Implementing Scalable Matrix-Vector Products for the Exact Diagonalization Methods in Quantum Many-Body Physics

Tom Westerhout and Bradford L. Chamberlain

Radboud Universiteit

SC23 Denver, CO | i am hpc.

Matrix-vector products

Matrix-vector products

$$y = A \cdot x$$

Matrix-vector products

$$y = A \cdot x$$

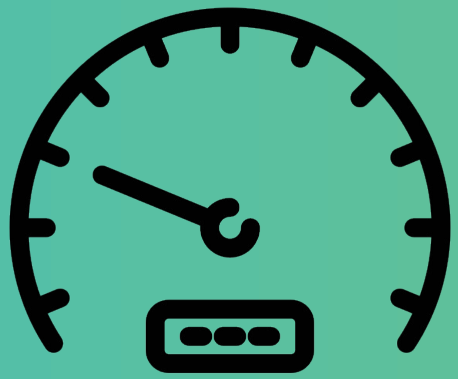
$$y_i = \sum_{j=0}^{N-1} A_{i,j} \cdot x_j$$

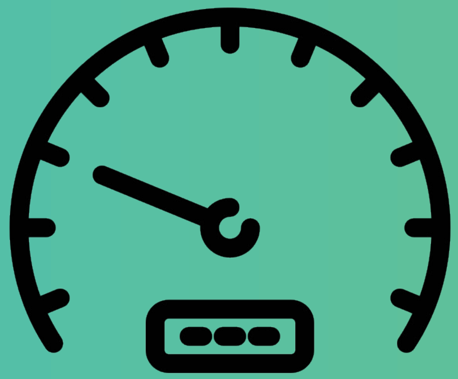
Matrix-vector products

$$y = A \cdot x$$

$$y_i = \sum_{j=0}^{N-1} A_{i,j} \cdot x_j$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ae + bf \\ ce + df \end{pmatrix}$$



