HPC 4MA 2021/2022

Members
PHAM Tuan Kiet
VO Van Nghia

Date
12 Dec, 2021

Contents

1 Introduction

In this reports, we study two techniques that are widely-known: `OpenMP` and `MPI`. For a less biased value, each result has been run 100 times and taking average. All the source code could be found on github.

2 OpenMP

2.1 Optimization techniques

2.1.1 Naive dot

We first mention here the original `naive_dot` function. This function serves as an anchor (or base case) for performance comparision as well as for making sure we have the right result when using other techniques.

```
  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb * j];
```

Below is the output of `naive_dot` for `M = 1`, `K = 2` and `N = 2`:

```
## ( 1.00  1.50 )
##
## ( 1.00  1.50 )
## ( 1.50  2.00 )
##
## Frobenius Norm   = 5.550901
## Total time naive = 0.000001
## Gflops           = 0.013333
##
## ( 3.25  4.50 )
```

As

$$\begin{pmatrix} 1 & 1,5 \end{pmatrix} \begin{pmatrix} 1 & 1,5 \\ 1,5 & 2 \end{pmatrix} = \begin{pmatrix} 3,25 & 4,5 \end{pmatrix}$$

The result of this function is correct. We move on to the next technique.

2.1.2 Spatial locality

Spatial locality refers to the following scenario: if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In order to take advantages of this property, we notice that:

- In memory, `A`, `B`, `C` are stored in contiguous memory block.
- When using the index order `i`, `j`, `k`, we access `B` consecutively (as we access `B` by `B[k + ldb * j]`), but not `A` and `C`.
- Data from `A`, `B`, `C` are loaded in a memory block consisting of severals consecutive elements to cache. Thus, we could make use of spatial locality when reading data continously.

From 3 points above, we decide to switch the index order to `k`, `j`, `i`. Now we see that both reading and writing operations on `C` are in cache, this brings us a critical gain in performance. In addition, reading operations on `A` are in cache too but those on `B` are not.

```
  for (k = 0; k < K; k++)
    for (j = 0; j < N; j++)
      for (i = 0; i < M; i++) C[i + ldc * j] += A[i + lda * k] * B[k + ldb * j];
```

For comparision, we have a table below with `M = 4`, `K = 8` and `N = 4`.

Table 1: naive vs saxpy when M, N, K is small

algo	time	norm	gflops
naive	0.0000021	3.461352	0.1262202
saxpy	0.0000017	3.461352	0.1724373

We have the frobenius norm of both techniques are 3,461352 which indicate we have the right computation result. In addition, calculating time is already significantly small (≈ 0 second in both methods) and the difference between these two can therefore be ommited.

However, if we set `M = 2048`, `K = 2048` and `N = 2048`, there will be a huge performance gain as in the table shown below. In addition, from now, for an easier comparision between results, we will consider the default value of `M`, `K` and `N` is `M = 2048`, `K = 2048` and `N = 2048` if not explicitly mentioned.

Table 2: naive vs saxpy when M, N, K is big

algo	time	norm	gflops
naive	120.06615	2.323362	0.1430886
saxpy	57.37033	2.323362	0.2995959

Here, the `naive_dot` function is approximately 2,09 times slower than the `saxpy_dot` function.

2.1.3 OpenMP parallelization

In this section, we will analyse the main technique of this chapter: `OpenMP`. First, we show how we enable it on each function. We add a directive above each loop we want to parallelize whose general form is as below:

- Variables inside `private` are tied to one specific thread (each thread has their own copies of those variables).
- `SCHEDULE_HPC` is replaced by the schedule[1] we want.
- `NUM_THREADS_HPC` is corresponding to the number of threads to use for `parallel` regions.

```
#pragma omp parallel for schedule(SCHEDULE_HPC) default(shared) private(i, j) \
    num_threads(NUM_THREADS_HPC)
```

In addition, inside `norm` function, we add a reduction clause `reduction(+ : norm)` as we want to sum up every `norm` from each thread to one final `norm` and taking square of that final sum. Finally, we have to add `#pragma omp atomic` above each line that updating the result matrix (`C`). It is because that matrix is shared among threads, `atomic` makes sure that there is only one `+=` operation (which is essentially reading and writing) on one specific pair of indices at a given time. Note that `norm` does not need atomicity thank to `reduction`.

Here, we show a comparision between with and without `OpenMP`. Default `OpenMP` options will be `SCHEDULE_HPC = static` and `NUM_THREADS_HPC = 4`.

Thank to `OpenMP`, naive approach is faster than 0,37 times while the `saxpy_dot` took less 0,25 times than before. Both approachs performance are significantly improved.

[1] https://www.openmp.org/spec-html/5.0/openmpse49.html#x288-20520006.1

Table 3: naive vs saxpy with OpenMP

algo	time	norm	gflops	omp
naive	44.61439	2.323362	0.3863321	x
naive	120.06615	2.323362	0.1430886	
saxpy	14.42142	2.323362	1.1913447	x
saxpy	57.37033	2.323362	0.2995959	

2.1.4 Cache blocking (Tiled)

The main idea of the cache blocking technique (or tiled) is breaking the whole matrices into smaller sub-matrices so the data needed for one multiplication operation could fit into the cache, therefore leads to a much faster calculation. Furthermore, if we enable `OpenMP`, the computation would be even faster as each sub-matrice is processed by a separate thread. However, if we set `BLOCK` size too small, the benefit of dividing matrix is overshadowed by the additional loops and operations. Meanwhile, a too large `BLOCK` size leads to an overfitting (data for one operation can not be fitted into the cache), and therefore a slower operation. The principal source code is shown below:

```
for (k = 0; k < K; k += BLOCK)
  for (j = 0; j < N; j += BLOCK)
    for (i = 0; i < M; i += BLOCK)
      for (kk = 0; kk < BLOCK; kk++)
        for (jj = 0; jj < BLOCK; jj++)
          for (ii = 0; ii < BLOCK; ii++)
            C[(ii + i) + ldc * (jj + j)] +=
                A[(ii + i) + lda * (kk + k)] * B[(kk + k) + ldb * (jj + j)];
```

The above code will work only if `M`, `N` and `K` are divisible by `BLOCK`. A more generic version could be found in full source-code.

