Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

The Abdus Salam
International Centre
for Theoretical Physics

Sparse matrix computations with common libraries for HPC

México, February 2018

Marlon Brenes
brenesnm@tcd.ie
https://www.tcd.ie/Physics/research/groups/qusys/people/navarro/

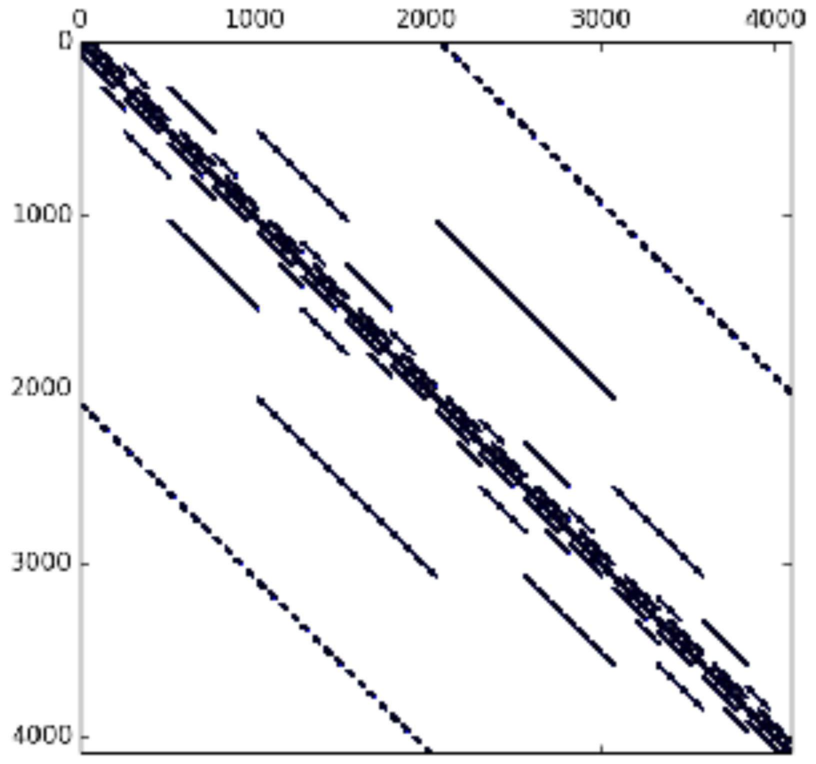
Outline

- Sparse matrices
- Representation of sparse matrices
    - CSR format
- PETSc and SLEPc
    - Example of usage
- Case of study: Large scale simulations of unitary dynamics of quantum systems

Sparse matrices

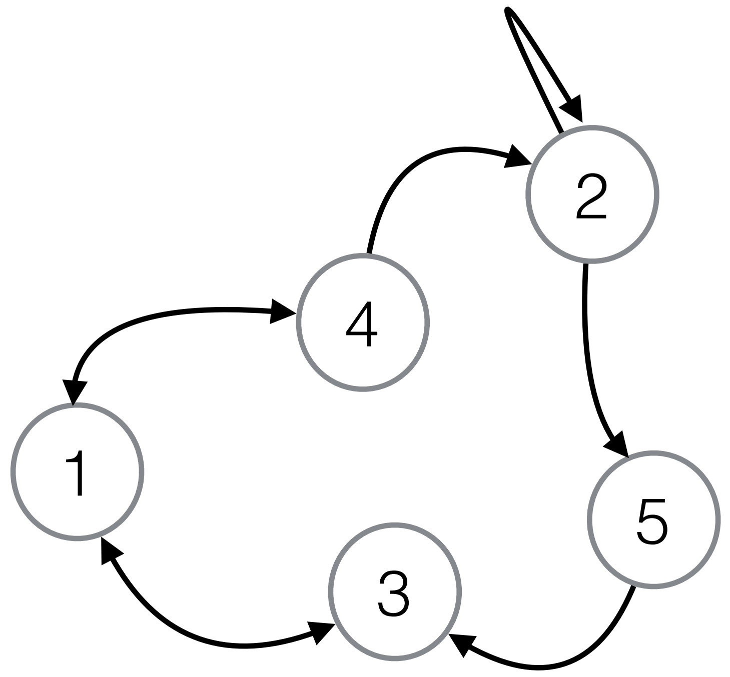
Sparse matrices (SM)

- A SM is a matrix for which **most** of the elements are **zero**
- It is sufficient for the number of non-zero entries to be of order $O(n)$ for the matrix to be considered sparse
- As opposed to this, a dense matrix contains $O(n^2)$ non-zero entries



Sparse matrices (SM)

- Conceptually, a sparse matrix is an object that represents a system with $O(n)$ *connections or interactions*



$$\Longrightarrow \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Representation of sparse matrices

- The objective of a sparse matrix representation is to store only non-zero elements of the matrix
- There are several ways to represent a sparse matrix
- Each representation provides benefits depending on the type of operations to be computed on the matrix

Representation of sparse matrices

- Many types:
  - Coordinate format (COO)
  - Compressed row format (CSR)
  - Compressed column format (CSC)
  - Diagonal format (DIA)
  - More…
- Performance trade-offs based on operations, lookups and increments
- We will focus on CSR

CSR sparse format (compressed sparse row)

$$\begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 1 & 0 \\ 0 & 0 & 6 & 1 \end{pmatrix}$$

- Represents the matrix using **three one-dimensional arrays**
- Fast matrix-vector products, efficient arithmetic operations and row-slicing
- Changes in sparsity are **expensive**, slow column slicing

CSR sparse format (compressed sparse row)

$$\begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 1 & 0 \\ 0 & 0 & 6 & 1 \end{pmatrix}$$

- Can be represented as follows:

$$\text{value} = [\text{nz values sorted by row}]$$

$$\text{c\_row} = [0, \ldots, \text{c\_row}[\text{i}] = \text{c\_row}[\text{i - 1}] + \text{nnz on row (i - 1)}]$$

$$\text{column} = [\text{column index}]$$

CSR sparse format (compressed sparse row)

$$\begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 1 & 0 \\ 0 & 0 & 6 & 1 \end{pmatrix}$$

- Can be represented as follows:

$$\text{values} = [1\ \ 3\ \ 1\ \ 2\ \ 4\ \ 1\ \ 6\ \ 1]$$

$$\text{c\_row} = [0\ \ 2\ \ 4\ \ 6\ \ 8]$$

$$\text{column} = [0\ \ 2\ \ 1\ \ 2\ \ 1\ \ 2\ \ 2\ \ 3]$$

Parallel CSR sparse format (compressed sparse row)

$$\begin{array}{r|cccc|l}
\text{Proc 0} & 1 & 0 & 3 & 0 & \\
 & 0 & 1 & 2 & 0 & \\ \hdashline
\text{Proc 1} & 0 & 4 & 1 & 0 & \\ \hdashline
\text{Proc 2} & 0 & 0 & 6 & 1 & \cdots \\
 & & & \vdots & & \ddots
\end{array}$$

