# Performance analysis of a stencil code in modern C++

Victor Eijkhout, Yojan Chitkara
Texas Advanced Computing Center
eijkhout@tacc.utexas.edu

IxPUG2024@SC24

# Scientific computing parallelism

- Large amounts of data:
  often cartesian multi-dimensional arrays, sometimes unstructured data
- Large amounts of parallelism:
  each element of output array independent.
- No explicit threading
  parallelism created by some runtime
- Range algorithm notion:
  do some operation on each element of a dataset

# Why modern C++

- Support for multi-D arrays
- ‘Range algorithms’: iteration without indices or bound checking
- Native support for parallelism. Ish.
- Libraries / language extensions for heterogeneous CPU/GPU/FPGA computing.

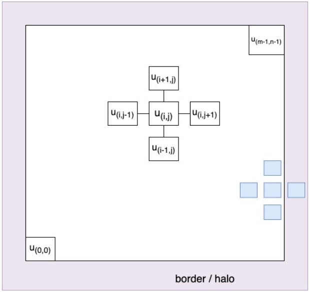
## Basic algorithm

# Power method

*Let $A$ a matrix of interest*
*Let $x$ be a random vector*
*For iterations until convergence*
    *compute the product* $y \leftarrow Ax$
    *compute the norm* $\gamma = \|y\|$
    *normalize* $x \leftarrow y/\gamma$

- Method for computing largest eigenvalue of a matrix
- Also Google Pagerank
- Stands for many scientific codes: Krylov methods, eigenvalues

# Stencil operations




| | | |
|---|---|---|
| · | $-1$ | · |
| $-1$ | 4 | $-1$ |
| · | $-1$ | · |

- This rectangular $m \times n$ thing is the vector
- The $4, -1, \ldots$ stencil is / stands for the matrix.
- Goes by: difference stencil, convolution, Toeplitz matrix

# Array parallelism

Traditional C implementation:

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ )
    out[ IINDEX(i,j,m,n,b) ] = in[ IINDEX(i,j,m,n,b) ] * factor;
```

```
// seq.cpp
#define IINDEX( i,j,m,n,b ) ((i)+b)*(n+2*b) + (j)+b
```

- Two / three-dimensional loop
- all dimensions large
- every output element independent

# Reductions

$\ell_2$ reduction:

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ ) {
    auto v = in[ IINDEX(i,j,m,n,b) ];
    sum_of_squares += v*v;
  }
return std::sqrt(sum_of_squares);
```

- Parallel except for the accumulation
- Obviously should not be done through atomic operation

# Stencil computation

Apply stencil to each $(i,j)$ index:

```
for ( idxint i=0; i<m; i++ ) {
  for ( idxint j=0; j<n; j++ ) {
    out[ IINDEX(i,j,m,n,b) ] = 4*in[ IINDEX(i,j,m,n,b) ]
      - in[ IINDEX(i-1,j,m,n,b) ] - in[ IINDEX(i+1,j,m,n,b) ]
      - in[ IINDEX(i,j-1,m,n,b) ] - in[ IINDEX(i,j+1,m,n,b) ];
  }
}
```

- Differential operator / image convolution
- Structure can be more complicated in scientific codes

# Implementations

# Implementation 1: OpenMP parallelism

Annotate loops as parallel and/or reduction:

```
    #pragma omp parallel for reduction(+:sum_of_squares)
    for ( idxint i=0; i<m; i++ )
      for ( idxint j=0; j<n; j++ ) {
        auto v = in[ IINDEX(i,j,m,n,b) ];
        sum_of_squares += v*v;
      }
    return std::sqrt(sum_of_squares);
  };
```

- Static assigment of iterations to threads by default
- Highly controlled affinity
- ‘`oned`’ as above, ‘`clps`’ for both loops collapsed
- Can be formulated as range algorithm.

# Tools: mdspan and cartesian_product

Data is declared as *mdspan*:

```
private:
  real *_data{nullptr};
  md::mdspan<
    real,
    md::dextents<idxint,2>
            > cartesian_data;
```

```
//! pointer to the data as 2D
    ↪array
auto& data2d() {
  return cartesian_data; };
const auto& data2d() const {
  return cartesian_data; };
```

```
// base.cpp
template< typename real >
bordered_array_base<real>::bordered_array_base
      ( idxint m,idxint n,int border )
  : _m(m),_n(n),_border(border)
  , _data( new real[ (m+2*border)*(n+2*border) ] )
  , data_owned(true)
  , cartesian_data
    ( md::mdspan
      ( _data,md::extents{m+2*border,n+2*border} )
      )
```

No performance loss [**?**].

