Performance analysis of a stencil power method

Victor Eijkhout[*], Yojan Chitkara, Daksh Chaplot

August 13, 2024

Part I
Writeup

1 Introduction

In this paper we evaluate multiple parallel programming models with respect to both ease of expression, and resulting performance. We do this by implementing the mathematical algorithm known as the 'power method' in a variety of ways, for the moment all in C++.

2 Algorithm aspects

We briefly discuss the power method, mostly going into its computational aspects. For the mathematical side, see HPC book, section **??**.

Computationally, the power method is an attractive paradigmatic example in that it exhibits the most common parallelism patterns:

- independent or 'convenient' parallelism;
- global reduction operations
- point-to-point communications.

As follows. The method is given by

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 $x \leftarrow y/\gamma$

[*] eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin.

where in the limit γ will be the largest eigenvalue of A, and x the corresponding eigenvector.

The basis to parallelizing the power method lies in the distribution of x. Each processing element p, whether that be a core or processor, takes care (in a sense that we will later defined) of a disjoint set of indices I_p. The operations then are:

- The normalization step has independent parallelism: each component $y_i = x_i/\gamma$ can be independently computed, so each process p can compute y_i for $i \in I_p$ fully independently;
- Computing γ is a all-reduction: each processing element p first computes $\gamma_p = \sqrt{\sum_{i \in I_p} y_i^2}$ independently, and $\gamma = \sqrt{\sum_p \gamma_p}$ is then formed by combination, and made available to all p;
- Finally, for the operator A we assume a sparse matrix, for instance a Finite Difference (FD) stencil for a Partial Diffential Equation (PDE), or a *convolution* kernel. This is characterized by each component y_i of the output needing several components x_j of the input. In our case, putting a two-dimensional indexing on the vectors:

$$y_{ij} = 4x_{ij} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$$

2.1 Loop parallelism

In our power method application a first example of a parallel loop is scaling an array by a factor.

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ )
    out[ IINDEX(i,j,b,n2b) ] = in[ IINDEX(i,j,b,n2b) ] * factor;
```

where `m`, `n` are the size of the domain and `b` is the width of a halo region.

Some remarks:

1. We use template parameters `idxint` and `real` for integers and floating point quantities, respectively.
2. The loops range only over the $m \times n$ interior part of the domain, which is allocated with size $(m+b) \times (n+b)$.
3. We are using a traditional macro to translate from two-dimensional to one-dimensional indexing, skipping the border points:

   ```
   // seq.cpp
   #define IINDEX( i,j,b,n2b ) ((i)+b)*n2b + (j)+b
   ```

   Later we will discuss other indexing schemes.
4. For the initial implementation, the `out`, `in` data are `double*` pointers, for instance obtained as the `data` member of a `std::vector`. Later we will explore implementing the data as an `mdspan` object so we can use the two-dimensional `[i,j]` type indexing.

2.2 Reduction

The ℓ_2 reduction of our example application looks in code like:

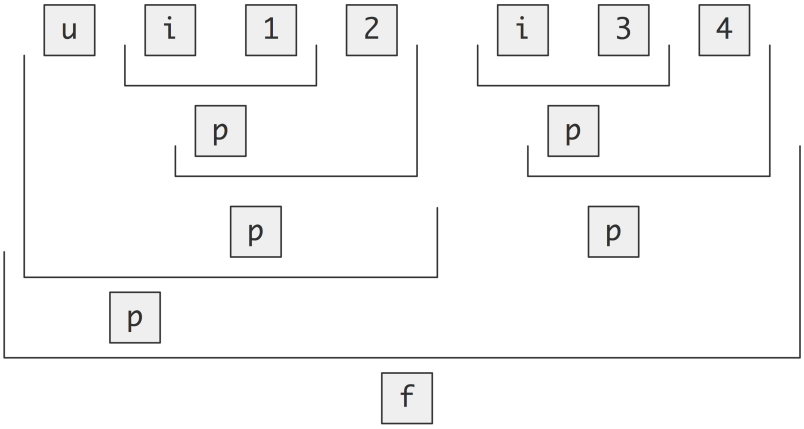

Figure 1: Structure of a reduction of four items on two threads, where u is the user-supplied initial value, and i is the natural initial value for the reduction operator.

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ ) {
    auto v = out[ IINDEX(i,j,b,n2b) ];
    sum_of_squares += v*v;
  }
return std::sqrt(sum_of_squares);
```

OpenMP resolves the race condition on the reduction variable by giving each thread a local copy of `sum_of_squares`, summing into that, and adding the local copies together in the end.