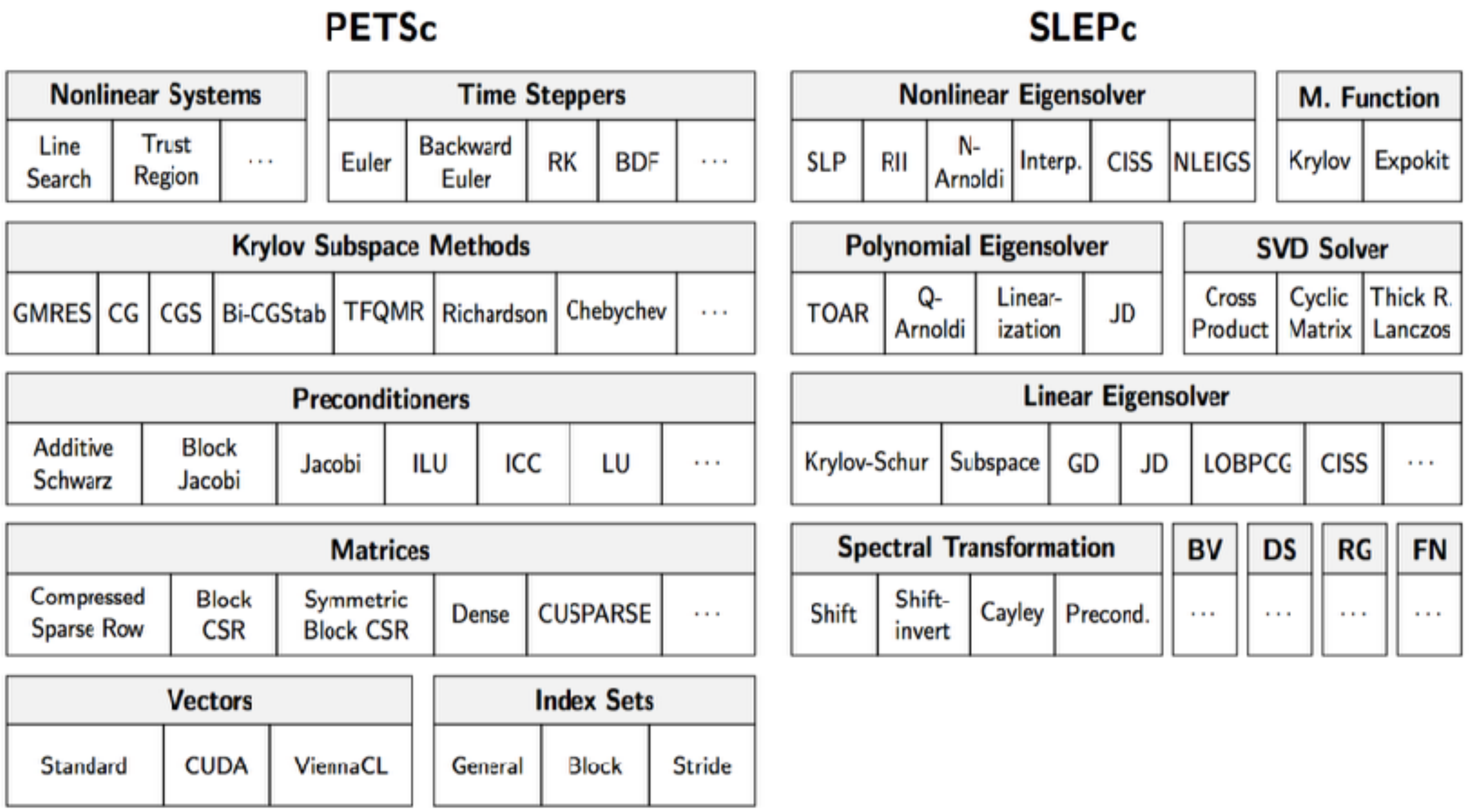
PETSc/SLEPc

- HPC library developed by Argonne National Lab
    - Devoted to sparse matrix operations with massive parallelism in mind
    - Pure distributed memory parallelism
- Highly tested, solid community of developers and users (over 25 years of development)
- Profiling tool, clean API
- Low-lying linear algebra operations can be linked using different libraries (Intel MKL, for instance)
- SLEPc extension for eigenvalue problems (depends on PETSc)

PETSc/SLEPc

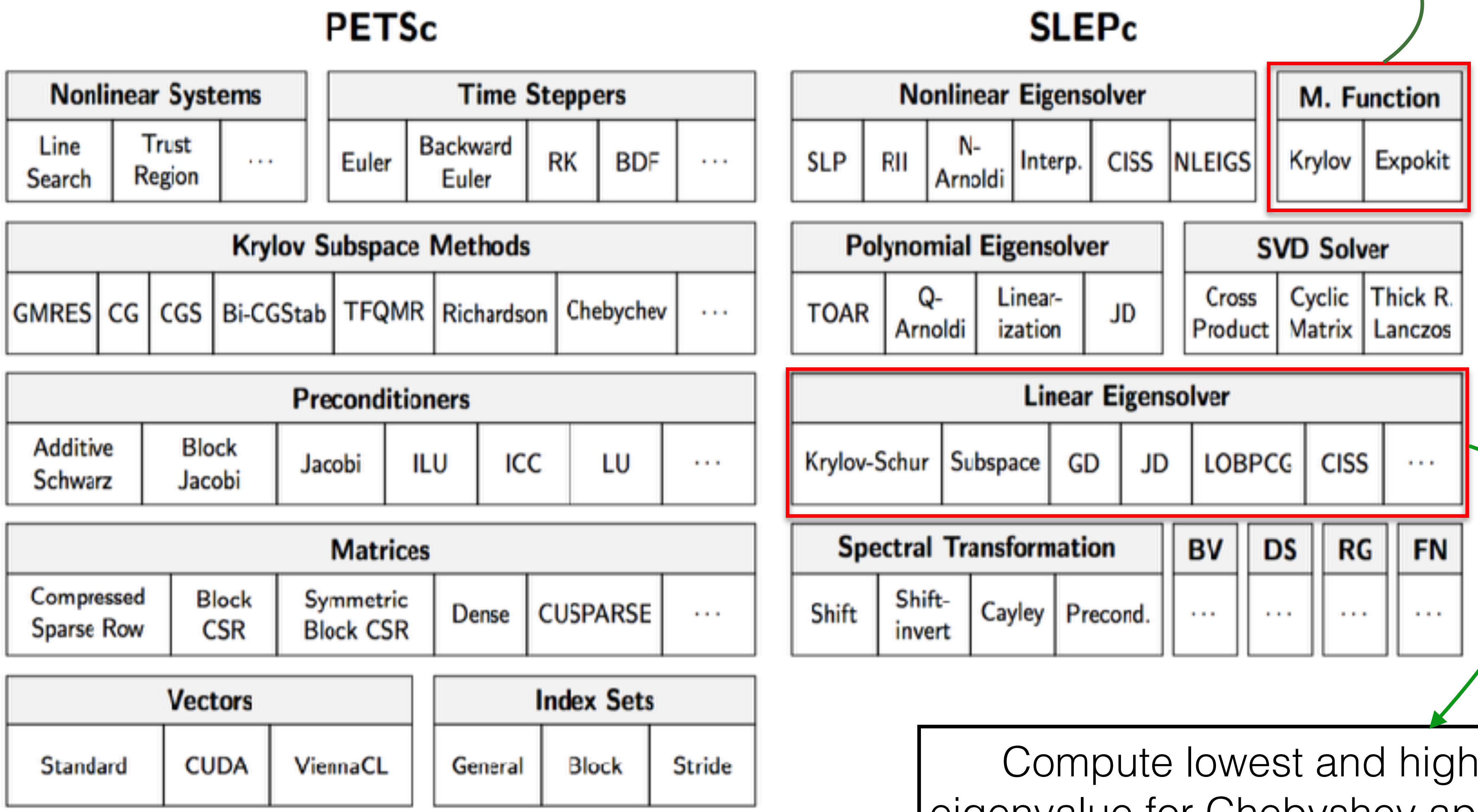
- Pro:
  - MPI is hidden from the user
- Con:
  - MPI is hidden from the user

PETSc/SLEPc


PETSc
Nonlinear Systems
Line Search
Trust Region
...
Time Steppers
Euler
Backward Euler
RK
BDF
...
Krylov Subspace Methods
GMRES
CG
CGS
Bi-CGStab
TFQMR
Richardson
Chebychev
...
Preconditioners
Additive Schwarz
Block Jacobi
Jacobi
ILU
ICC
LU
...
Matrices
Compressed Sparse Row
Block CSR
Symmetric Block CSR
Dense
CUSPARSE
...
Vectors
Standard
CUDA
ViennaCL
Index Sets
General
Block
Stride
SLEPc
Nonlinear Eigensolver
SLP
RII
N-Arnoldi
Interp.
CISS
NLEIGS
M. Function
Krylov
Expokit
Polynomial Eigensolver
TOAR
Q-Arnoldi
Linear-ization
JD
SVD Solver
Cross Product
Cyclic Matrix
Thick R. Lanczos
Linear Eigensolver
Krylov-Schur
Subspace
GD
JD
LOBPCG
CISS
...
Spectral Transformation
Shift
Shift-invert
Cayley
Precond.
BV
...
DS
...
RG
...
FN
...