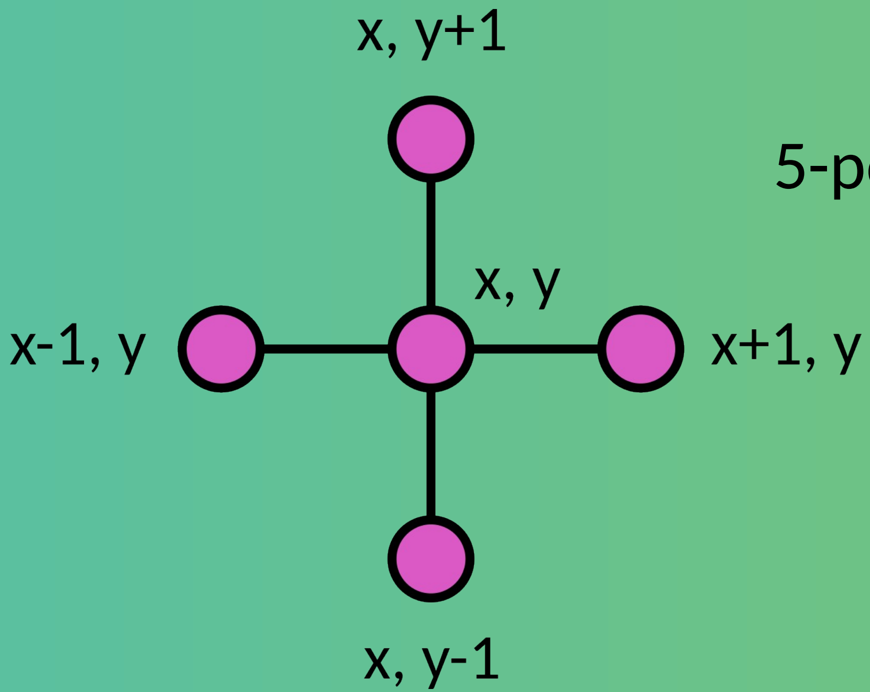



RAM

Lucky

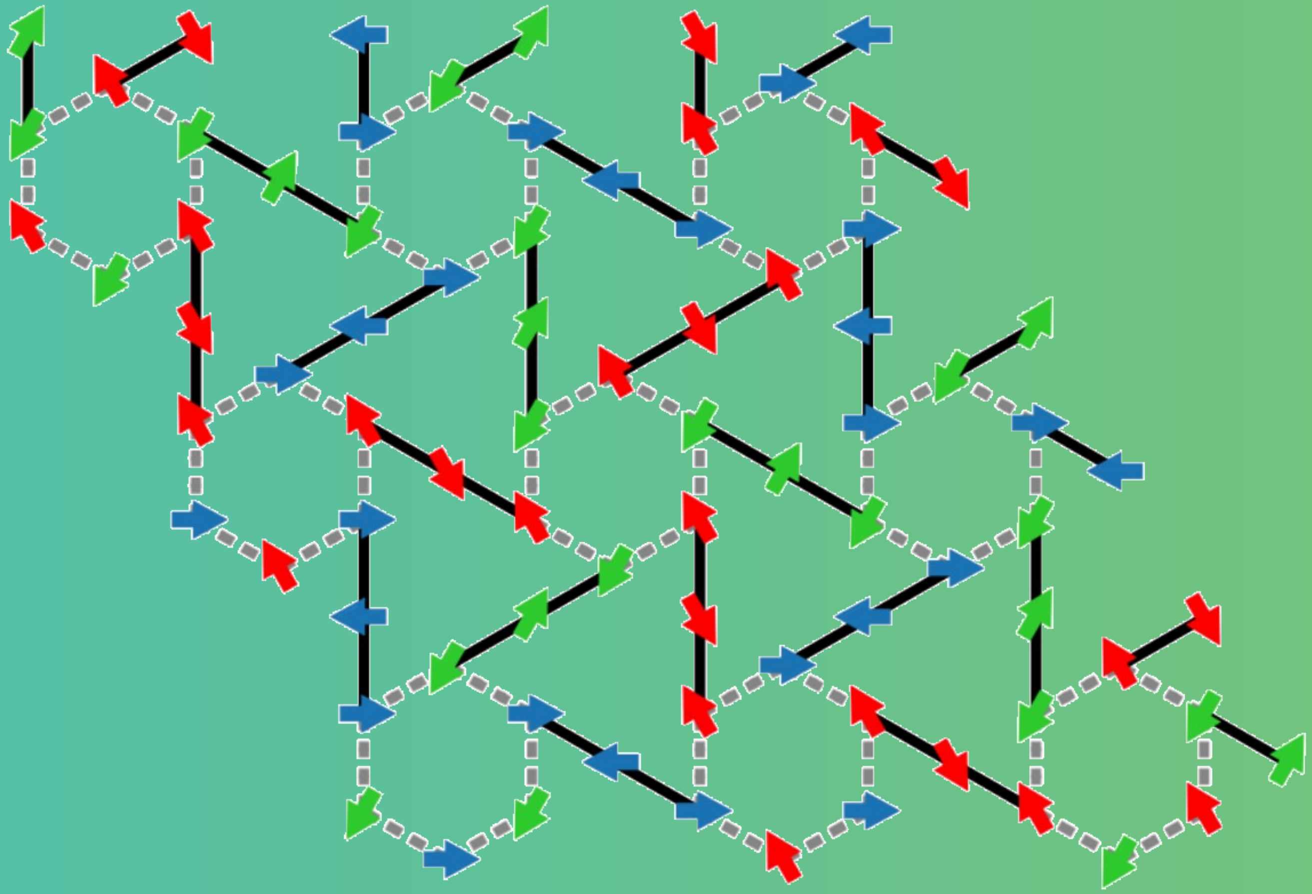
Lucky

Finite differences methods



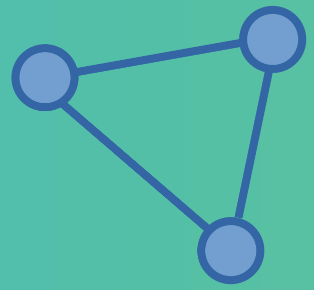
5-point stencil

Quantum mechanics



Exact diagonalization

N=3



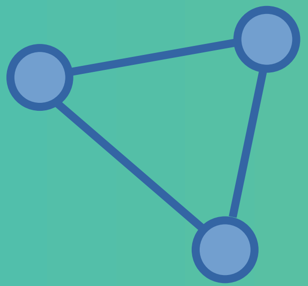
Physical system

```
[[ 3  0  0  0  0  0  0  0]
 [ 0 -1  2  0  2  0  0  0]
 [ 0  2 -1  0  2  0  0  0]
 [ 0  0  0 -1  0  2  2  0]
 [ 0  2  2  0 -1  0  0  0]
 [ 0  0  0  2  0 -1  2  0]
 [ 0  0  0  2  0  2 -1  0]
 [ 0  0  0  0  0  0  0  3]]
```

Hamiltonian matrix

Exact diagonalization

$N = 3$



Physical system

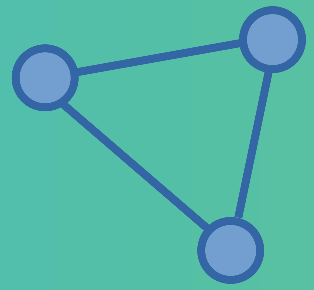
```
[[ 3  0  0  0  0  0  0  0]
 [ 0 -1  2  0  2  0  0  0]
 [ 0  2 -1  0  2  0  0  0]
 [ 0  0  0 -1  0  2  2  0]
 [ 0  2  2  0 -1  0  0  0]
 [ 0  0  0  2  0 -1  2  0]
 [ 0  0  0  2  0  2 -1  0]
 [ 0  0  0  0  0  0  0  3]]
```

Hamiltonian matrix

☹ Dimension: $\mathcal{O}\left(2^N\right)$

Exact diagonalization

$N=3$



Physical system

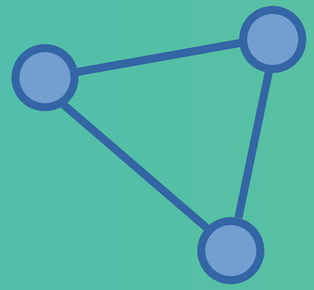
```
[[ 3  0  0  0  0  0  0  0]
 [ 0 -1  2  0  2  0  0  0]
 [ 0  2 -1  0  2  0  0  0]
 [ 0  0  0 -1  0  2  2  0]
 [ 0  2  2  0 -1  0  0  0]
 [ 0  0  0  2  0 -1  2  0]
 [ 0  0  0  2  0  2 -1  0]
 [ 0  0  0  0  0  0  0  3]]
```

Hamiltonian matrix

$$\hat{H}=\sum_{i=0}^{2}\sigma_i^x\sigma_{i+1}^x+\sigma_i^y\sigma_{i+1}^y+\sigma_i^z\sigma_{i+1}^z$$

Exact diagonalization

$N = 3$



Physical system

```
"σˣ₀σˣ₁ + σʸ₀σʸ₁ + σᶻ₀σᶻ₁ + \
 σˣ₁σˣ₂ + σʸ₁σʸ₂ + σᶻ₁σᶻ₂ + \
 σˣ₂σˣ₀ + σʸ₂σʸ₀ + σᶻ₂σᶻ₀"
```

Expression

$$\hat{H} = \sum_{i=0}^{2} \sigma_i^x \sigma_{i+1}^x + \sigma_i^y \sigma_{i+1}^y + \sigma_i^z \sigma_{i+1}^z$$

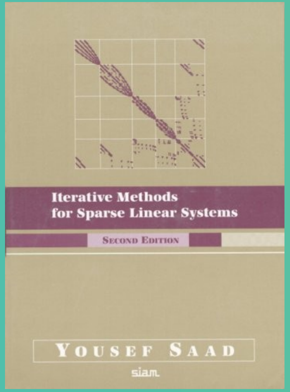
Procedure

Store the Hamiltonian
as an equation

```
"σˣ₀σˣ₁ + σʸ₀σʸ₁ + σᶻ₀σᶻ₁ + \
 σˣ₁σˣ₂ + σʸ₁σʸ₂ + σᶻ₁σᶻ₂ + \
 σˣ₂σˣ₀ + σʸ₂σʸ₀ + σᶻ₂σᶻ₀"
```

Solve

Describe the algorithm
with linear algebra primitives



Receive the
Nobel Prize

Memory for one vector: $\mathcal{O}\left(2^{N}\right)$ $\rightarrow$ distributed implementation required

Memory for one vector: $\mathcal{O}\left(2^N\right)$ $\rightarrow$ distributed implementation required

A Ph.D. or postdoc contract is short

lattice-symmetries (LS)

lattice-symmetries (LS)

Haskell

lattice-symmetries (LS)

Haskell

Halide

lattice-symmetries (LS)

CHAPEL

Haskell

Halide

lattice-symmetries (LS)

python™

APPTAINER

CHAPEL

Haskell

Halide

lattice-symmetries (LS)

python™

APPTAINER

CHAPEL

Haskell

Halide

Nix

Nix — a package manager and a programming language

Nix — a package manager and a programming language

```
toContainer = package: singularity-tools.buildImage {
  name = package.pname;
  contents = [ package ];
};
```

Nix — a package manager and a programming language

```
toContainer = package: singularity-tools.buildImage {
  name = package.pname;
  contents = [ package ];
};
```

(just 200 MB with Chapel & Haskell runtimes,
Infiniband support etc.)