We have a table comparision between all techniques we are dicussing so far below. Also, we set the default size of `BLOCK = 4`.

Table 4: naive vs saxpy vs tiled

algo	time	norm	gflops	omp
naive	44.61439	2.323362	0.3863321	x
naive	120.06615	2.323362	0.1430886	
saxpy	14.42142	2.323362	1.1913447	x
saxpy	57.37033	2.323362	0.2995959	
tiled	15.62542	2.323362	1.0995711	x
tiled	15.61854	2.323362	1.1000269	

In the table above, cache blocking technique is already fast enough. However, `OpenMP` does not help speeding it up as the default `BLOCK` size is not optimized in this case.

2.1.5 BLAS

One last technique that is used in our code is calling the `cblas_dgemm` function which use the optimized `BLAS` implementation. This function is the fastest method even if other methods are *"cheated"* (by using `OpenMP`) as their implementation is optimized based on many factors: algorithms, software and hardware.

Table 5: all techniques with default options

algo	time	norm	gflops	omp
naive	44.614388	2.323362	0.3863321	x
naive	120.066146	2.323362	0.1430886	
saxpy	14.421419	2.323362	1.1913447	x
saxpy	57.370325	2.323362	0.2995959	
tiled	15.625424	2.323362	1.0995711	x
tiled	15.618538	2.323362	1.1000269	
blas	5.986271	2.323362	2.8699601	

2.2 Benchmarks

2.2.1 Sequential

In this section, we fix `NUM_THREADS_HPC = 1` and vary the matrix size. Instead of using environmental variables, we use a script for generating code with the hard-coded configurations we want as reading environmental variables is an expensive operation.

For the sake of simplicity, we first consider the case where M and K and N are all equal and equal to a 2^s where `s = 2` to `s = 11`. In addition, we have included a non `OpenMP` result (which is also sequential) for studying how the overhead time of `OpenMP` impacts the overall performance.

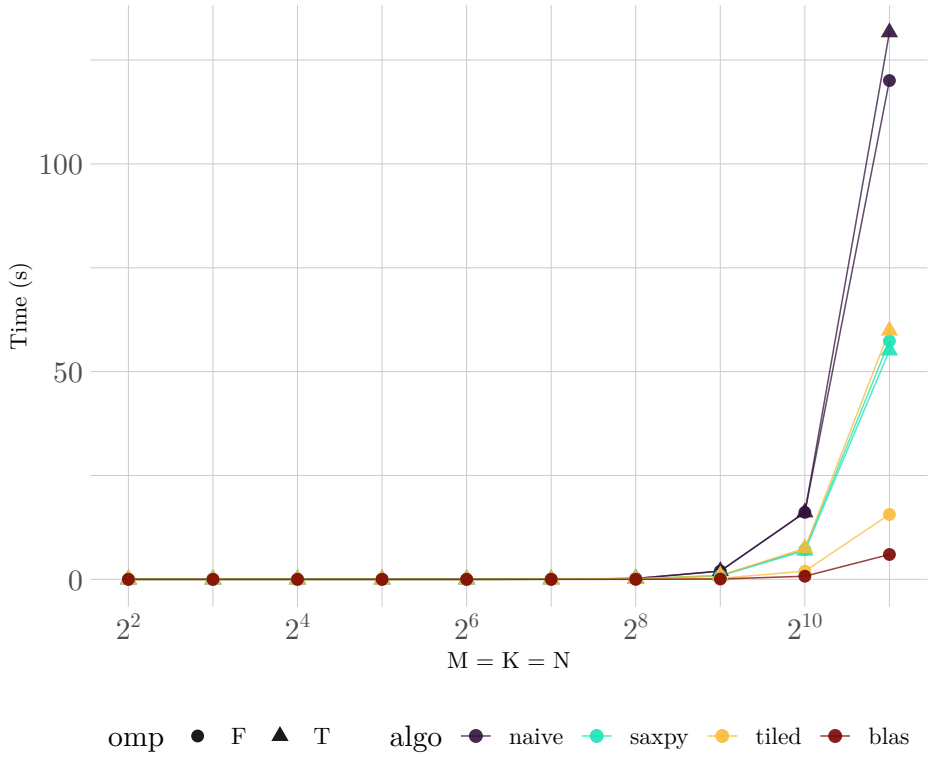


Figure 1: Computational time in function of matrix size

In the graph below, we see that the fastest method is no doubt `blas` method, followed by `tiled`, `saxpy` and the slowest is `naive`. This is aligned with what we see in the section 1. In addition, the time for calculating

Table 6: Computation time when M = N = K = 2048

algo	time	omp
naive	131.680261	x
naive	120.066146	
saxpy	55.097613	x
saxpy	57.370325	
tiled	59.886455	x
tiled	15.618538	
blas	5.986271	

matrices whose size is less than $2^{10} = 1024$ is around $5s$ for all methods. This could be explained by the fact that these matrices could be fitted entierly into the cache, which leads to a significant drop in computation time.

Another property that could be interesting is the version with `OpenMP` is close or even faster than the non `OpenMP` version regardless the overhead of parallelization. This could be explained by many factors [2] [3], but the most significant one is As `OpenMP` is just API specification and C compilers are free to implement it in any way they want as long as they respect the specification, many compilers (notably modern `gcc` and `clang`) are smart enough to treat `OpenMP` version of only 1 thread the same as the sequential version. Therefore, we only see a small difference between each run.

2.2.2 Threading

Right now, we will be able to see the true power of parallelism, we will keep increasing the number of threads in form of 2^s where `s = 1` to `s = 7`.