PETSc/SLEPc

Krylov subspace method for matrix functions



Compute lowest and highest eigenvalue for Chebyshev approach

Steps towards using PETSc for sparse operations

1. Initialise PETSc environment
2. Allocate memory for sparse matrix
3. Initialise matrix
4. Assemble matrix
5. Allocate and initialise initial vector using same matrix parallel distribution
6. Assemble vectors
7. Call functionality
8. Free memory
9. Close PETSc environment

Allocating memory for a parallel sparse matrix in PETSc

- Critical step for performance reasons
- One has to compute the number of non-zero elements that the matrix will contain, or provide an *estimation*
- *If this step is omitted, there's a huge performance penalty related to dynamic increase/resize memory buffers*

Allocating memory for a parallel sparse matrix in PETSc

$$\begin{array}{r} \\ \text{Proc 0} \\ \\ \text{Proc 1} \\ \text{Proc 2} \end{array} \left(\begin{array}{cc:c:c} 1 & 0 & 3 & 0 \\ 0 & 1 & 2 & 0 \\ \hdashline 0 & 4 & 1 & 0 \\ \hdashline 0 & 0 & 6 & 1 \end{array}\right)$$

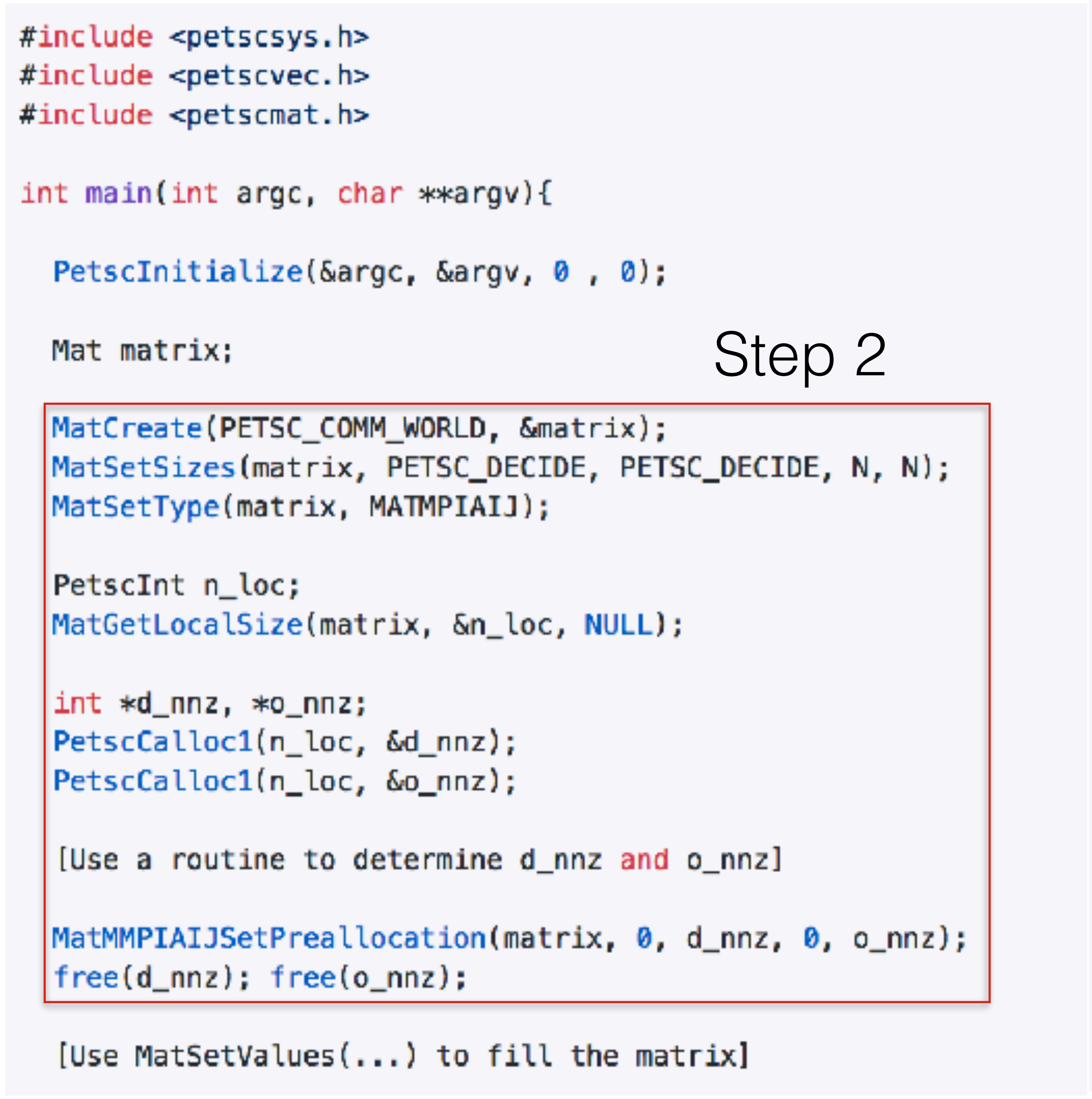
Allocation buffers

	d_nnz	o_nnz
Proc 0	[1 1]	[1 1]
Proc 1	[1]	[1]
Proc 2	[1]	[1]