Nix — a package manager and a programming language

```
toContainer = package: singularity-tools.buildImage {
  name = package.pname;
  contents = [ package ];
};
```

(just 200 MB with Chapel & Haskell runtimes,
Infiniband support etc.)

```
mkShell {
  buildInputs = [
    lattice-symmetries.kernels
    lattice-symmetries.haskell
    hdf5 hdf5.dev
  ];
  nativeBuildInputs = [
    chapel chapelFixupBinary
    gcc pkg-config
  ];
};
```

$$y_i = \sum_j H_{i,j} x_j$$

$$y_i = \sum_j H_{i,j} x_j$$

```
forall i in y.domain {
  var acc = 0.0;
  for (Hij, j) in getRow(H, i) {
    acc += Hij * x[j];
  }
  y[i] = acc;
}
```

parallelism over
nodes and cores

$$y_i = \sum_j H_{i,j} x_j$$

```
forall i in y.domain {
  var acc = 0.0;
  for (Hij, j) in getRow(H, i) {
    acc += Hij * x[j];
  }
  y[i] = acc;
}
```

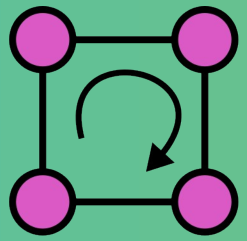
parallelism over
nodes and cores

$$y_i = \sum_j H_{i,j} x_j$$

```
forall i in y.domain {
  var acc = 0.0;
  for (Hij, j) in getRow(H, i) {
    acc += Hij * x[j];
  }
  y[i] = acc;
}
```

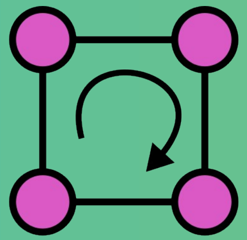
potentially remote
memory access

Symmetries



0	$\lvert\downarrow\downarrow\downarrow\downarrow\rangle$
1	$\lvert\downarrow\downarrow\downarrow\uparrow\rangle$
2	$\lvert\downarrow\downarrow\uparrow\downarrow\rangle$
3	$\lvert\downarrow\downarrow\uparrow\uparrow\rangle$
4	$\lvert\downarrow\uparrow\downarrow\downarrow\rangle$
5	$\lvert\downarrow\uparrow\downarrow\uparrow\rangle$
	$\vdots$
15	$\lvert\uparrow\uparrow\uparrow\uparrow\rangle$

Symmetries



0	$\lvert\downarrow\downarrow\downarrow\downarrow\rangle$
1	$\lvert\downarrow\downarrow\downarrow\uparrow\rangle$
2	$\lvert\downarrow\downarrow\uparrow\downarrow\rangle$
3	$\lvert\downarrow\downarrow\uparrow\uparrow\rangle$
4	$\lvert\downarrow\uparrow\downarrow\downarrow\rangle$
5	$\lvert\downarrow\uparrow\downarrow\uparrow\rangle$
	$\vdots$
15	$\lvert\uparrow\uparrow\uparrow\uparrow\rangle$

impose translational invariance

0	$\{\lvert\downarrow\downarrow\downarrow\downarrow\rangle\}$
1	$\{\lvert\downarrow\downarrow\downarrow\uparrow\rangle, \lvert\downarrow\downarrow\uparrow\downarrow\rangle, \lvert\downarrow\uparrow\downarrow\downarrow\rangle, \lvert\uparrow\downarrow\downarrow\downarrow\rangle\}$
2	$\{\lvert\downarrow\downarrow\uparrow\uparrow\rangle, \lvert\downarrow\uparrow\uparrow\downarrow\rangle, \lvert\uparrow\downarrow\downarrow\uparrow\rangle, \lvert\uparrow\uparrow\downarrow\downarrow\rangle\}$
3	$\{\lvert\downarrow\uparrow\downarrow\uparrow\rangle, \lvert\uparrow\downarrow\uparrow\downarrow\rangle\}$
4	$\{\lvert\downarrow\uparrow\uparrow\uparrow\rangle, \lvert\uparrow\downarrow\uparrow\uparrow\rangle, \lvert\uparrow\uparrow\downarrow\uparrow\rangle, \lvert\uparrow\uparrow\uparrow\downarrow\rangle\}$
5	$\{\lvert\uparrow\uparrow\uparrow\uparrow\rangle\}$

$$y_i = \sum_j H_{i,j} x_j$$

```
forall i in basisStates.domain {
  const beta = basisStates[i];
  var acc = 0.0;
  for (Hij, alpha) in getRow(H, beta) {
    const ref srcLocale = Locales[localeIdxOf(alpha)];
    on srcLocale {  // ← remote task spawn
      const j = stateToIndex(basisStates, alpha);
      acc += Hij * x[j];
    }
  }
  y[i] = acc;
}
```

$$y_i = \sum_j H_{i,j} x_j$$

```
y = 0;
forall j in basisStates.domain {
  const alpha = basisStates[j];
  for (Hij, beta) in getColumn(H, alpha) {
    const ref destLocale = Locales[localeIdxOf(beta)];
    const coeff = Hij * x[j];
    on destLocale {
      const i = stateToIndex(basisStates, beta);
      y[i] += coeff; // atomically
    }
  }
}
```

$$y_i = \sum_j H_{i,j} x_j$$

```
y = 0;
forall j in basisStates.domain {
  const alpha = basisStates[j];
  for (Hij, beta) in getColumn(H, alpha) {
    const ref destLocale = Locales[localeIdxOf(beta)];
    const coeff = Hij * x[j];
    on destLocale {
      const i = stateToIndex(basisStates, beta);
      y[i] += coeff; // atomically
    }
  }
}
```

Producer

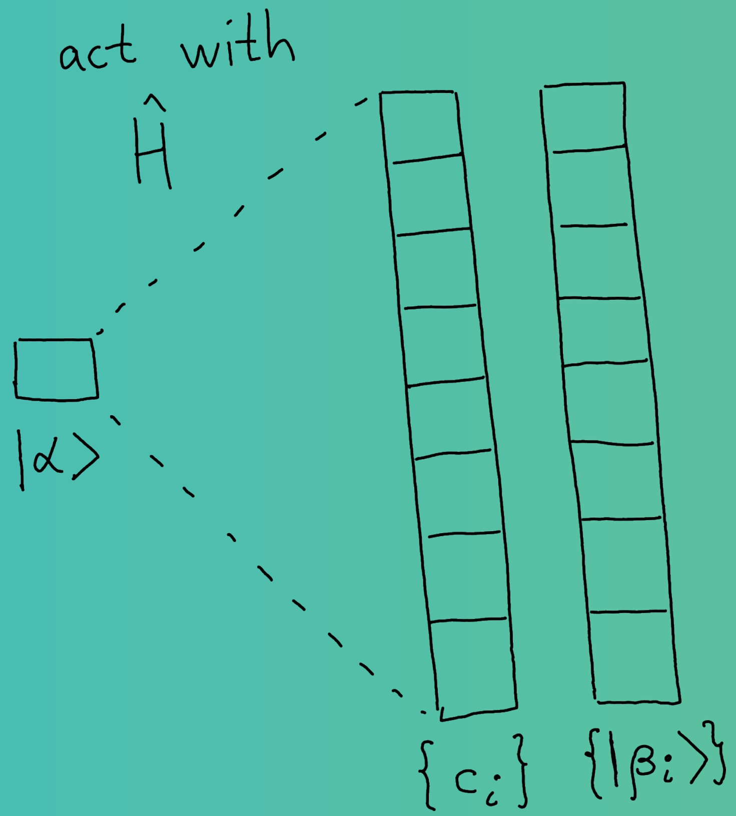
$$y_i = \sum_j H_{i,j} x_j$$

```
y = 0;
forall j in basisStates.domain {
  const alpha = basisStates[j];
  for (Hij, beta) in getColumn(H, alpha) {
    const ref destLocale = Locales[localeIdxOf(beta)];
    const coeff = Hij * x[j];
    on destLocale {
      const i = stateToIndex(basisStates, beta);
      y[i] += coeff; // atomically
    }
  }
}
```