2.3 Five-point stencil

The evaluation of the 5-point difference operator is perfectly parallel in that there are no dependencies between the points of the output vector. However, the loop is more complicated than the scaling operation above:

```
for ( idxint i=0; i<m; i++ ) {
  for ( idxint j=0; j<n; j++ ) {
    out[ IINDEX(i,j,b,n2b) ] = 4*in[ IINDEX(i,j,b,n2b) ]
      - in[ IINDEX(i-1,j,b,n2b) ] - in[ IINDEX(i+1,j,b,n2b) ]
      - in[ IINDEX(i,j-1,b,n2b) ] - in[ IINDEX(i,j+1,b,n2b) ];
  }
}
```

While this operation is somewhat like a 'transform' range operation, the complexity of the right-hand-side indexing precludes using an actual range algorithm. Therefore, in later versions of this code we will range over the indices, rather than the actual data.

3 Parallelization strategies

There are various ways we can parallelize our example application.

3.1 OpenMP loop parallelism

Above in section 2.1 we already showed the simplest parallelization scheme: we loop two-dimensionally over the index space, and translate that through a C Preprocessor (CPP) macro to one-dimensional indexing in a traditional container. The loops are then parallelized by OpenMP.

In graphs to follow we indicate by 'oned' the mode where we only parallelize the outer loop. For instance, the norm calculation is:

```
template< typename real >
real bordered_array_1d<real>::l2norm() {
  real sum_of_squares{0};
  auto out = this->data();
#   pragma omp parallel for reduction(+:sum_of_squares)
  for ( size_t i=0; i<_m; i++ )
    for ( size_t j=0; j<_n; j++ ) {
      auto v = out[ IINDEX(i,j) ];
      sum_of_squares += v*v;
    }
  log_flops(_m*_n*3); log_bytes( sizeof(real)*_m*_n*1 );
  return std::sqrt(sum_of_squares);
};
```

We designate by 'clps' this same code, but adding `collapse(2)` to each OpenMP loop nest.

3.2 `mdspan` indexing

While the scaling and the norm calculation can be expressed as range algorithms, doing so for the five-point stencil update is trickier, so we take a two-step approach:

1. We access data through an `mdspan`, so that have multidimensional indexing:

   Class data:

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

   Accessor:

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

2. We define the iteration space as a `cartesian_product` range view:

   ```
   mutable std::optional< decltype( rng::views::cartesian_product
                         ( rng::views::iota(idxint{0},idxint{0}),
                           rng::views::iota(idxint{0},idxint{0}) ) ) >
         range2d = {};
   auto inner() const {
     if (not range2d.has_value()) {
       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) );
    }
    return *range2d;
  };
```

This allows us to write, cleanly and compactly:

```
// span.cpp
template< typename real >
void bordered_array_span<real>::central_difference_from
    ( const linalg::bordered_array_base<real>& _other,bool trace ) const {
  const auto& other =
    dynamic_cast<const linalg::bordered_array_span<real>&>(_other);
  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 ];
  }
};
```

Unlike the OpenMP double loop, we now have a single loop spanning the two-dimensional domain. Also, note that this domain is the interior of a larger domain.

In graphs to follow we indicate this mode by ‘span’.

3.3 DIY cartesian product

The *cartesian_product* view is quite general so one could wonder about overhead. We write a custom iterator over a contiguous domain. It maintains internally i, j coordinates which are updated as follows:

Simple

```
auto& operator++( ) {
  j++; i+= (j/m); j = j%m;
  return *this; };
```

Optimized

```
auto& operator++( ) {
  c--;
  j++; j *= (c>0);
  i += (c==0);
  c += m*(c==0);
  return *this;
};
```

3.4 Kokkos and Sycl

We also use the Kokkos and Sycl libraries in their ‘host’ mode. In graphs to follow we indicate these modes by ‘kokkos’ and ‘sycl’ respectively.