# mdspan and cartesian_product

Index range is declared as *range::views::cartesian_product*:

```
const auto& s = data2d();
int b = this->border();
idxint
  lo_m = static_cast<idxint>(b),
  hi_m = static_cast<idxint>(s.extent(0)-b),
  lo_n = static_cast<idxint>(b),
  hi_n = static_cast<idxint>(s.extent(1)-b);
range2d = rng::views::cartesian_product
  ( rng::views::iota(lo_m,hi_m),rng::views::iota(lo_n,hi_n) );
```

- Vector allocated with size $(m+2b)\times(n+2b)$ to include border
- for handling of boundary conditions / halo regions in PDEs.

# Implementation 2: range over indices

Range-based for loop:

```
auto array = this->data2d();
#pragma omp parallel for reduction(+:sum_of_squares)
for ( auto ij : this->inner() ) {
  auto [i,j] = ij;
  auto v = array[i,j];
  sum_of_squares += v*v;
}
return std::sqrt(sum_of_squares);
```

- Range over indices, not over data
  (Indices are a subset of the full data!)
- OpenMP can handle iterators.

# Stencil operation

Most complicated operation of the bunch:

```
// span.cpp
auto out = this->data2d();
const auto in = other.data2d();
#pragma omp parallel for
for ( auto ij : this->inner() ) {
  auto [i,j] = ij;
  out[ i,j ] = 4*in[ i,j ]
    - in[ i-1,j ] - in[ i+1,j ] - in[ i,j-1 ] - in[ i,j+1 ];
}
```

- Hard to formulate as range algorithm
- Performance not necessarily determined by floating point operations.

# Implementation 3: Kokkos

Open Source heterogeneous execution layer

```
Kokkos::parallel_for
  ("Update x",
   Kokkos::MDRangePolicy<Kokkos::Rank<2>>
       ({1, 1}, {msize-1, nsize-1}),
   KOKKOS_LAMBDA(int i, int j) {
    x(i, j) = Ax(i, j) / norm;
  });
```

- Same code for CPU and GPU
- Implicit task queue
- Two-dimensional indexing
- Range algorithm-like philosophy

# Implementation 4: Sycl

Open standard, but mostly pushed by Intel

```
q.submit([&] (handler &h) {
  accessor D_a(Buf_a,h,write_only);
  h.parallel_for
    (range<2>(msize-2,nsize-2),
     [=](auto index){
       auto row = index.get_id(0) + 1;
       auto col = index.get_id(1) + 1;
       D_a[row][col] = 1.;
     });
}).wait();
```

- Heterogeneous CPU/GPU code, transparent data movement
- Range algorithm-like syntax, but explicit task queue

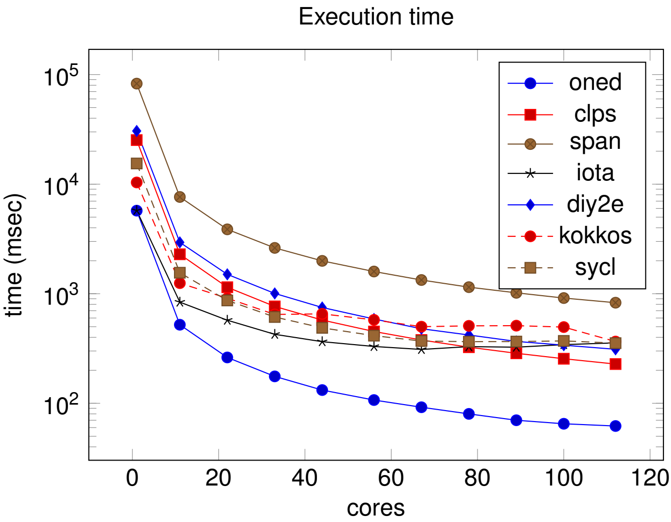
# Implementation X: ranges

```
#include <range>
std::ranges::for_each
  ( std::execution::par_unseq,
    some_container,
    [] ( /* ... */ ) { /* ... */ }
  );
```

- ‘Apply this point function to each element of this range’
- Elegant, trivially parallelized
- Eh . . . 1. relies too much on TBB and such: no affinity control
- Eh . . . 2. hard to account for the halo region
- Eh . . . 3. hard to express stencil operations (although see `https://www.youtube.com/watch?v=ImM7f5IQOaw`)
- Eh . . . 4. compiler bug prevented parallel execution of the cartesian product iterator.