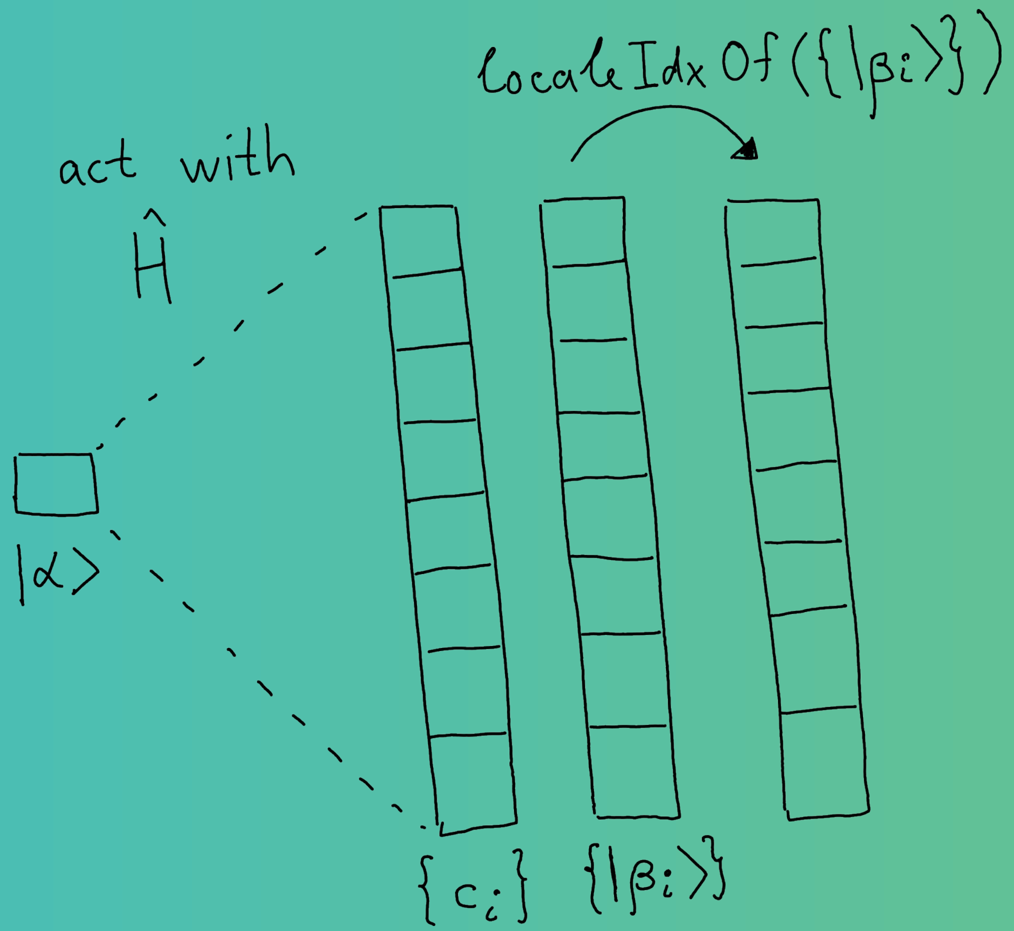
Producer

Consumer

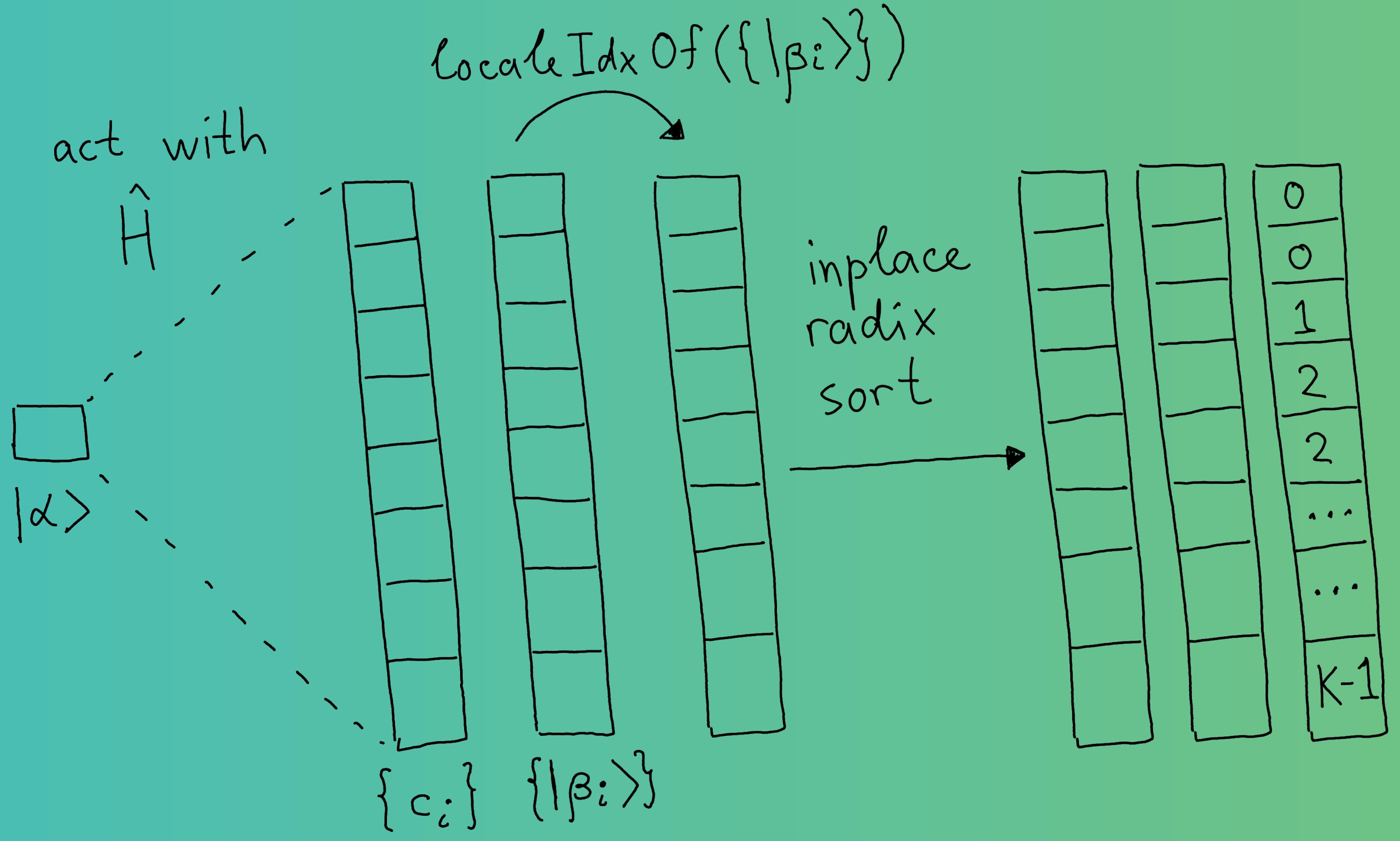
Producer



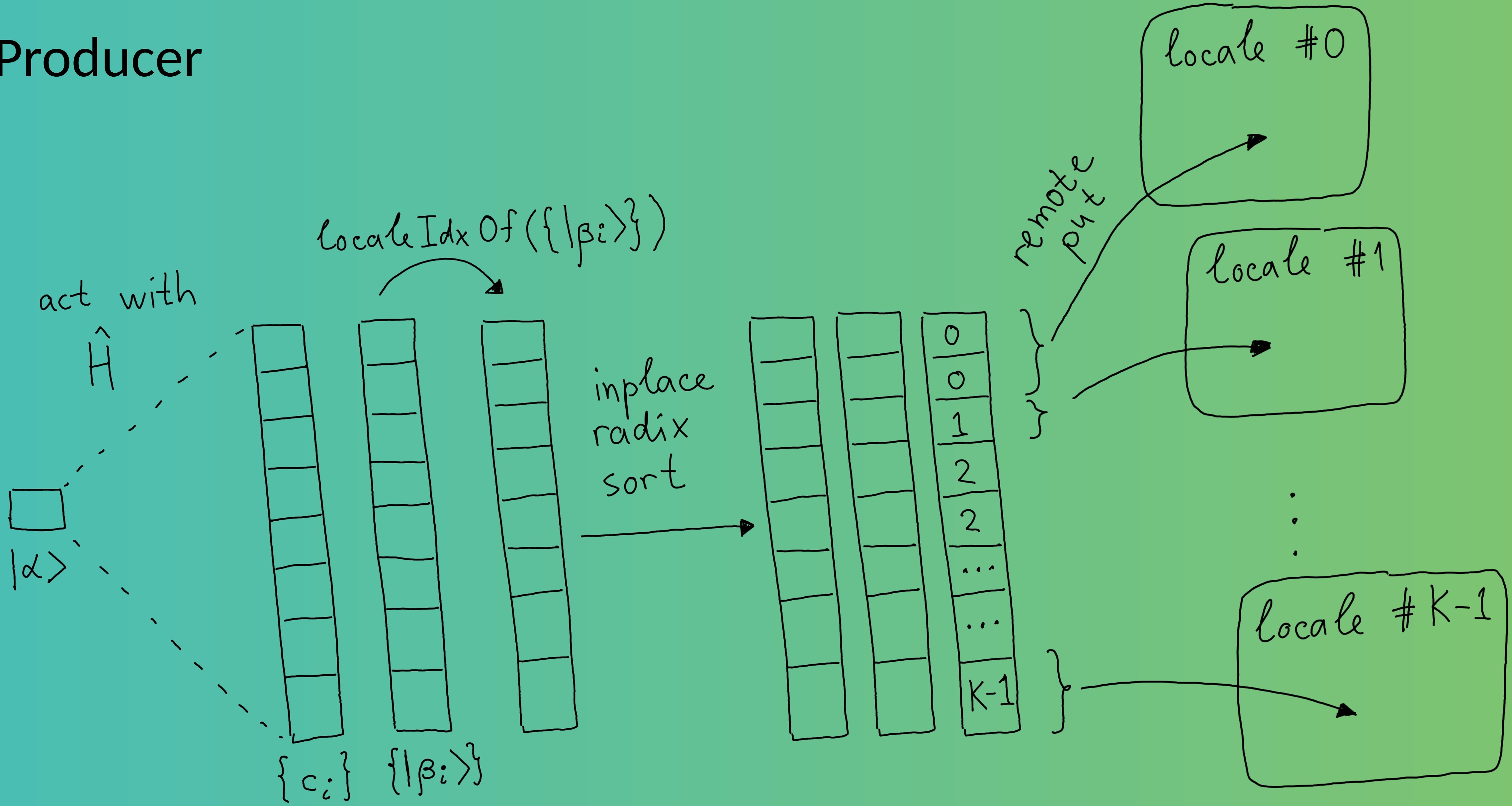
Producer



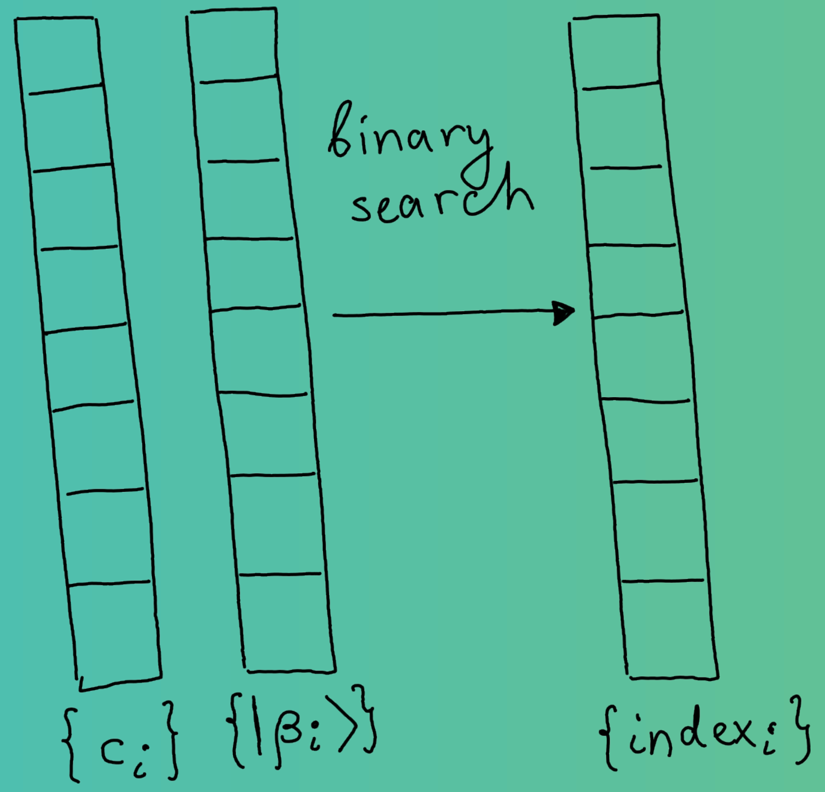