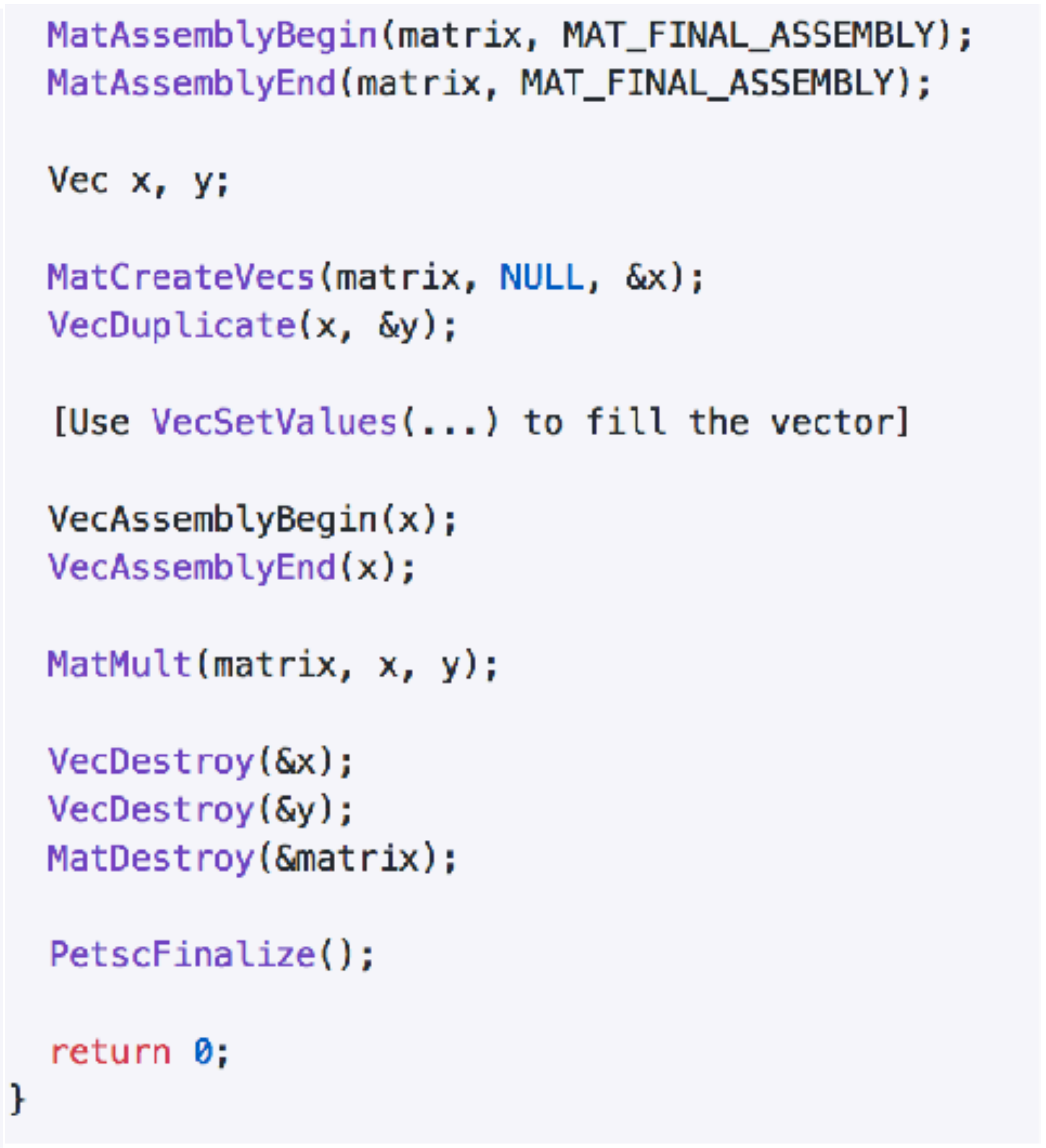
Step 1

```c
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]

  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

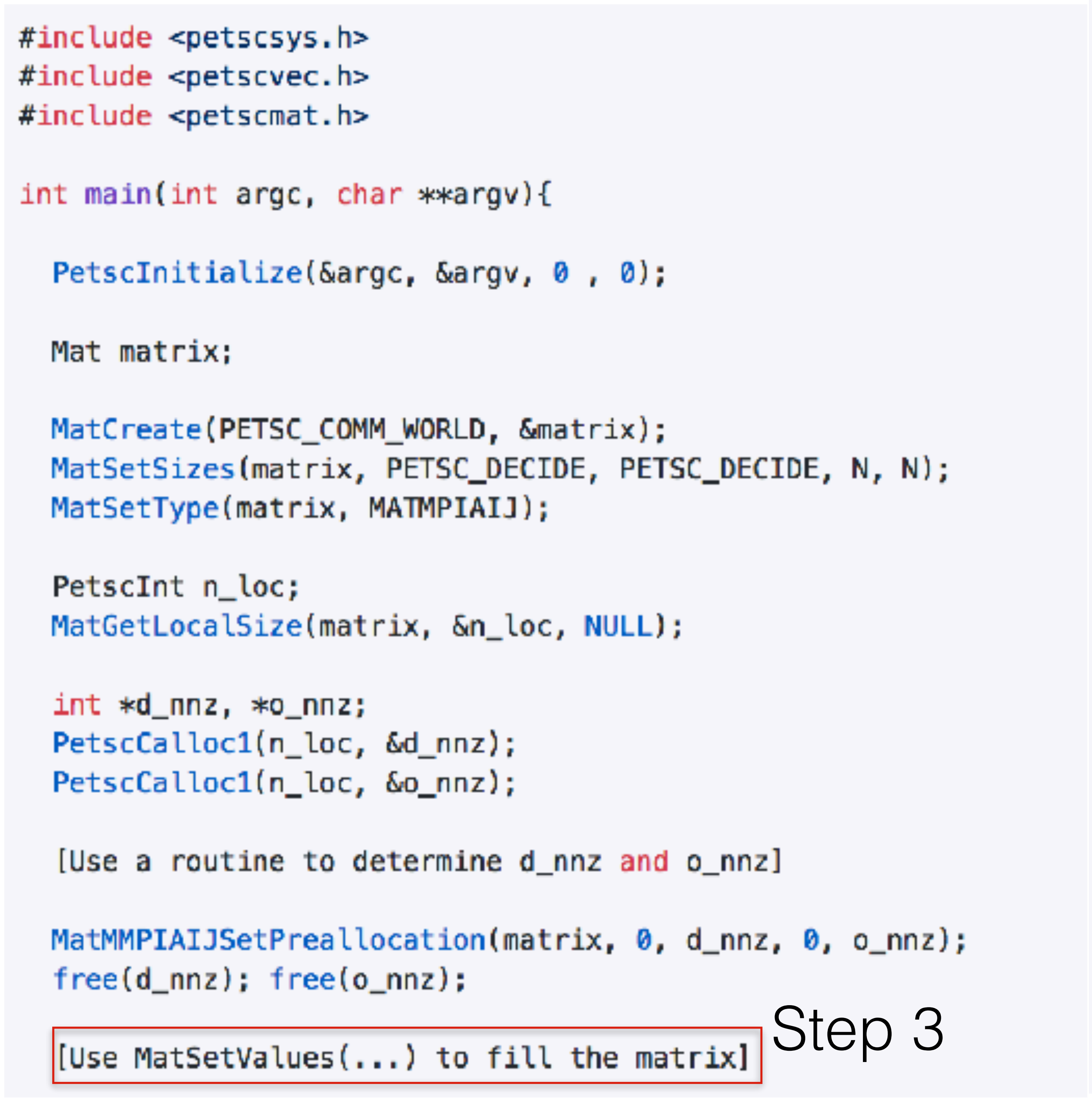
Step 2



```c
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]
```

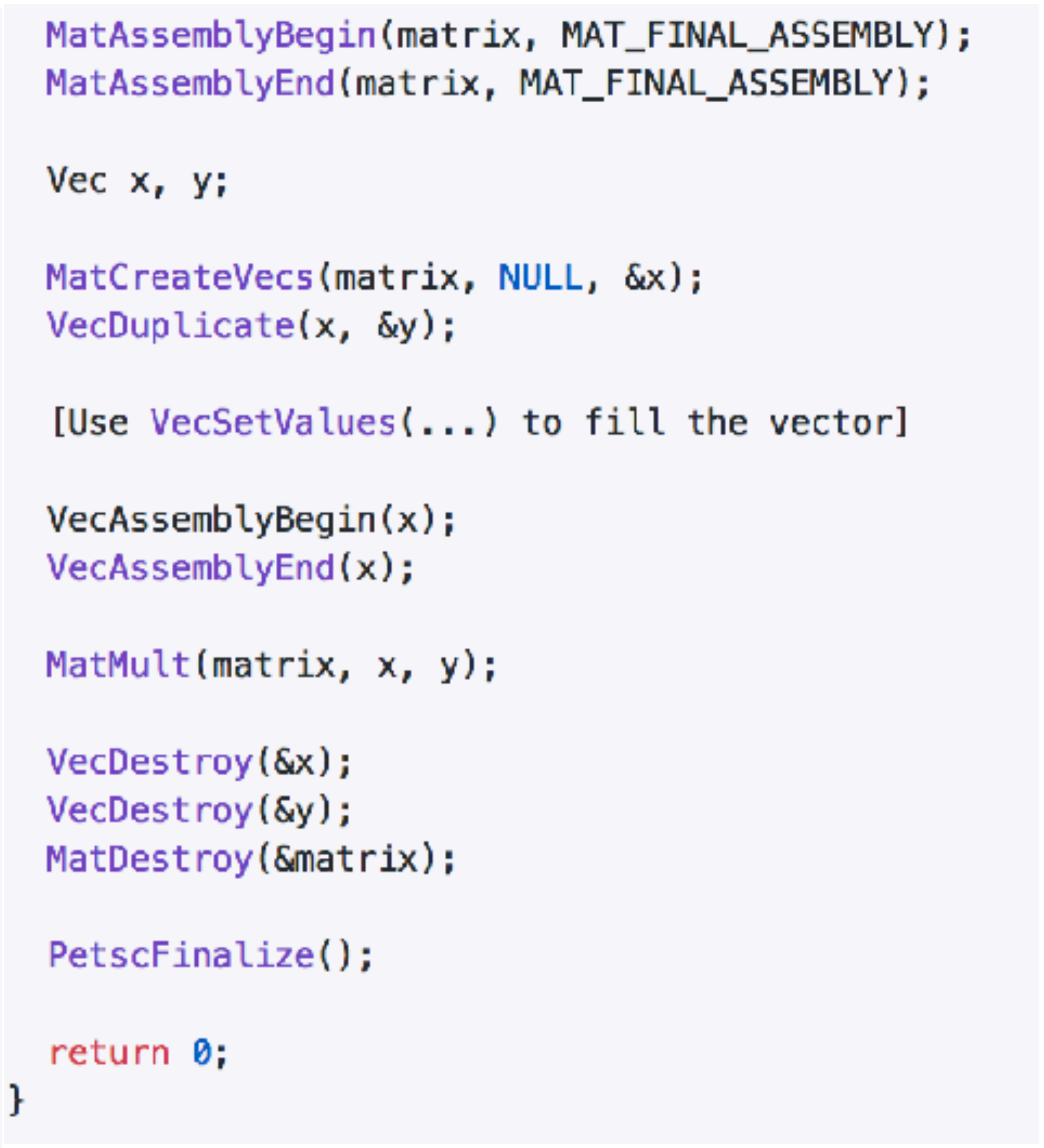


