Optimized Power Method

Hikaru N. Belzer

November 1, 2024

Introduction

The purpose of this report is to compare the results obtained using Basic Linear Algebra Subprograms (BLAS) in the utility functions with those from HW 3b, where we implemented a parallel version of the power method in C. For context, the power method is used to compute approximations to the largest eigenvalue in magnitude and associated eigenvector of a matrix. For this assignment, we are using the utility functions from the previous report that included a parallel version of the power method.

We used the standard initial guess for the eigenvector $x \in R^n$ with components $x_j = 1/\sqrt{n}$ for all $0 \leq j < n$. Our power method function returns the eigenvalue approximation λ, the eigenvector approximation x, and the number of iterations taken. We used the same n-values that we used in HW 3b (1024, 2048, 4096, 8192, 16384, 32768, 65536, and 131072). After conducting a test for each n-value, we compared the new λ values to the values we obtained in HW 3b and verified that they matched.

Methodology

Implementing BLAS

We used a strategy involving #ifdef BLAS ... #else ... #endif to implement an alternative version using BLAS in our existing utility functions. We revisited all of our previous utility functions and made the following changes:

- Added an MPI_COMM_WORLD parameter to the normE() function
- Added a BLAS implementation to the serial_dot() function
- Edited the Makefile to handle BLAS
- Added the BLAS2 implementation to the matvec() function in utilities.c (which is the highest order BLAS we can use for matrix-vector multiplication)

Immediately, we observed that adding the MPI_COMM_WORLD parameter to the normE function was beneficial for parallel processing. We adjusted the corresponding function calls for this as well to include the new parameter. Here is how the updated function appeared:

```
double normE (double *l_x, int l_n, MPI_Comm comm) {
  return sqrt(parallel_dot(l_x, l_x, l_n, comm));
}
```

Next, we will discuss the changes we made to the serial_dot() function. Previously, we used a for-loop to perform the multiplication, but now, we are using cblas_ddot(n, x, 1, y, 1). Here is the updated function:

```
double serial_dot (double *x, double *y, long n)
{
  double dp;
#ifndef BLAS
  long i;
#endif

#ifdef BLAS
  dp = cblas_ddot(n, x, 1, y, 1);
#else
  dp = 0.0;
  for(i = 0; i < n; i++)
    dp += x[i] * y[i];
#endif

  return dp;
}
```

The above function calculates the dot product of the vectors x and y and takes several parameters: n specifies the number of elements in the vectors, x is a pointer to the first vector, 1 is the increment for accessing elements in x, y is a pointer to the second vector, and 1 is the increment for accessing elements in y.

Next, we will address the two changes we made to the Makefile to handle BLAS. First, in the CFLAGS section, we added -qmkl, which is a flag that is used when compiling with the Intel compiler to enable the use of the Intel Math Kernel Library (MKL). Then, in the DEFS section, we added -DBLAS, which is a flag that informs the compiler that the BLAS functionality will be used in our code. These changes were added below:

```
# choose flags:
# flags for Intel compiler icc on taki:
CFLAGS := -O3 -std=c99 -Wall -qmkl
# flags for GNU compiler gcc anywhere:
# CFLAGS := -O3 -std=c99 -Wall -Wno-unused-variable

DEFS := -DPARALLEL -DBLAS
INCLUDES :=
LDFLAGS := -lm
```

