process. However, this does not hold for the second and third loop, therefore their runtimes increased inversely proportionally to $N$. As for threading, all three loops benefited from parallelization.
process could be split among all processes and their threads. Theoretically, the whole basis-generation algorithm does not reflect the number of MPI processes generated basis-generation algorithm with respect to the base-line algorithm does not represent an issue. For example, it takes just 5.39 s on Blue Waters system to generate $3.43 \times 10^9 (J = 0)$ basis states spanning complete $N_{max} = 8$ model space of $^{20}$Ne.

An alternative approach would be to generate the basis with $N$ MPI processes, store it into files, and then read particular basis parts from these files by all processes would require additional redistribution of generated basis states according to the rules introduced in Section 4.1 to guarantee further load-balanced calculation of matrix operator elements. Such redistribution would involve all-to-all communication operations with complex communication patterns. All-to-all operations typically take a non-negligible amount of time especially when the majority of computational nodes of a supercomputer are involved. For illustration, we reported some measurements based on $\text{MPI\_Alltoallv}$ communication operation on Blue Waters and it took hundreds of seconds when a large number of nodes were used (Langr et al., 2014a). Redistribution of basis states would therefore introduce an additional overhead and possible bottleneck into the basis generation process.

In our approach, each process generates its basis states locally, thereby avoiding a redistribution of basis states among processes. At the same time, this implies lower scalability. However, as we dramatically reduced the basis generation time to a negligible amount even for largest model spaces attainable on contemporary and next-generation supercomputers, such a lower scalability does not represent an issue. For example, it takes just 5.39 s on Blue Waters system to generate $3.43 \times 10^9 (J = 0)$ basis states spanning complete $N_{max} = 8$ model space of $^{20}$Ne.

5.2. Simulation mode

As mentioned earlier, we conducted experiments in the simulation mode of our SA-NCSM implementation where only a single MPI process ran on a single Blue Waters node. To prove the prediction accuracy of such an approach, we measured the load balance of the accelerated basis generation process over 100 randomly selected MPI processes $p_{i,j}$ out of 22,366 in case of $N = 211$ for the problem of $^{20}$Ne, $N_{max} = 10$, $T = 32$. The average (mean) value of the basis generation runtime was 53.03 s, while the maximum and minimum values were 53.86 and 52.67 s, respectively. Our mapping of basis states to MPI processes with respect to segments (see Section 4) is clearly well balanced and the measurements made within the simulation mode with a single MPI process reflects the runtime of the overall basis generation in real-world SA-NCSM studies with the relative error of no more than few percents.

Also, the simulation mode does not include the runtime of the MPI all-reduce communication operation. However, such a reduction over an array of size $N$ is in practice very fast. Namely, on Blue Waters, it takes up to few seconds even if the majority of the nodes are involved (Langr et al., 2014b). Since our accelerated basis generation algorithm was developed to enable SA-NCSM studies of large model spaces, for which the whole application runtime takes hours, the effect of this MPI operation is in practice insignificant.

5.3. Additional considerations

One might argue that the measured speedup of our accelerated basis-generation algorithm with respect to the baseline algorithm does not reflect the number of MPI processes and their threads. Theoretically, the whole basis-generation process could be split among all $P$ processes and their $T$ threads, which in the largest calculations on Blue Waters would result in 715,712 threads running in total in parallel. The speedups observed in the presented experiments were several orders of magnitude lower (see Figure 5). On the other hand, splitting the basis generation among all the processes would require additional redistribution of generated basis states according to the rules introduced in Section 4.1 to guarantee further load-balanced calculation of matrix operator elements. Such redistribution would involve all-to-all communication operations with complex communication patterns. All-to-all operations typically take a non-negligible amount of time especially when the majority of computational nodes of a supercomputer are involved. For illustration, we reported some measurements based on $\text{MPI\_Alltoallv}$ communication operation on Blue Waters and it took hundreds of seconds when a large number of nodes were used (Langr et al., 2014a). Redistribution of basis states would therefore introduce an additional overhead and possible bottleneck into the basis generation process.

In our approach, each process generates its basis states locally, thereby avoiding a redistribution of basis states among processes. At the same time, this implies lower scalability. However, as we dramatically reduced the basis generation time to a negligible amount even for largest model spaces attainable on contemporary and next-generation supercomputers, such a lower scalability does not represent an issue. For example, it takes just 5.39 s on Blue Waters system to generate $3.43 \times 10^9 (J = 0)$ basis states spanning complete $N_{max} = 8$ model space of $^{20}$Ne.