```c
  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```



```
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]
```

Step 3

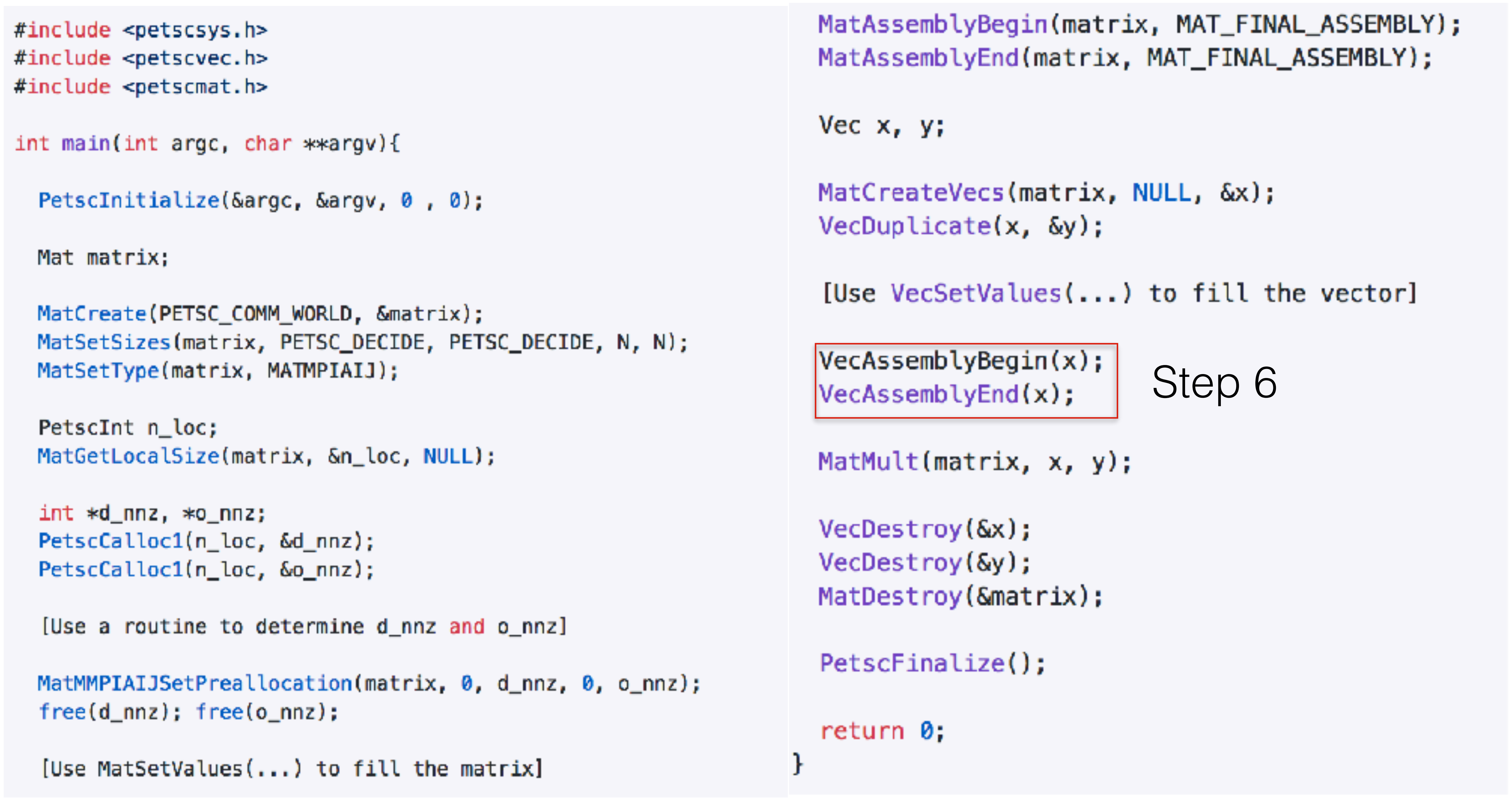


```
  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

```
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]
```

```
  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

Step 4



```
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]

  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

Step 6

```c
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]

  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

Step 7

```
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]
```

```
  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

Step 8

```c
#include <petscsys.h>
#include <petscvec.h>
#include <petscmat.h>

int main(int argc, char **argv){

  PetscInitialize(&argc, &argv, 0 , 0);

  Mat matrix;

  MatCreate(PETSC_COMM_WORLD, &matrix);
  MatSetSizes(matrix, PETSC_DECIDE, PETSC_DECIDE, N, N);
  MatSetType(matrix, MATMPIAIJ);

  PetscInt n_loc;
  MatGetLocalSize(matrix, &n_loc, NULL);

  int *d_nnz, *o_nnz;
  PetscCalloc1(n_loc, &d_nnz);
  PetscCalloc1(n_loc, &o_nnz);

  [Use a routine to determine d_nnz and o_nnz]

  MatMMPIAIJSetPreallocation(matrix, 0, d_nnz, 0, o_nnz);
  free(d_nnz); free(o_nnz);

  [Use MatSetValues(...) to fill the matrix]

  MatAssemblyBegin(matrix, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(matrix, MAT_FINAL_ASSEMBLY);

  Vec x, y;

  MatCreateVecs(matrix, NULL, &x);
  VecDuplicate(x, &y);

  [Use VecSetValues(...) to fill the vector]

  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

  MatMult(matrix, x, y);

  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&matrix);