Lastly, we will describe the changes we made to the matvec() function. As we discussed in class, changing the matvec() function to incorporate BLAS2 is the only change that will have a substantial difference in our overall timing results. So, we edited the matvec() function as follows:

```
void matvec (double *l_y, double *l_A, double *l_x,
             int n, int l_n, int id, int np, double *partial_y, double *y) {

  int i, l_j;

  /* Step 1: local matrix-vector product partial_y = l_A * l_x: */
#ifdef BLAS
  // printf("Using BLAS for matvec\n");
  cblas_dgemv(CblasColMajor, CblasNoTrans,
              n, l_n, 1.0, l_A, n, l_x, 1, 0.0, partial_y, 1);
#else
  for (i = 0; i < n; i++)
    partial_y[i] = 0.0;
  for (l_j = 0; l_j < l_n; l_j++) {
    for (i = 0; i < n; i++) {
      partial_y[i] += l_A[i + n * l_j] * l_x[l_j];
    }
  }
#endif

  /* Step 2: reduce all partial_y to y with MPI_SUM: */
  MPI_Allreduce(partial_y, y, n, MPI_DOUBLE, MPI_SUM,
  //MPI_COMM_WORLD);

  /* Step 3: scatter y to l_y on each process: */
  // MPI_Scatter(y, l_n, MPI_DOUBLE, l_y, l_n, MPI_DOUBLE, 0,
     MPI_COMM_WORLD);
  for (int l_j=0; l_j < l_n; l_j++) {
    l_y[l_j] = y[id*l_n+l_j];
  }
}
```

Here, the function checks whether BLAS is being used. If it is, it utilizes cblas_dgemv, which is a BLAS2 implementation. As discussed in the report for HW 3b, we determined that we should always use the highest level BLAS available to achieve optimal timing results. So, we are using BLAS2 because it is the highest order BLAS that can handle matrix-vector products.

The cblas_dgemv call takes 12 parameters: CblasColMajor specifies that the matrix is stored in column-major order, CblasNoTrans means the matrix should not be transposed, n is the number of columns in the matrix, l_n is the number of rows in the matrix, 1.0 is the scalar multiplier for the matrix-vector product, l_A is the pointer to the matrix data, n is the number of rows in the matrix, l_x is the pointer to the input vector, 1 is the increment for the elements of the input vector, 0.0 is the scalar used to scale the result before adding to partial_y, partial_y is the pointer to the output vector where the result will be stored, and 1 is the increment for the elements of the output vector.

The #else case will only run when BLAS is not being used, and Step 2 and Step 3 are the same from HW 3b. We want to see how implementing BLAS will affect our timing results.

Performance Studies

Timing Results

For our tables, we included our timing results, in seconds (rounded to 3 decimal places) with 1 to 16 nodes and 1 to 32 processes per node for each n-value. We will show the results from HW 3b and compare them to the new results. At the end, we will compare the speedup and efficiency plots and make conclusions.

Results from HW 3b: Wall clock time in seconds expressed in decimal notation (rounded to 3 decimal places).

n = 1024	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.029	0.019	0.004	0.008
2 processes per node	0.005	0.003	0.004	0.004
4 processes per node	0.002	0.002	0.001	0.002
8 processes per node	0.002	0.001	0.001	0.002
16 processes per node	0.002	0.001	0.001	0.002
32 processes per node	0.003	0.001	0.001	0.003
n = 2048	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.067	0.032	0.013	0.023
2 processes per node	0.069	0.013	0.006	0.007
4 processes per node	0.011	0.006	0.004	0.014
8 processes per node	0.007	0.004	0.003	0.024
16 processes per node	0.004	0.011	0.011	0.044
32 processes per node	0.004	0.027	0.019	0.027
n = 4096	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.326	0.167	0.083	0.048
2 processes per node	0.154	0.199	0.034	0.023
4 processes per node	0.075	0.040	0.015	0.023
8 processes per node	0.039	0.022	0.008	0.021
16 processes per node	0.023	0.013	0.014	0.028
32 processes per node	0.016	0.024	0.018	0.042
n = 8192	1 node	2 nodes	4 nodes	16 nodes
1 process per node	1.461	0.730	0.447	0.292
2 processes per node	1.760	0.369	0.204	0.106
4 processes per node	0.358	0.220	0.111	0.060
8 processes per node	0.175	0.107	0.048	0.041
16 processes per node	0.093	0.057	0.031	0.032
32 processes per node	0.075	0.056	0.035	0.046
n = 16384	1 node	2 nodes	4 nodes	16 nodes
1 process per node	6.500	5.119	2.078	1.787
2 processes per node	3.240	1.872	1.001	0.523
4 processes per node	3.817	0.952	0.482	0.253
8 processes per node	1.903	0.453	0.188	0.143
16 processes per node	0.392	0.233	0.117	0.102
32 processes per node	0.318	0.174	0.093	0.079