3.4.1 Sycl

Sycl is an open standard that targets heterogeneous parallelism through strict standard C++. It has mechanisms for memory management between host CPU and devices (both GPU and FPGA), and for expressing common parallel algorithms.

The preferred mechanism for handling memory coherence is through buffers. Host memory is wrapped in a buffer structure:

```
std::vector<real> Mat_A(msize*nsize);
buffer<real,2> Buf_a(Mat_A.data(),range<2>(msize,nsize));
```

This memory can then transparently be accessed in a kernel, whether run on host or device, where coherence is ensured by the runtime. This turns out to be as efficient as more explicit mechanisms.

```
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();
```

Note that Sycl has a true two-dimensional indexing mode.

We see that the `parallel_for` construct resembles a C++ range algorithm: it combines a range – though of the index space, not the data space – and a lambda expression to be applied at each point.

Sycl is alone among the models studied here in that it requires index shifting in order to range over the interior of the index space.

3.4.2 Kokkos

Kokkos is the execution layer of the Trilinos project. It is, like Sycl, a data-parallel programming mode that supports host and device execution with the same code. Unlike Sycl there is no explicit queue; instead, objects are explicitly associated with the host or device data space.

```
// diff2d.cpp
using MemSpace = Kokkos::HostSpace;
using Layout = Kokkos::LayoutRight;
using HostMatrixType = Kokkos::View<real**, Layout, MemSpace>;
HostMatrixType x("x", msize,nsize );
```

Like Sycl, there are explicit parallelism constructs. These closely resemble C++ range algorithms, specifying an explicit index space over which to iterate, and the lambda expression to apply at each index.

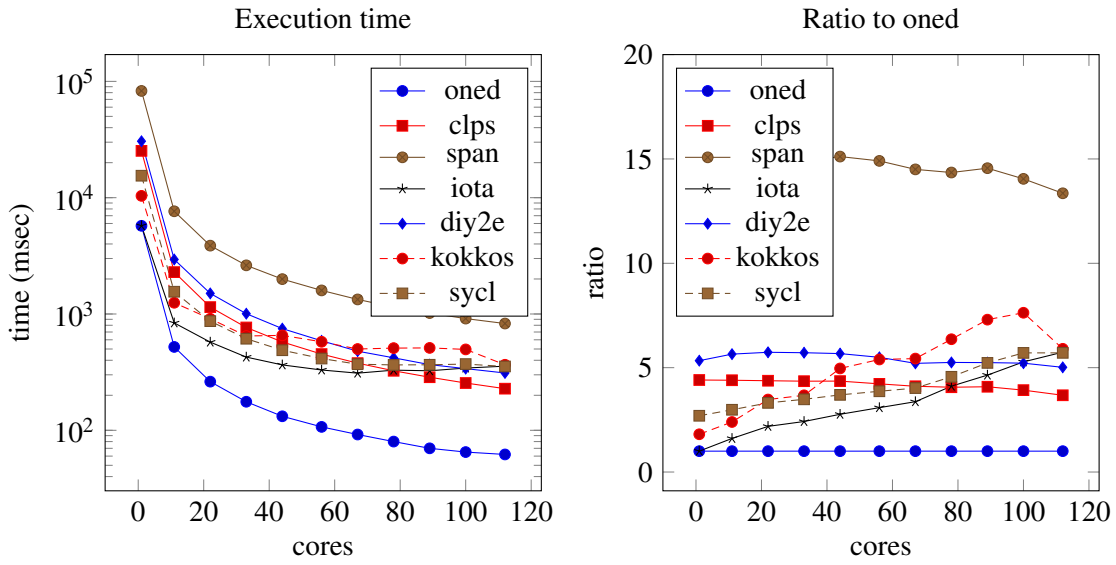

Figure 2: Comparing implementation strategies, Intel 2024 compiler on a 112-core Sapphire Rapids node.

```
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;
  });
```

4 Timing comparison and discussion

We perform different comparisons:

- different parallelization models on a given compiler and CPU;
- different Intel processor generations, given a model and compiler;
- different compilers, given a model and CPU.

4.1 Comparing parallelization models

Let's compare various implementation strategies. Test given here are on an *Intel Sapphire Rapids* dual socket node with 112 cores total. We compare the Intel 2024 (figure 2) and GCC 13 (figure 3) compilers.

We make the following observations.