Producer



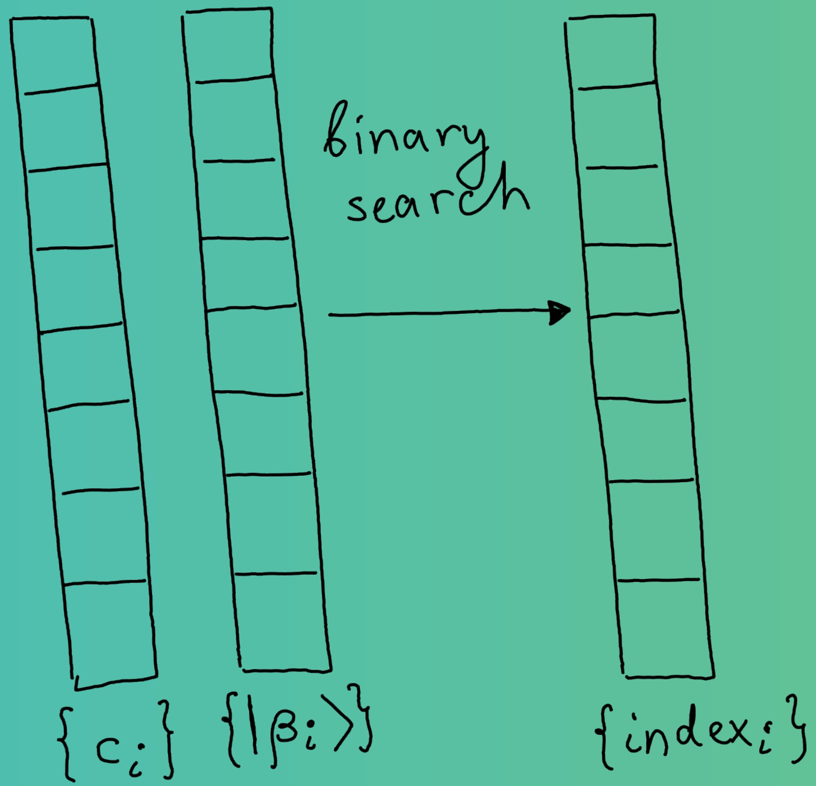
Producer


act with $\hat{H}$
$|\alpha\rangle$
$\{c_i\}$
$\{|\beta_i\rangle\}$
localeIdxOf({|βi⟩})
inplace radix sort
0
0
1
2
2
...
...
K-1