[2] https://stackoverflow.com/questions/22927973/openmp-with-single-thread-versus-without-openmp
[3] https://stackoverflow.com/questions/2915390/openmp-num-threads1-executes-faster-than-no-openmp

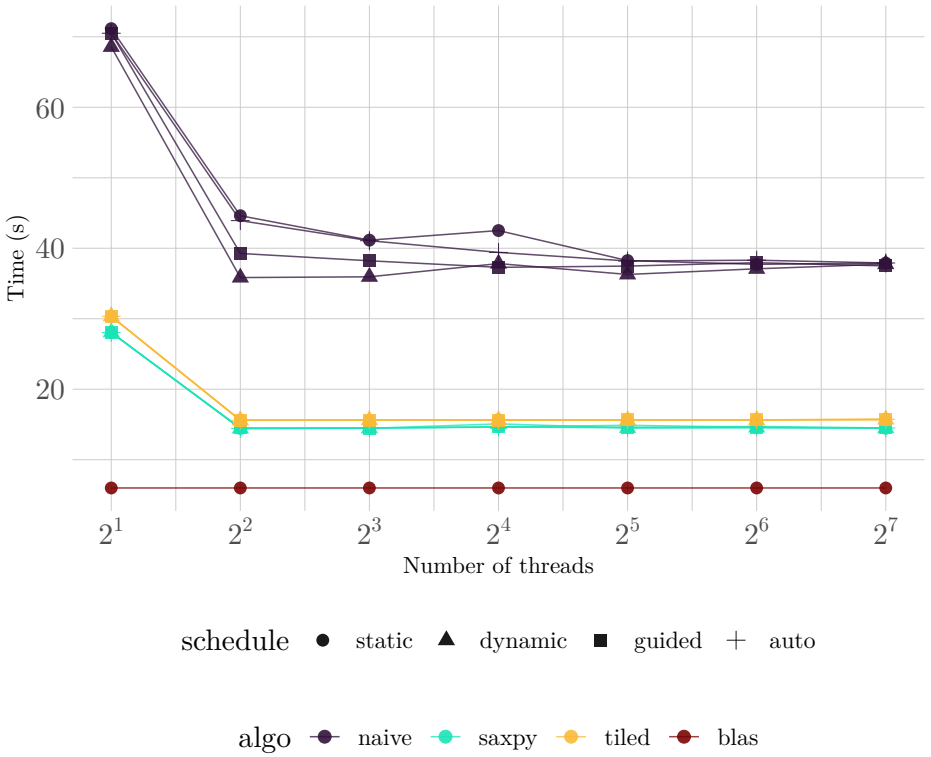


Figure 2: Computational time in function of number of threads and schedule

We see that `BLAS` method is still the fastest regardless the number of threads and schedule (since it isn't affected by `OpenMP` options). It shows that in order to achieve high speed computation, we have to not only parallelize, but also make improvements on multiplication algorithms, memory accesses and even use assembly instructions.

In addition, the 4 schedule cuvres of each technique are overlapping each others and there are only very small difference in term of computational time. The phenomenon happened because our problem (matrix multiplication) has a nearly the same workload at each iterations. That means the first iteration will take almost the same as the last iteration or any other iterations. For each schedule:

- `static` evenly-divides the total workloads into each threads, which is the best schedule for our problem.
- `dynamic` and `guided` are designed for different situation, where each iteration takes different amount of time to finish their work. There is overhead compared to `static`, however, it does not have big effect on overall performance as our matrices are not too big.
- `auto` lets the compiler choose how to schedule and divide work among threads, so it is compiler-specific. For example, `gcc` maps `auto` to `static` [4], at a consequence, we see a similar pattern with `static`.

Finally, more threads **doesn't** always mean better performance. After we increased thread to 4, time taking for one multiplication fluctuates but does not have any real decline. The reason is there is only a limit number of physical cores inside each computer, when the number of threads goes up too high, the overhead in creating and synchronize threads will overshadowed any benefits we gain.

[4] https://github.com/gcc-mirror/gcc/blob/61e53698a08dc1d9a54d785218af687a6751c1b3/libgomp/loop.c#L195-L198

2.2.3 Blocking

In the last section, we will concentrate ourselves on the impact of `BLOCK` size to overall performance. We will vary the `BLOCK` size in a power of 2, from `s = 0` to `s = 10`

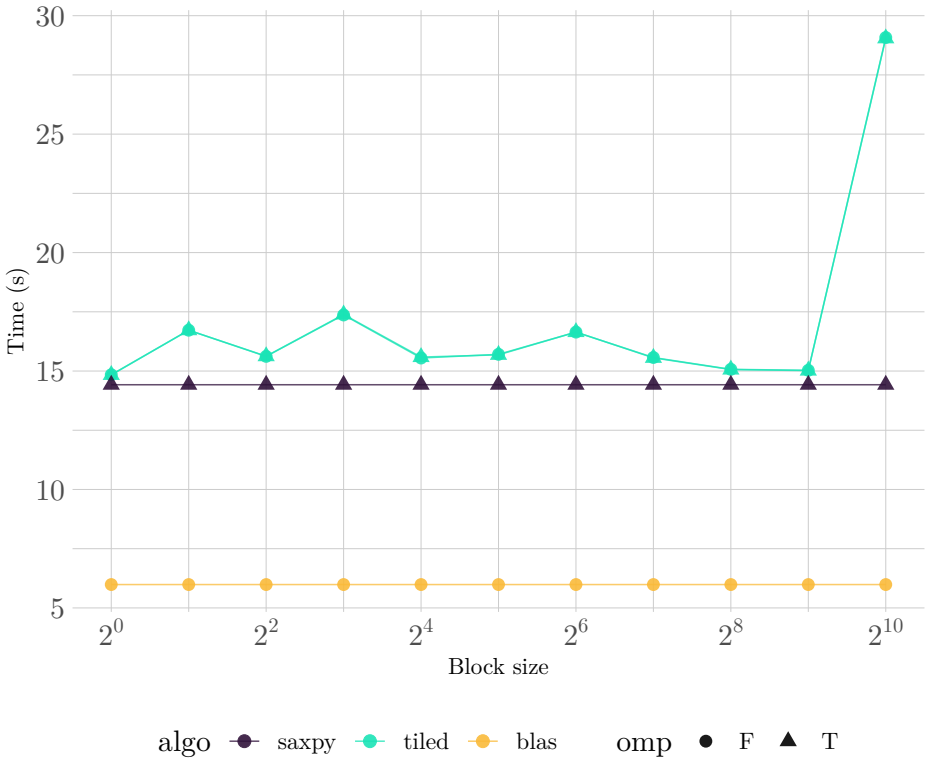


Figure 3: Computational time in function of BLOCK size

We see clearly that as `BLOCK` size grows, the performance generally becomes better but get worse after `BLOCK` size grows to approximately 2^{10}. As explained in the section 1, `BLOCK` should not be too small and neither too large.