  PetscFinalize();

  return 0;
}
```

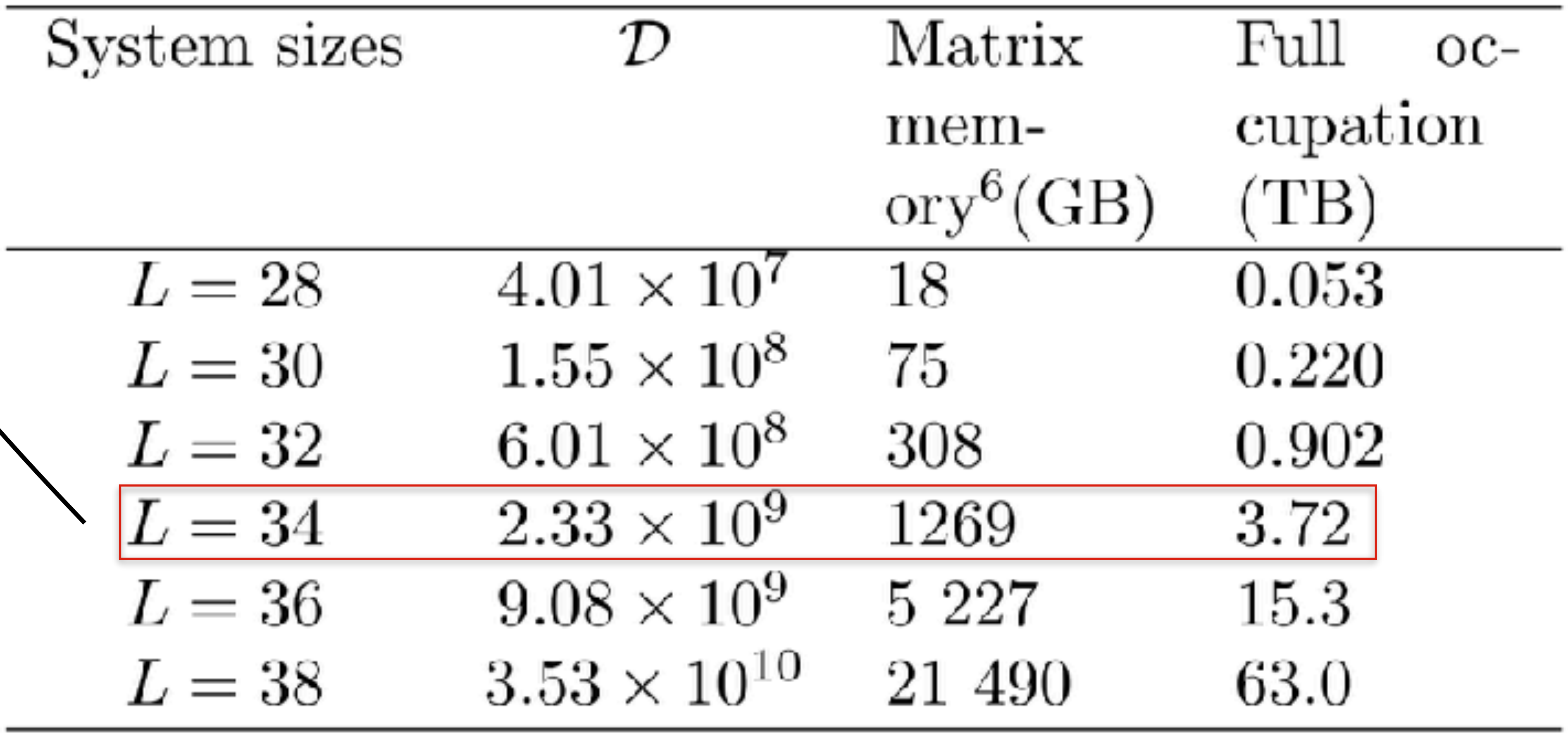
Step 9

Case of study: Large scale simulations of unitary dynamics of quantum systems

$$\begin{tikzcd}
 & \text{Solution: HPC!} & \\
 & \text{Problems when this matrix gets large!} \arrow[u] & \\
i\hbar\frac{\partial}{\partial t}|\Psi(\mathbf{r},t)\rangle = \hat{H}|\Psi(\mathbf{r},t)\rangle \arrow[ur, bend left] \arrow[d] & & \mathcal{K}_m \\
|\Psi(t)\rangle = e^{-i\hat{H}t/\hbar}|\Psi(t=0)\rangle \arrow[rr] & & \mathbf{w}(t) = e^{t\mathbf{A}}\mathbf{v} \arrow[u] \arrow[uul, bend right]
\end{tikzcd}$$

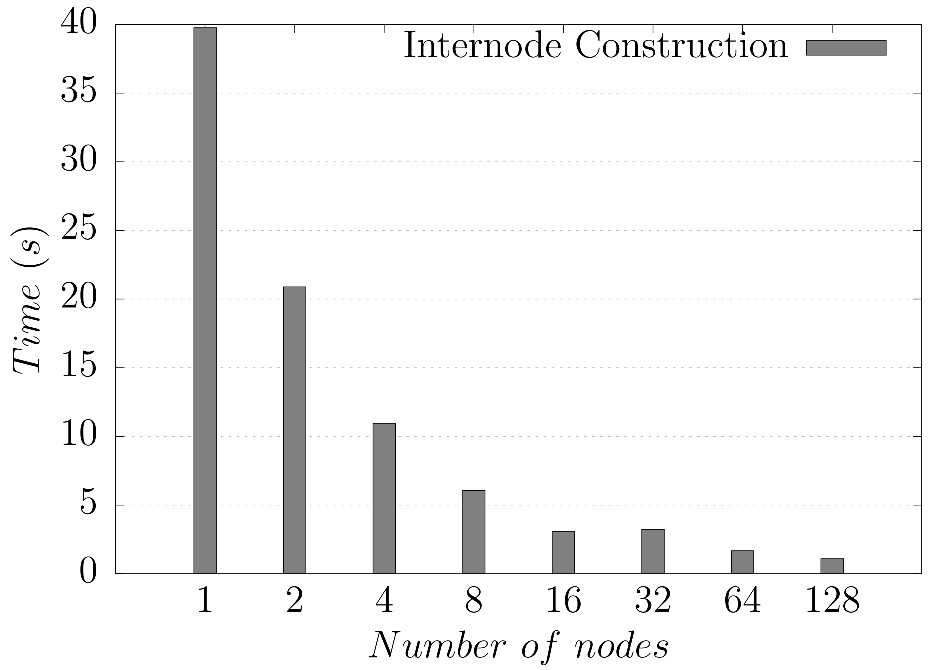
Case of study: Large scale simulations of unitary dynamics of quantum systems

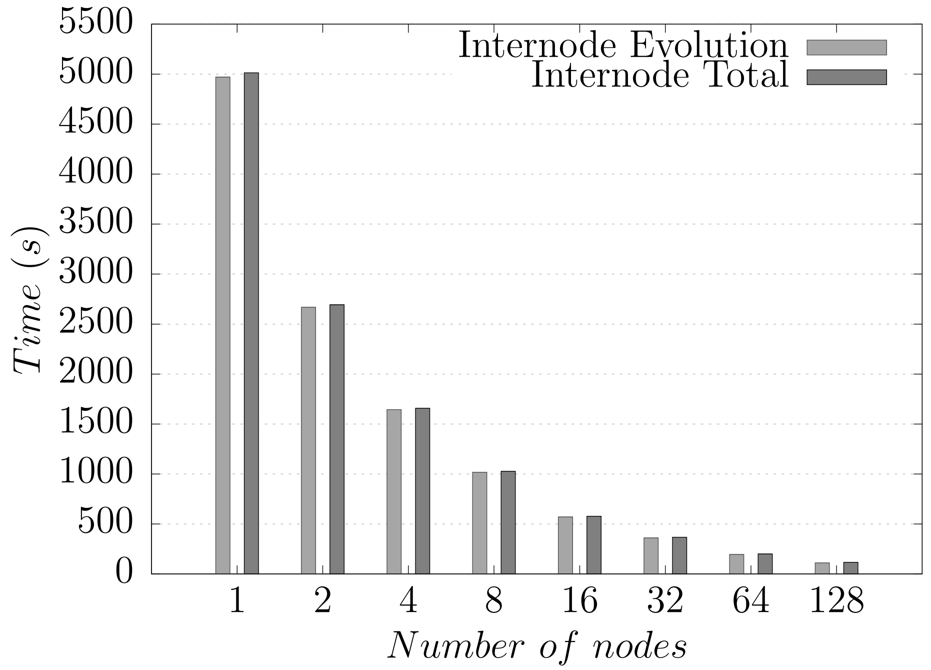
Indexing of rows overflows 32-bit integers!

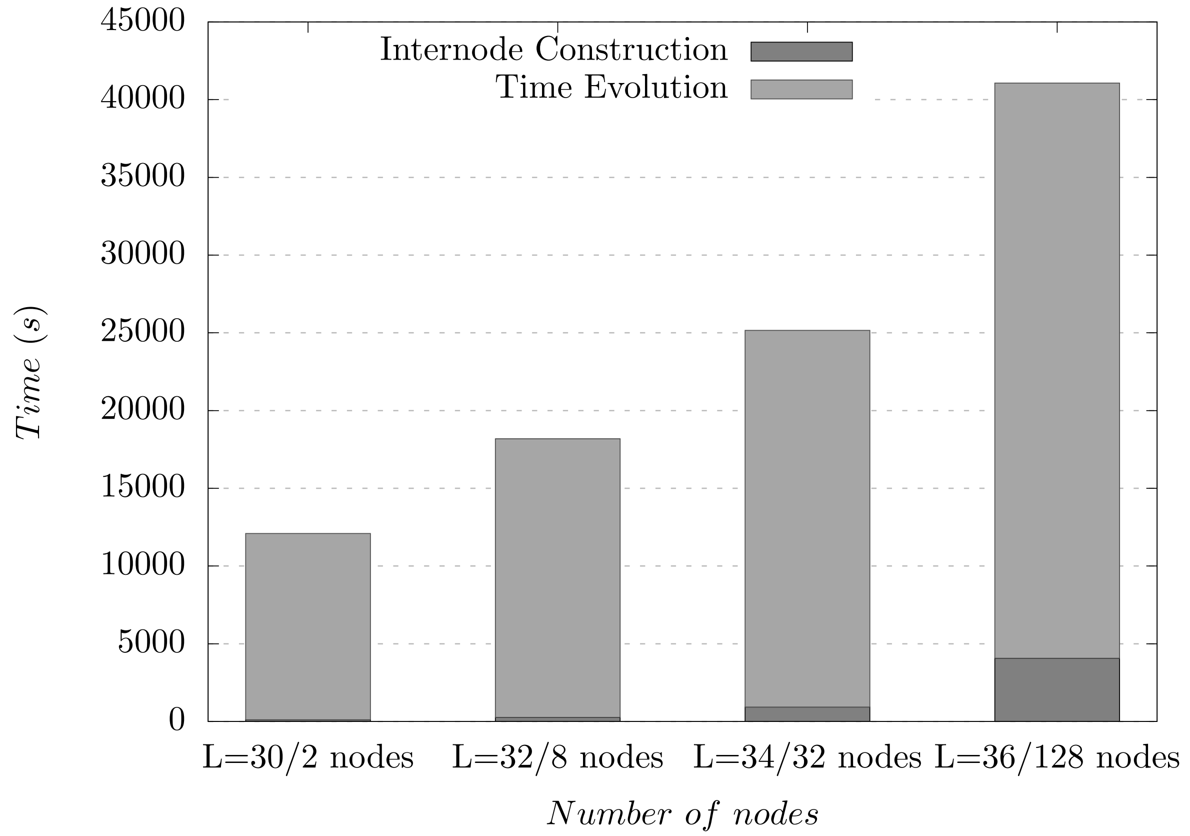


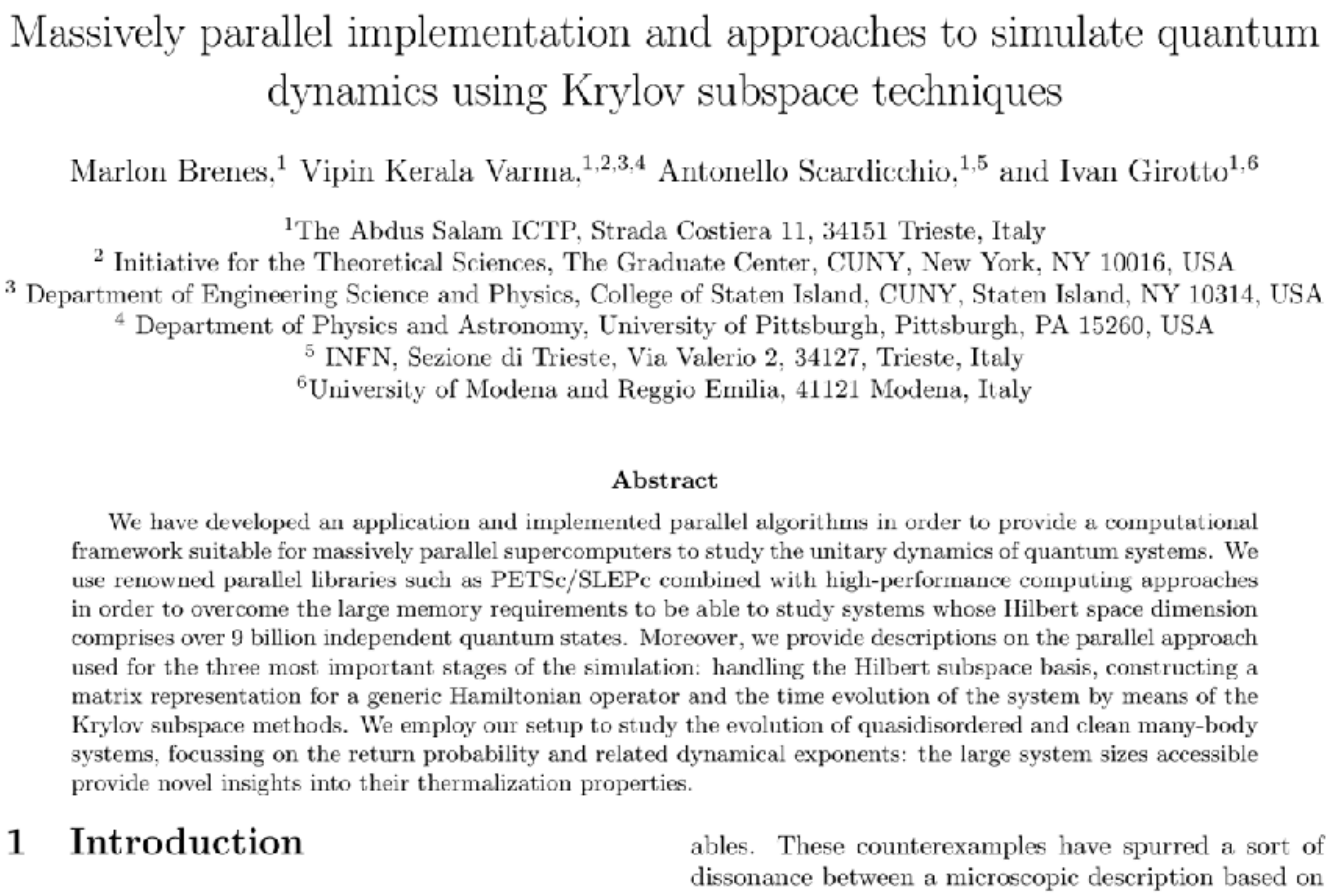
System sizes	$\mathcal{D}$	Matrix memory[6] (GB)	Full occupation (TB)
$L = 28$	4.01×10^7	18	0.053
$L = 30$	1.55×10^8	75	0.220
$L = 32$	6.01×10^8	308	0.902
$L = 34$	2.33×10^9	1269	3.72
$L = 36$	9.08×10^9	5 227	15.3
$L = 38$	3.53×10^{10}	21 490	63.0

PETSc/SLEPc performance on Marconi A1 for the Krylov subspace approach









Massively parallel implementation and approaches to simulate quantum dynamics using Krylov subspace techniques

Marlon Brenes,[1] Vipin Kerala Varma,[1,2,3,4] Antonello Scardicchio,[1,5] and Ivan Girotto[1,6]

[1]The Abdus Salam ICTP, Strada Costiera 11, 34151 Trieste, Italy
[2] Initiative for the Theoretical Sciences, The Graduate Center, CUNY, New York, NY 10016, USA
[3] Department of Engineering Science and Physics, College of Staten Island, CUNY, Staten Island, NY 10314, USA
[4] Department of Physics and Astronomy, University of Pittsburgh, Pittsburgh, PA 15260, USA
[5] INFN, Sezione di Trieste, Via Valerio 2, 34127, Trieste, Italy
[6]University of Modena and Reggio Emilia, 41121 Modena, Italy

Abstract