An alternative approach would be to generate the basis with $N$ MPI processes, store it into files, and then read particular basis parts from these files by all $P$ MPI processes. In fact, this is the approach that we used before we implemented the accelerated basis generation algorithm. However, similarly to using a communication network, accessing a parallel file system would introduce an additional overhead that might become non-negligible for a large number of MPI processes/computational nodes. And, again, since we have reduced the basis generation times for model spaces attainable in SA-NCSM calculations to such small amounts as mentioned above, there is no need to employ the file system-based approach.

Finally, note that we presented the strong scalability results (fixed problem size) but did not present the weak scalability measurements (fixed problem size per processing unit). The reason is that the space of input parameters for SA-NCSM calculations is highly coarse-grained in the sense that even the smallest change in any of the parameters (number of protons, number of nucleons, $N_{max}$) results in a large change in the number of basis states. It was therefore not possible to maintain the fixed problem size per processing units (threads/processes) in our measurements.

6. Conclusions

We introduced the $\text{SU}(3) \times \text{SU}(2)$ symmetry-based classification scheme of the many-nucleon basis, which is utilized by the SA-NCSM framework for ab initio modeling of atomic nuclei, along with the selection rules that are used for downselecting the model space into a small set of physically relevant basis states. We also described the main
steps for the basis construction and implementation details of selection rules. Next, we introduced an efficient scalable algorithm for the generation of this basis starting from a set of proton and neutron SU(3) × SU(2) irreps and a given set of selection rules that define a model space. Our implementation based on the hybrid MPI + OpenMP parallel programming model speeded up the basis generation by two orders of magnitude for large model spaces on Blue Waters.

For illustration, we measured the runtime of the application steps for 20Ne and 22O full-scale calculations performed with the accelerated basis-generation algorithm on Blue Waters by using its 22,366 nodes (N = 211). In both cases, more than 95% of runtime was in both cases spent in the matrix assembly step, around 4% in the eigensolver step, and only less than 0.2% in the basis generation step. Consequently, we reduced the basis generation times for model spaces attainable on contemporary and next-generation supercomputers to a negligible amount in the context of the whole SA-NCSM calculations. The proposed algorithm eliminates the basis-generation bottleneck that have heretofore hindered applications of SA-NCSM approach for ab initio modeling of important collective and cluster nuclear states in medium- and light-mass nuclei.

Declaration of Conflicting Interests
The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding
The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Czech Science Foundation under grant no. 16-16772S, by the Czech Ministry of Education, Youth and Sports under grant no. CZ.02.1.01/0.0/0.0/16_019/0000765, and by the National Science Foundation under award numbers ACI-1713690 and OIA-1738287. This work is part of the Blue Waters sustained-petascale computing project, which is supported by NSF (awards OCI-0725070 and ACI-1238993) and the state of Illinois (Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications).

ORCID iD
Daniel Langr † https://orcid.org/0000-0001-9760-7068

Notes
1. https://sourceforge.net/projects/lsu3shell/ (In the time of writing this article, the latest updates were included in the LSU3develop repository branch.)
2. Generally, the matrix can be split into M × N blocks where M ≠ N. For sake of simplicity, we assume M = N in this text which covers most applications of our SA-NCSM framework and also allows us to present the text in a more concise and legible form.
3. J is a standard symbol for denoting the total angular momentum. However, to maintain consistency with our conference paper (Langr et al., 2018), we use J in this article also for denoting a process column coordinate. These two meanings should be clearly distinguishable from the context at each particular place.
4. Within our implementation, this step is represented by the Reshuffle member function of the C++ class called CnscmSU3xSU2Basis.

References


Author biographies

Daniel Langr is research assistant professor at the Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague. His research interests include sparse matrix computations, parallel sorting algorithms, efficient implementations of parallel algorithms, and tuning codes for large supercomputers.

Tomáš Dytrych is researcher at the Nuclear Physics Institute of the Czech Academy of Sciences and adjunct assistant professor of physics at Louisiana State University. His research interests include physics of atomic nuclei, computational group theory, and application of modern programming techniques for high performance computing of nuclear structure.

Kristina D Launey is assistant professor of physics at Louisiana State University. Her research focuses on strong interaction physics ranging from modeling atomic nuclei from first principles, through electroweak-and strong-interaction driven nuclear reactions, to further understanding the nuclear interaction from the underlying foundation of Quantum Chromodynamics.

Jerry P Draayer is Roy P. Daniels professor of physics at Louisiana State University and the president of the Southeastern Universities Research Association. His research interest include study of the low-energy structure of atomic nuclei from first principles by combining group theory, many-body, and high performance computing methodologies.