remote put
locale #0
locale #1
locale #K-1

Consumer



Consumer



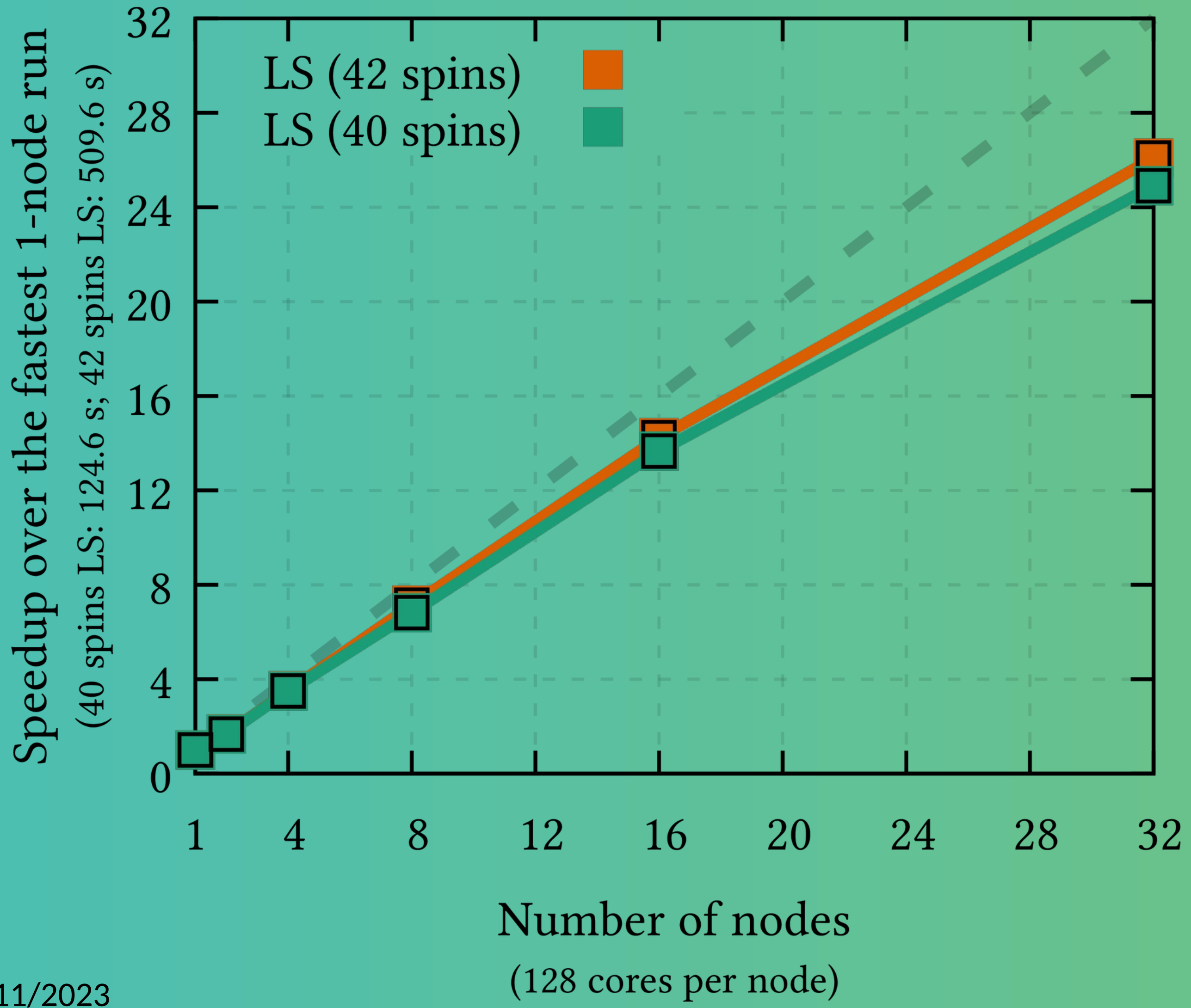

```
for i in 0 ..# count {
  // atomic +=
  y[index[i]].add(c[i], memoryOrder.relaxed);
}
```