3 MPI

In this section, we will focus on the optimization and parallelization of Google PageRank problem, more precisely, the calculation of the eigenvectors associated with the highest eigenvalue of a stochastic matrix that has some special attributes.

3.1 Introduction

First of all, we mention briefly the problem. Given a web page P and a measure of its importance $I(P)$, called the page's PageRank. Suppose that page P_i has ℓ_i links. If one of those links is to page P_j, then P_i will pass on $frac1\ell_i$ of its importance to P_j. The importance ranking of P_j is then the sum of all the contributions made by pages linking to it. That is, if we denote the set of pages linking to P_j by B_j,

$$I(P_j) = \sum_{P_i \in B_j} \frac{I(P_i)}{\ell_i}$$

Mathematically, if we define a matrix H where:

$$H_{i,j} = \begin{cases} \frac{1}{\ell_i} & \text{if } P_i \in B_j \\ 0 & \text{otherwise} \end{cases}$$

H is called "hyperlink matrix" and we have $\pi^\top = \pi^\top H$ where $\pi_i = I(P_i)$, a vector contains measure of importance from all pages. In the other words, $(\pi^\top)^\top = (\pi^\top H)^\top \implies \pi = H^\top \pi$ or π is an eigenvector of $H^\top$ whose eigenvalue is 1.

In addition, we could prove that the largest eigenvalue of $H^\top$ less than or equal to 1. Indeed, if λ is an eigenvalue of $H^\top$ and u is its eigenvector (we will choose a vector whose the sum of all of its components is positive), as we can see all entries of H are non negative and the sum of of each row is either 1 or 0 (if that page has no link to other page), we have:

$$\begin{aligned}
& H^\top u = \lambda u \\
\iff & \begin{pmatrix} H_{1,1} & \dots & H_{n,1} \\ \vdots & \ddots & \vdots \\ H_{1,n} & \dots & H_{n,n} \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \lambda u_1 \\ \vdots \\ \lambda u_n \end{pmatrix} \\
\implies & \begin{cases} H_{1,1}u_1 + \dots + H_{n,1}u_n & = \lambda u_1 \ (1) \\ & \vdots \\ H_{1,n}u_1 + \dots + H_{1,n}u_n & = \lambda u_n \ (n) \end{cases} \\
& \text{By summing up from (1) to } (n) \\
\implies & u_1(\sum_{i=1}^{n} H_{1,i}) + \dots + u_n(\sum_{i=1}^{n} H_{n,i}) = \lambda(\sum_{i=1}^{n} u_i) \\
& \text{As } \sum_{i=1}^{n} H_{j,i} = \text{ either 0 or 1 } \forall j \\
\implies & \sum_{i=1}^{n} u_i \geq \lambda(\sum_{i=1}^{n} u_i) \\
\implies & \lambda \leq 1
\end{aligned}$$

Futhermore, if $\sum_{i=1}^{n} H_{j,i} = 1 \ \forall j$, we have $\lambda = 1$ is the largest eigenvalue of $H^\top$. Therefore, we return to the problem of finding the eigenvector associated with the largest eigenvalue of $H^\top$, which already has a very simple but powerful solution called "Power Iteration" that we will use in next sections. Each component of that vector is a measure of importance of one page and the index of the largest element inside that vector will also be the index of the most meaningful page.

3.2 Power Iteration Method

First, we choose a matrix H as following for testing:

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

As proved about, since every sum of a row of H is equal to 1, the largest eigenvalue will be 1 and therefore, we could apply this method to find out the PageRank score of each page.

The method could be described as follow:

Data: A diagonalizable matrix G

Result: Largest eigenvalue and associated eigenvector

$u \leftarrow random$;

$u \leftarrow \frac{u}{\|u\|}$;

while $\|u - v\| > \epsilon$ **do**

  compute $v = Gu$;

  define $\lambda = u^\top v$ and $u = \frac{v}{\|v\|}$;

end

And its implementation in `Python`:

```
def power_iteration(matvec, G, u, v, tol=1e-8, itmax=200):
    t = time.time()
    it = 0
    diff_norm = 2 * tol
    while it < itmax and diff_norm > tol:
        v = matvec(G, u)
        l = dot_product(u, v)
        v = (1 / norm2(v)) * v
        diff_norm = norm2(u - v)
        u = v
        it += 1
    return l, v
n = H.shape[0]
u = np.random.uniform(0, 1, n)
v = np.zeros(n)
l, u = power_iteration(matvec, H.T, u, v)
```

Here, we choose to keep the original power iteration method by calculating $v = Gu$ instead of $v = G^\top u$. It helps us prevent some confusions. Since we want to calculate the eigenvector of $H^\top$, we could pass $G \leftarrow H^\top$. From the output below, the method has successfully converged to 1 and the most meaningful page is the page number 8.

```
## number of iterations = 115
## residual = 8.87e-09
## eigenvalue = 1.000000e+00
## eigenvector = [0.14  0.158 0.07  0.158 0.228 0.473 0.42  0.689]
## highest pagerank = 7
## Computational time = 0.003090381622314453
```

In addition, we check the correctness by calling the function `eigs` from `scipy.sparse.linalg`. As we can see, the two outputs are the similar. Our implementation of power iteration is correct.

```
from scipy.sparse.linalg import eigs
vals, vecs = eigs(H.T)
val, vec = vals[0].real, vecs[:, 0].real
print("Largest eigenvalue ", val)
```

```
## Largest eigenvalue  1.0000000000000018
```

```
print("Associated eigenvector ", vec)
```

```
## Associated eigenvector  [-0.14  -0.158 -0.07  -0.158 -0.228 -0.473 -0.42  -0.689]
```

3.3 Google matrix

In general, there is no guarantee that the algorithm will work. As we can see in the introduction, if one web page does not link to any other page or also called as a dangling node, the sum of that row in the hyperlink matrix will be 0, the largest eigenvalue will be eventually < 1 and therefore we fail. To address this problem, we introduce another type of matrix, so-called Google matrix and use this matrix for calculating the PageRank. It has the following property:

$$G = \alpha(H + \frac{1}{n}de^\top) + (1-\alpha)\frac{1}{n}ee^\top$$

Where

H : a very sparse hyperlink matrix.