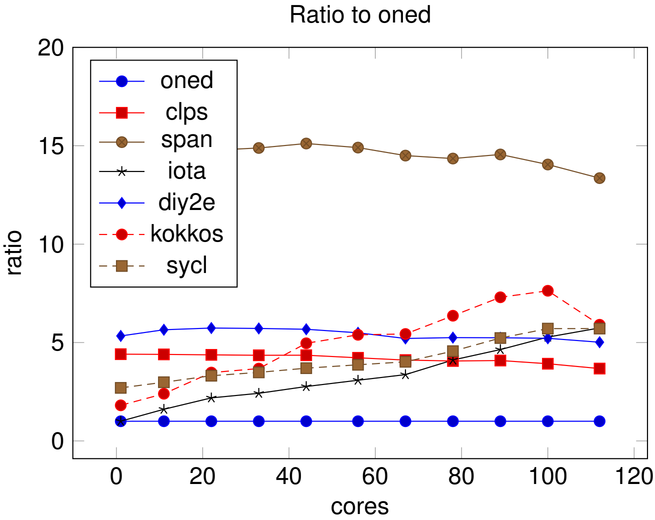
# Tests

# Comparing models (Intel)

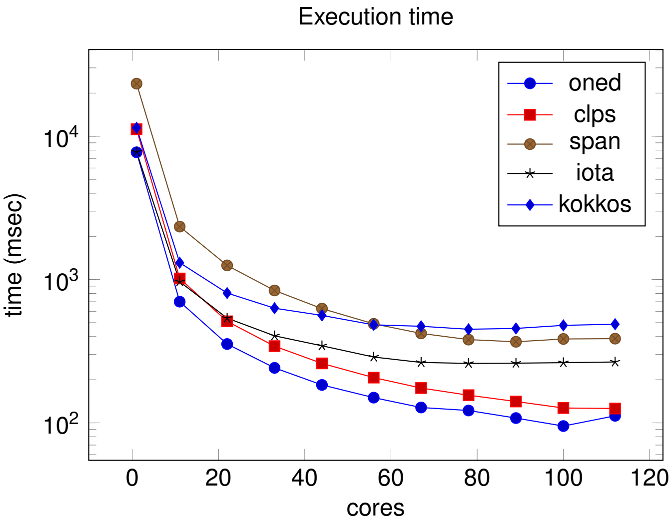



Intel compiler. C-style variant fastest.

# Ratio to fastest (Intel)

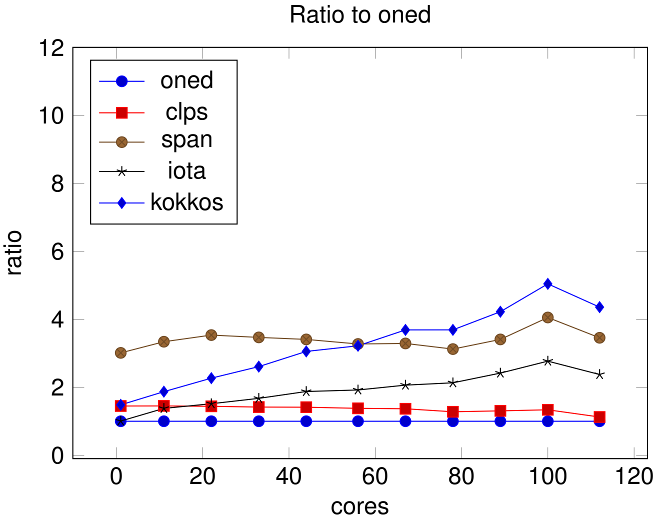


# Comparing models (Gcc)




Gcc compiler. less variance between variants

# Ratio to fastest (Gcc)



# Where do we lose performance?

Hint: `perf` output on the ‘span’ variant:

```
      55.60%  [.] std::ranges::cartesian_product_view<std::ranges::iota_view<long, long>, std::ranges::iota_view<long, long>>::_Iterator<true>::operator+=
      18.73%  [.] __divti3
      11.33%  [.] linalg::bordered_array_span<float>::central_difference_from
       5.37%  [.] linalg::bordered_array_span<float>::scale_interior
       5.01%  [.] linalg::bordered_array_span<float>::l2norm
       2.69%  [.] __divti3@plt
```

Index calculations take lots of time.

# GCC vs Intel

Observation:
GCC performance seems better than Intel on the 2D indexing schemes

I'm working on that.

# Conclusion and Acknowledgement

- ‘Fancy’ schemes suffer from indexing overhead
  strongly implementation and compiler dependent.
- This work was supported by the Intel oneAPI Center of Excellence, and the TACC STAR Scholars program, funded by generous gifts from TACC industry partners, including Intel, Shell, Exxon, and Chevron.

TEXAS ADVANCED COMPUTING CENTER

PARTNERSHIP PROGRAM

github

# Bibliography


David S. Hollman, Bryce Adelstein Lelbach, H. Carter Edwards, Mark Hoemmen, Daniel Sunderland, and Christian R. Trott.
mdspan in c++: A case study in the integration of performance portable features into international language standards.
In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, volume 74, pages 60–70. IEEE, November 2019.