Scaling

Snellius
2× AMD Rome 7H12 @2.6GHz
64 cores/socket
100 Gb/s Infiniband

	No symmetries	With symmetries
40 spins	1.0×10^{12}	8.6×10^{8}
42 spins	4.4×10^{12}	3.2×10^{9}

Scaling



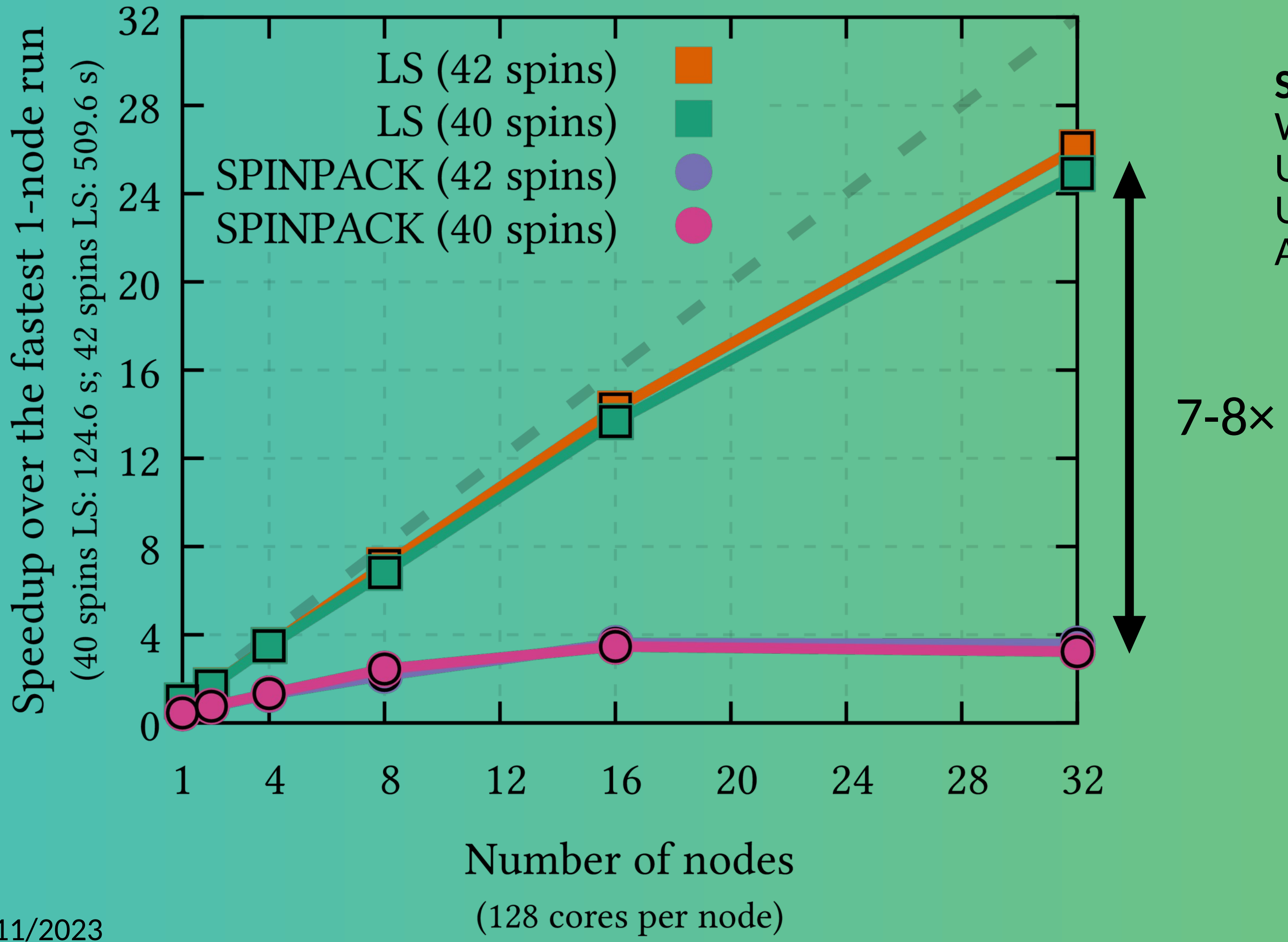
Snellius
2× AMD Rome 7H12 @2.6GHz
64 cores/socket
100 Gb/s Infiniband

	No symmetries	With symmetries
40 spins	1.0×10^{12}	8.6×10^{8}
42 spins	4.4×10^{12}	3.2×10^{9}