α : a scaling parameter between 0 and 1 (generally set at the "magic" value of 0.85).

d : the binary dangling node row vector ($d_i = 1$ if page i is a dangling node and 0 otherwise).

$e^\top$: the row vector of all entries 1.

We could easily prove that G is stochastic, i.e, sum of each row equals to 1. Indeed,

$$\sum_{i=1}^{n} G_{i,j} = \alpha(\sum_{i=1}^{n} H_{i,j} + \frac{1}{n}\sum_{1}^{n} d_j) + (1-\alpha)$$

$$\implies \sum_{i=1}^{n} G_{i,j} = \alpha((\sum_{i=1}^{n} H_{i,j}) + d_j) + 1 - \alpha$$

Since one of $\sum_{i=1}^{n} H_{i,j}$ or d_j equals to 1 and the other term equals to zero

$$\implies \sum_{i=1}^{n} G_{i,j} = \alpha + 1 - \alpha$$

$$\implies \sum_{i=1}^{n} G_{i,j} = 1$$

We have two approaches for this problem.

3.3.1 Dense

In this approach, we compute in the "traditional" way, i.e, we construct the matrix G and pass it to the power iteration method. The output is shown below.

```
## number of iterations =  54
## residual = 7.99e-09
```

```
## eigenvalue = 1.000000e+00
## eigenvector = [0.159 0.233 0.115 0.245 0.277 0.464 0.394 0.632]
## highest pagerank = 7
## Computational time = 0.0007550716400146484
```

3.3.2 Sparse

On the other hand, we could take advantage of two factors: the sparsity of H and the nature of the matrices $de^\top$ and $ee^\top$. First, we note that we want to calculating the eigenvector of

$$G^\top = \alpha(H^\top + \frac{1}{n}ed^\top) + (1-\alpha)\frac{1}{n}ee^\top$$

The matrix multiplication between $G^\top$ and x turns out to be

$$G^\top x = \alpha H^\top x + \alpha\frac{1}{n}ed^\top x + (1-\alpha)\frac{1}{n}ee^\top x$$

Futhermore, give a an arbitrary column vector,

$$\begin{aligned} ea^\top x &= \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \begin{pmatrix} a_1 & \dots & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \\ &= \begin{pmatrix} a_1 & \dots & a_n \\ \vdots & \ddots & \vdots \\ a_1 & \dots & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \\ &= \begin{pmatrix} a_1x_1 + \dots + a_nx_n \\ \vdots \\ a_1x_1 + \dots + a_nx_n \end{pmatrix} \\ &= \langle a, x \rangle \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \end{aligned}$$

Therefore, instead of calculating $ed^\top x$ and $ee^\top x$, we could just calculate two scalars $\langle d, x\rangle$ and $\langle e, x\rangle$. Thank to the fact that in `Python` world, an operation between an array and a scalar will be resulted in an element-wise operation, we have our desired matrix multiplication. Its implementation in `Python` is:

```
def matvec_sparse(M, x, d, alpha=0.85):
    m, n = np.shape(M)
    e = np.ones(m)
    return (
        alpha * M.dot(x)
        + alpha / n * d.dot(x)
        + (1 - alpha) / n * e.dot(x)
    )
```

The output of this method is shown below.

```
## number of iterations =  54
## residual = 7.99e-09
## eigenvalue = 1.000000e+00
## eigenvector = [0.159 0.233 0.115 0.245 0.277 0.464 0.394 0.632]
## highest pagerank = 7
## Computational time = 0.001833200454711914
```

3.3.3 Benchmarking

We see the difference between two methods in term of performance is quite small, given that our matrix is too tiny. We varied the initial matrix H and observeb that, while the sparse approach performs worse in small cases, it gets better when the size of H increases. That could be explained by the fact that the density of H (number of non-zero elements) decreases when the size of H is bigger (based on the external data). Since 0 is a really special number, a sparsity-aware matrix could take advantage of that fact and perform a lot faster. In our case, the speed-up comes from two factors:

- `M.dot(x)` as $M \leftarrow H$ is a sparse matrix, which means `M` contains only non-zero entries. Therefore while normal algorithm have to do operations with every elements of `M` and `x`, the sparse algorithm will only doing operations between those elements of `M` and `x` (0 multiplies something is 0 anyway) which reduce significanly the number of operations required. if we convert H to a normal matrix beforehand, the performance will be similar to the dense approach.
- When the size of H is too big to be fitted within the RAM, we have an Out Of Memory (OOM) if we try the first approach. The last matrix `ucam2006`, the dense version needs roughly 361,97 GB which is impossible on almost computers. On the opposite side, the second approach only needs a fraction of that amount of memory to store the matrix H since its density is relatively low.

Below is a table that shown the results above where:

- density: the number of non-zero elements of H per mille.
- shape: the number of rows of H (and also of columns because H is a square matrix).
- important: the most meaningful page, we show it here just for asserting the true result.
- memory: the amount of RAM memory in GB that is needed to fit the dense version of that matrix.

Table 7: dense vs sparse approach

algo	time	matrix	shape	important	density	memory
dense	0.0006697	random10	10	4	200.00 ‰	0.00
sparse	0.0013359	random10	10	4	200.00 ‰	
dense	0.0042232	random100	100	90	20.00 ‰	0.00
sparse	0.0023179	random100	100	90	20.00 ‰	
dense	0.0195672	random1000	1000	980	2.00 ‰	0.01
sparse	0.0075064	random1000	1000	980	2.00 ‰	
dense	1.0330197	random10000	10000	6253	0.20 ‰	0.80
sparse	0.0063826	random10000	10000	6253	0.20 ‰	
dense	Inf	ucam2006	212711	-1	0.04 ‰	361.97
sparse	0.3676185	ucam2006	212711	2	0.04 ‰	

3.4 MPI-enabled

Now, since, we has already taken advantage of the sparsity of matrix for optimizing the power iteration method, algorithm-level is now harder to optimize. However, we could try to fully exploit our hardware by parallelization, which will be easier. In this section, we concentrate ourselves in `MPI`, one technology that will allows us to achieve that.