We have developed an application and implemented parallel algorithms in order to provide a computational framework suitable for massively parallel supercomputers to study the unitary dynamics of quantum systems. We use renowned parallel libraries such as PETSc/SLEPc combined with high-performance computing approaches in order to overcome the large memory requirements to be able to study systems whose Hilbert space dimension comprises over 9 billion independent quantum states. Moreover, we provide descriptions on the parallel approach used for the three most important stages of the simulation: handling the Hilbert subspace basis, constructing a matrix representation for a generic Hamiltonian operator and the time evolution of the system by means of the Krylov subspace methods. We employ our setup to study the evolution of quasidisordered and clean many-body systems, focussing on the return probability and related dynamical exponents: the large system sizes accessible provide novel insights into their thermalization properties.

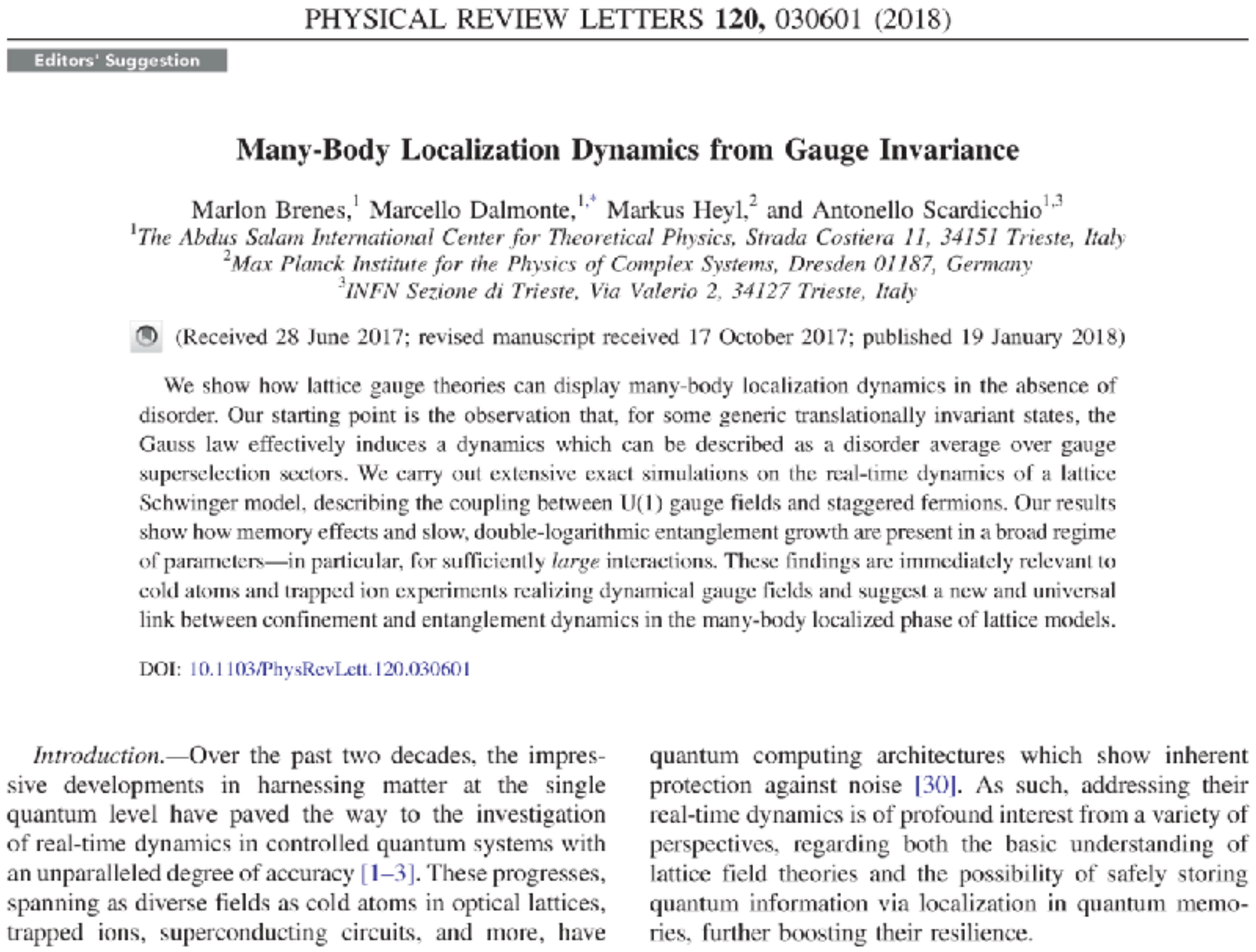
1 Introduction

ables. These counterexamples have spurred a sort of dissonance between a microscopic description based on

M Brenes, V.K. Varma, A. Scardicchio, I. Girotto. arXiv:1704.02770 (2017).

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

The Abdus Salam
International Centre
for Theoretical Physics


PHYSICAL REVIEW LETTERS **120,** 030601 (2018)

Editors' Suggestion

Many-Body Localization Dynamics from Gauge Invariance

Marlon Brenes,[1] Marcello Dalmonte,[1,*] Markus Heyl,[2] and Antonello Scardicchio[1,3]
[1]*The Abdus Salam International Center for Theoretical Physics, Strada Costiera 11, 34151 Trieste, Italy*
[2]*Max Planck Institute for the Physics of Complex Systems, Dresden 01187, Germany*
[3]*INFN Sezione di Trieste, Via Valerio 2, 34127 Trieste, Italy*

(Received 28 June 2017; revised manuscript received 17 October 2017; published 19 January 2018)