State-of-the-art MPI-based solution

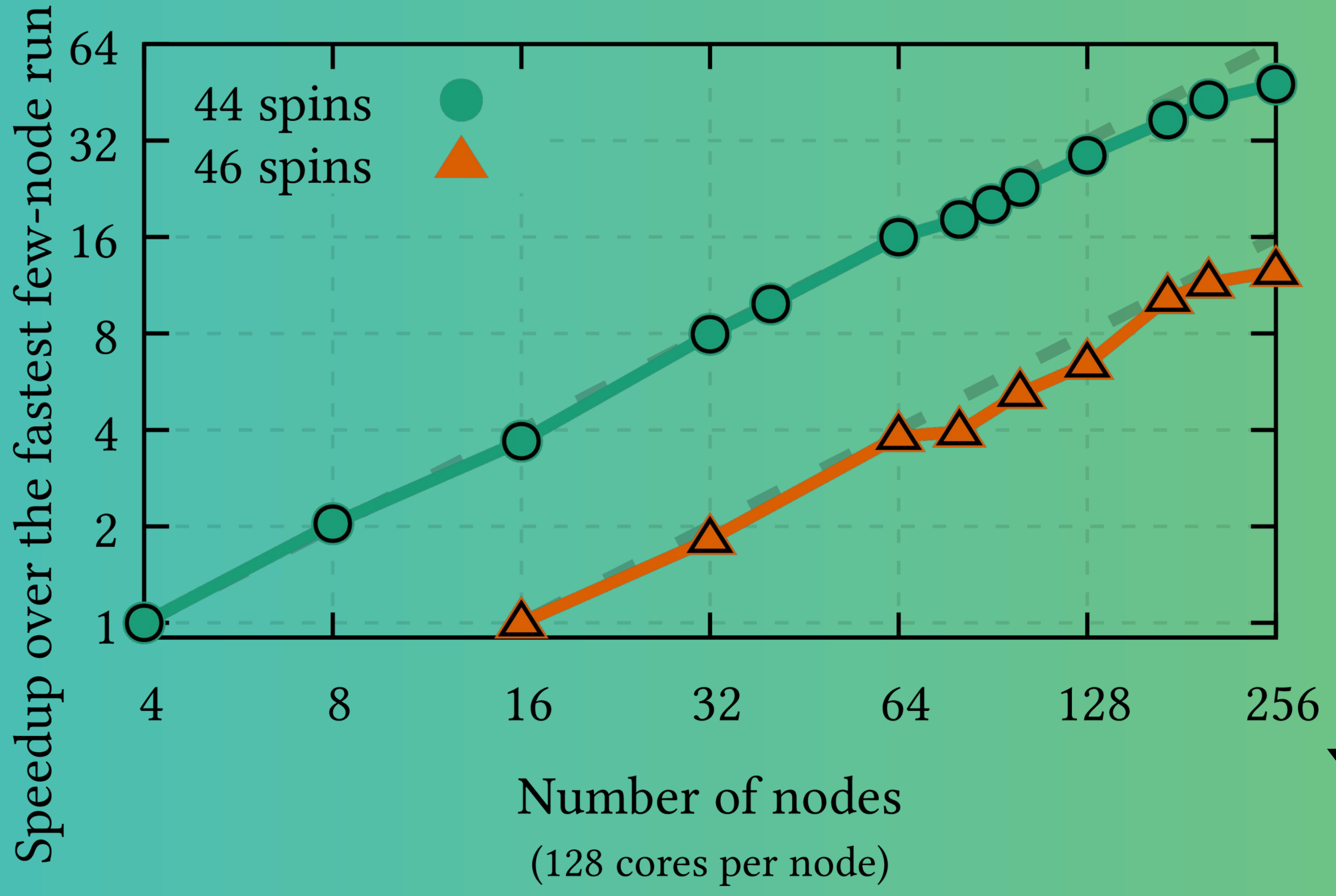
SPINPACK
Written in C
Uses MPI
Used in publications since 1995
About 26 000 software lines of code

State-of-the-art MPI-based solution



SPINPACK
Written in C
Uses MPI
Used in publications since 1995
About 26 000 software lines of code

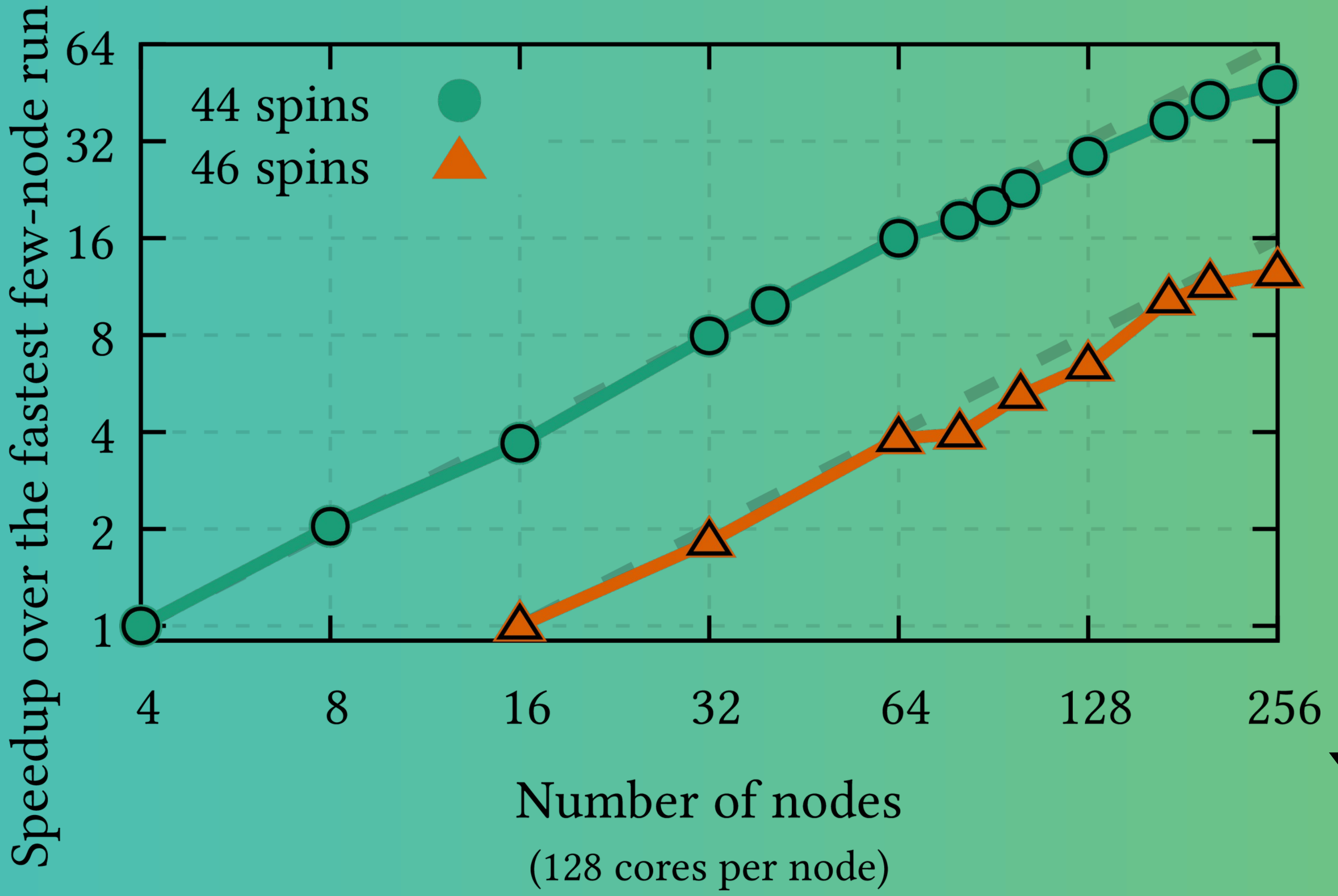
Further scaling



	No symmetries	With symmetries
44 spins	1.8×10^{13}	1.1×10^{10}
46 spins	7.0×10^{13}	4.5×10^{10}
48 spins	2.8×10^{14}	1.7×10^{11}

32 768 cores

Further scaling



32 768 cores

	No symmetries	With symmetries
44 spins	1.8×10^{13}	1.1×10^{10}
46 spins	7.0×10^{13}	4.5×10^{10}
48 spins	2.8×10^{14}	1.7×10^{11}

2.2 PB 😱

Conclusion

- Chapel-based implementation scales well to 256 nodes (or 32 768 cores)
- 7-8× improvement over the state of the art at 32 nodes
- 3× fewer SLOC than the state of the art
- Nix + Apptainer is an easy deployment solution
- Happy with Chapel — no desire to rewrite everything using MPI

Non-standard tools may still lead to competitive results