3.4.1 Principal functions

First, we will integrate `MPI` to 4 principal functions: $\|\cdot\|_1$, $\|\cdot\|_2$, $\langle\cdot,\cdot\rangle$ and `matvec` (which is essentially matrix multiplication with the second matrix is a vector).

For the first 3 functions, the principle is simple, we calculate them with a subset of values in each process and reduce them into one at the end. For example, $\langle \cdot, \cdot \rangle$ could be calculated as follow:

$$\begin{aligned}
\langle x, y \rangle &= \sum_{i=1}^{kn} x_i y_i \\
&= \sum_{i=1}^{k} \sum_{j=(i-1)n+1}^{ni} x_j y_j \\
&= \sum_{i=1}^{k} \underbrace{\langle \begin{pmatrix} x_{(i-1)n+1} \\ \vdots \\ x_{in} \end{pmatrix}, \begin{pmatrix} y_{(i-1)n+1} \\ \vdots \\ y_{in} \end{pmatrix} \rangle}_{\text{calculate in process } i}
\end{aligned}$$

Our implementation of those 3 functions with `MPI` is shown below. Note that `u` and `v` is only a subset of the real vectors `u` and `v`. However, since we reduce the result of each process in the end and send it to all processes by `Allreduce`, we still get the true value of these operations on vectors `u` and `v` regardless process.

```
def norm1_mpi(u, comm=MPI.COMM_WORLD):
    norm = norm1(u)
    result = np.empty(1)
    comm.Allreduce(norm, result, MPI.SUM)
    return result


def norm2_mpi(u, comm=MPI.COMM_WORLD):
    norm = dot_product(u, u)
    result = np.empty(1)
    comm.Allreduce(norm, result, MPI.SUM)
    return np.sqrt(result)


def dot_product_mpi(u, v, comm=MPI.COMM_WORLD):
    product = dot_product(u, v)
    result = np.empty(1)
    comm.Allreduce(product, result, MPI.SUM)
    return result
```

For `matvec`, everything get more complicated, we note that, given a matrix H and u, if we note $v = Hu$, we have:

$$v_i = \sum_{j=1}^{n} H_{i,j} u_j$$

Therefore, the vector that passed into `matvec` must be a "full" vector.

3.4.2 Power Iteration

With all those principles in mind, we are ready now for a `MPI`-enabled power iteration. The general idea will be:

Data: A diagonalizable matrix G whose size is kn and k processes
Result: Largest eigenvalue and associated eigenvector
In process i
  $v_i \leftarrow$ random
  gather $u \leftarrow \begin{pmatrix} v_1 & \dots & v_k \end{pmatrix}$ (each process has their own copy of u)
Define
  $G_i = \begin{pmatrix} G_{(i-1)n+1,\cdot} & \dots & G_{in,\cdot} \end{pmatrix}^\top$
while $\|u_i - v\| > \epsilon$ **do**
  define G_i and u_i
  compute $v_i = G_i u$ or in other world $v_i = \begin{pmatrix} [Gu]_{(i-1)n+1} & \dots & [Gu]_{in} \end{pmatrix}^\top$
  by using MPI-enabled functions above
    gather $v \leftarrow \begin{pmatrix} v_1 & \dots & v_k \end{pmatrix}$
    define $\lambda = u^\top v$ and $v_i = \frac{v_i}{\|v\|}$
  gather $u \leftarrow \begin{pmatrix} v_1 & \dots & v_k \end{pmatrix}$
end

Dense

For the dense approach, the main logic is implemented as follow:

```
        v = matvec(G[start:end], u)
        l = dot_product(u[start:end], v, comm)
        v = (1 / norm2(v, comm)) * v
        diff_norm = norm2(u[start:end] - v, comm)
        u = mpi_all_to_all(u, v, comm=comm)
```

Inside the code above, `matvec` is just a normal matrix multiplication. The step gather `v` above is executed inside `dot_product` and `norm2` functions. `mpi_all_to_all` is the function we used in order to gather `u` at the end with the help of `Allgather` or `Allgatherv` underlying.

Sparse

However, the sparse method requires a slightly more complicated solution. It has one different line from the the first version.

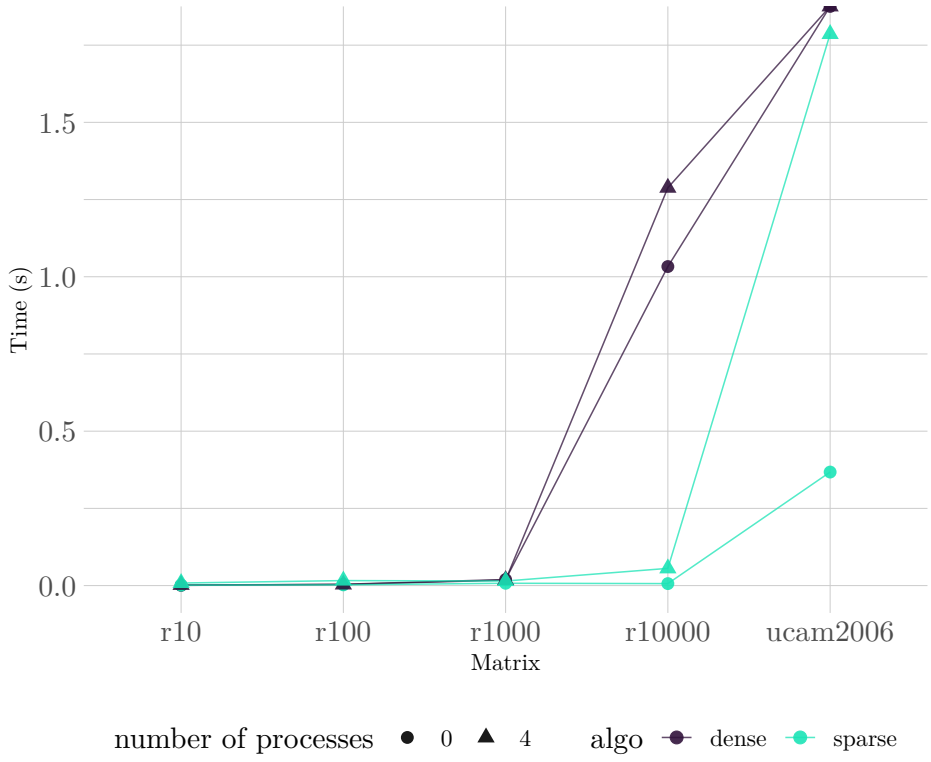
```
        v = matvec(G.T, u, d, start, end)
        l = dot_product(u[start:end], v, comm)
        v = (1 / norm2(v, comm)) * v
        diff_norm = norm2(u[start:end] - v, comm)
        u = mpi_all_to_all(u, v, comm=comm)
```