We show how lattice gauge theories can display many-body localization dynamics in the absence of disorder. Our starting point is the observation that, for some generic translationally invariant states, the Gauss law effectively induces a dynamics which can be described as a disorder average over gauge superselection sectors. We carry out extensive exact simulations on the real-time dynamics of a lattice Schwinger model, describing the coupling between U(1) gauge fields and staggered fermions. Our results show how memory effects and slow, double-logarithmic entanglement growth are present in a broad regime of parameters—in particular, for sufficiently *large* interactions. These findings are immediately relevant to cold atoms and trapped ion experiments realizing dynamical gauge fields and suggest a new and universal link between confinement and entanglement dynamics in the many-body localized phase of lattice models.

DOI: 10.1103/PhysRevLett.120.030601

Introduction.—Over the past two decades, the impressive developments in harnessing matter at the single quantum level have paved the way to the investigation of real-time dynamics in controlled quantum systems with an unparalleled degree of accuracy [1–3]. These progresses, spanning as diverse fields as cold atoms in optical lattices, trapped ions, superconducting circuits, and more, have quantum computing architectures which show inherent protection against noise [30]. As such, addressing their real-time dynamics is of profound interest from a variety of perspectives, regarding both the basic understanding of lattice field theories and the possibility of safely storing quantum information via localization in quantum memories, further boosting their resilience.

Final take-home message:
You could use your own implementation of linear algebra operations, but most likely you'll be better off using a good library if you care about performance.
HPC as means to open new avenues in scientific research.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

The Abdus Salam
ICTP
International Centre
for Theoretical Physics

Thank you!