Where `matvec` is

```
def matvec_sparse(M, x, d, start, end, alpha=0.85, comm=MPI.COMM_WORLD):
    M = M[start:end]
    d = d[start:end]
    xd = x[start:end]
    m, n = np.shape(M)
    e = np.ones(m)
    return (
        alpha * M.dot(x)
        + alpha / n * dot_product(d, xd, comm)
        + (1 - alpha) / n * dot_product(e, xd, comm)
    )
```

Although, `dot_product` produces a global result, it is still true because it is a scalar, not a vector as show above.

3.4.3 Benchmarking



First, both versions (non `MPI` and `MPI`) using the dense algorithm is defeated by the matrix `ucam2006` as explained above. Second, interestingly, when number of processes is 4, it took more time than non `MPI` version. This could be explained by the fact that `MPI` is optimized for a distributed cluster of machines but we are testing on only one computer, hence, the communication overhead overshaddowed any benefit it brought to us.

3.5 Further study: Matrix B

This section, we explore another matrix that has some special constructions and we try to exploit them for a faster computation.

3.5.1 Introduction

We define matrix B size kn as below (with k processes and n is the size of the submatrix that is owned processed in each process):

$$B^\top = \begin{pmatrix} & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{kn-1} & & \frac{1}{3} & & & \\ \frac{1}{kn-1} & \frac{1}{2} & & \ddots & & \\ \frac{1}{kn-1} & & \frac{1}{3} & & \frac{1}{3} & \\ \vdots & & & \ddots & & \frac{1}{2} \\ \frac{1}{n-1} & & & & \frac{1}{3} & \end{pmatrix}$$

$B^\top$ has the following non-zeros entries:

- the first column: consits of 0 followed by $n-1$ elements whose value is $\frac{1}{n-1}$.
- the first row: consits of 0 followed by an array that has $\frac{1}{2}$ at its head and tail; $\frac{1}{3}$ at other positions.
- the first superdiagonal: the same array as the first row without the leading 0.

- the first subdiagonal: the same array as the first row except its tail, and the leading 0 is replaced by $\frac{1}{n-1}$.

For example, with k = 3 and n = 3, we have $B^\top$:

$$\begin{pmatrix}
0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \frac{1}{2} \\
\frac{1}{8} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{8} & \frac{1}{2} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{8} & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\
\frac{1}{8} & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 \\
\frac{1}{8} & 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\
\frac{1}{8} & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 \\
\frac{1}{8} & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{2} \\
\frac{1}{8} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0
\end{pmatrix}$$

And its corresponding matrix transition:

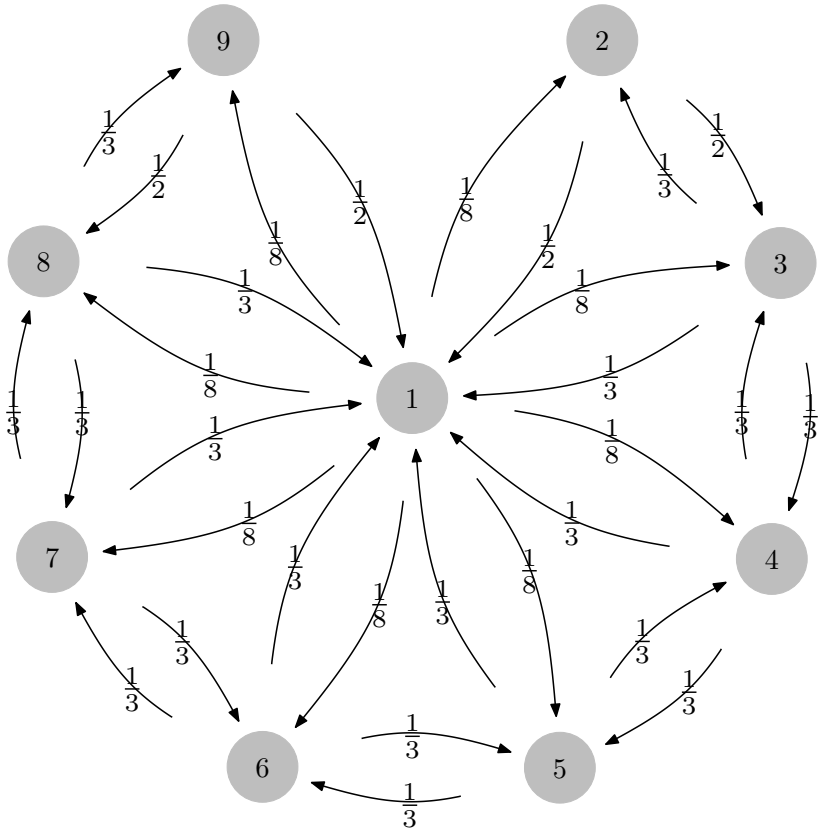


Figure 4: matrix transition of B when kn = 9

We could see easily that:

- Matrix B is non negative.
- Matrix B is stochastic $\forall kn$.
- It doesn't contain any dangling node.
- From 3 observations above, we conclude that largest eigenvalue of $B^\top$ is 1.
- Page 1 receives $\frac{1}{2}$ or $\frac{1}{3}$ of the importance of other pages while only share $\frac{1}{8}$ of its. Intuitively, it is safe to guess that page 1 is the most important page.

That assumption is verified by the output followed.

```
## number of iterations =  35
## residual = 9.45e-09
## eigenvalue = 1.000000e+00
## eigenvector = [0.713 0.178 0.267 0.267 0.267 0.267 0.267 0.267 0.178]
## highest pagerank = 0
## Computational time = 0.0010652542114257812
```

3.5.2 MPI-enabled

Same as the Google matrix, we tried to exploit the special properties of $B^\top$. As before, we first try to divide $B^\top$ into n submatrices, each with n rows and kn columns. This is however, the same as the dense version and we lost the sparsity of $B^\top$. But we notice that apart from the top-most submatrix whose the first row contains only 1 0, the other submatrices are quite sparse. Therefore, we have three types of matrix here:

- The first submatrix: nothing could be done here.
- The last submatrix:
  - There are n elements $\frac{1}{kn-1}$.
  - There are n elements of the first subdiagonal which lasts from $(kn-n)^{\text{th}}$ column to $(kn-1)^{\text{th}}$ column.
  - There are $n-1$ elements of the first superdiagonal which lasts from $(kn-n+2)^{\text{th}}$ column to kn^{th} column.
  - From three observations above, we could remove a block from 2^{nd} column to $(kn-n-1)^{\text{th}}$ column which is all zero, which left us a submatrix of size $n \times (n+2)$.
- The i^{th} submatrix:
  - There are n elements $\frac{1}{kn-1}$.
  - There are n elements of the first subdiagonal which lasts from $(in)^{\text{th}}$ column to $(in+n-1)^{\text{th}}$ column.
  - There are n elements of the first superdiagonal which lasts from $(in+2)^{\text{th}}$ column to $(in+n+1)^{\text{th}}$ column.
  - From three observations above, we only take the first column and the block from $(in)^{\text{th}}$ column to $(in+n+1)^{\text{th}}$ column and remove everything else, which left us a submatrix of size $n \times (n+3)$.

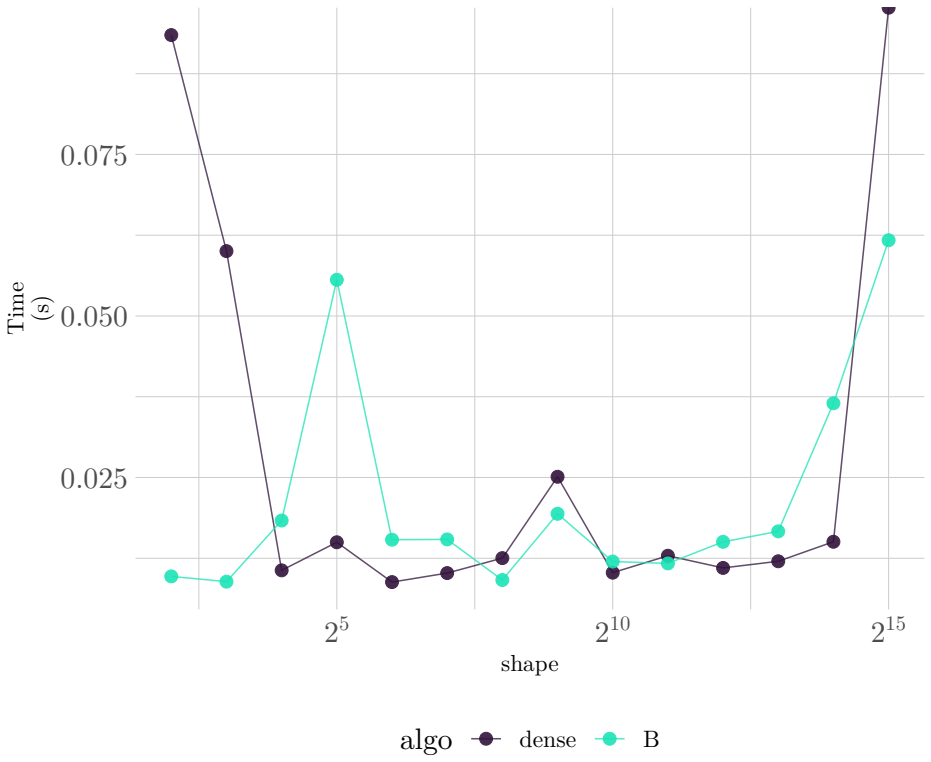
In order to multiply matrix and vector with the modified submatrices, we have to modified u as well because the length of u needs to be the same as the number of columns of each submatrix inside each process (which is constant in the previous methods). It could be done easily by noting that even if we don't remove those blocks, they still contribute nothing to the result, as they are all zeros. Therefore, we could extract the corresponding indices from u, this is how we could take advantage of the sparsity of $B^\top$.

There is one more problem to solve. We needs a modified gathering function. Note that apart from the first submatrix, vector u inside other processes only contain a subset of the real u (which is opposed to the previous method where u is constant in size). Hence, we dicided to send everything to the process that holds the first submatrix in order to assemble from v the real u, broadcast that to every other processes and then let each processed extracting the subset of u they need. The exact code is shown below:

```
def mpi_all_to_all(x, xd, comm=MPI.COMM_WORLD):
    rank = comm.rank
    size = comm.size
    n = np.shape(xd)[0]
    comm.Gather(xd, x, root=0)
    xbcast = np.empty(n * size) if rank != 0 else x
    comm.Bcast(xbcast, root=0)
    if rank == size - 1:
```

```
        x[0] = xbcast[0]
        x[1 : n + 2] = xbcast[n * rank - 1 : n * rank + n]
    elif rank != 0:
        x[0] = xbcast[0]
        x[1 : n + 3] = xbcast[n * rank - 1 : n * rank + n + 1]
    return x
```

3.5.3 Benchmarking



Compared to the dense version, the $B^\top$-optimized version takes less time to compute in general, which means our algorithm is correct.

4 Conclusion

Throughout the report, we see that HPC and speed-optimization is a hard problem. It does not only depend on the software, the main problem but also the environment we are running the computation. We would like to end the report with a very famous quote relating to it:

> ***Premature optimization is the root of all evil.***