n = 32768				
	1 node	2 nodes	4 nodes	16 nodes
1 process per node	34.347	23.401	9.892	8.131
2 processes per node	16.955	8.867	5.110	2.955
4 processes per node	8.035	4.976	2.417	1.214
8 processes per node	3.892	2.099	0.796	0.601
16 processes per node	1.925	0.912	0.478	0.310
32 processes per node	1.492	0.695	0.361	0.283
n = 65536				
	1 node	2 nodes	4 nodes	16 nodes
1 process per node	143.611	135.197	52.136	26.087
2 processes per node	77.718	40.774	22.292	14.114
4 processes per node	38.190	22.960	11.977	6.218
8 processes per node	47.108	11.126	5.324	3.125
16 processes per node	9.734	5.685	2.449	1.694
32 processes per node	7.759	3.839	1.694	1.272
n = 131072				
	1 node	2 nodes	4 nodes	16 nodes
1 process per node	1263.125	603.240	235.410	132.327
2 processes per node	501.062	691.333	106.301	61.961
4 processes per node	199.273	93.962	54.744	28.386
8 processes per node	144.364	48.335	25.271	16.887
16 processes per node	45.672	30.069	13.071	11.653
32 processes per node	34.680	18.602	9.938	6.852

New Results Using BLAS: Wall clock time in seconds expressed in decimal notation (rounded to 3 decimal places). This uses the full code that is provided in the Appendix:

n = 1024				
	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.047	0.034	0.028	0.473
2 processes per node	0.013	0.027	0.025	0.427
4 processes per node	0.021	0.032	0.020	0.033
8 processes per node	0.031	0.029	0.028	0.030
16 processes per node	0.036	0.028	0.010	0.034
32 processes per node	0.018	0.005	0.340	0.033
n = 2048				
	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.062	0.034	0.021	0.030
2 processes per node	0.042	0.039	0.028	0.037
4 processes per node	0.036	0.037	0.035	0.043
8 processes per node	0.639	0.043	0.545	0.048
16 processes per node	0.020	0.032	0.035	0.037
32 processes per node	0.030	0.053	0.042	0.032

n = 4096	1 node	2 nodes	4 nodes	16 nodes
1 process per node	0.259	0.146	0.085	0.593
2 processes per node	0.201	0.108	0.042	0.025
4 processes per node	0.072	0.044	0.043	0.055
8 processes per node	0.050	0.068	0.043	0.039
16 processes per node	0.056	0.034	0.059	0.045
32 processes per node	0.032	0.080	0.055	0.908

n = 8192	1 node	2 nodes	4 nodes	16 nodes
1 process per node	1.166	0.572	0.776	0.034
2 processes per node	0.585	0.453	0.557	0.043
4 processes per node	0.308	0.177	0.189	0.044
8 processes per node	0.167	0.101	0.023	0.048
16 processes per node	0.107	0.152	0.051	0.040
32 processes per node	0.106	0.079	0.072	0.056

n = 16384	1 node	2 nodes	4 nodes	16 nodes
1 process per node	4.915	2.425	2.259	0.842
2 processes per node	2.544	1.288	0.665	0.209
4 processes per node	1.254	0.641	0.337	0.053
8 processes per node	0.670	0.339	0.488	0.046
16 processes per node	0.412	0.501	0.138	0.112
32 processes per node	0.350	0.183	0.134	0.042

n = 32768	1 node	2 nodes	4 nodes	16 nodes
1 process per node	23.330	10.978	5.942	3.529
2 processes per node	12.100	5.797	3.548	1.829
4 processes per node	5.549	2.815	0.753	0.990
8 processes per node	3.170	1.467	0.858	0.070
16 processes per node	1.797	0.905	0.527	0.058
32 processes per node	1.513	0.720	0.402	0.057

n = 65536	1 node	2 nodes	4 nodes	16 nodes
1 process per node	129.117	50.811	28.798	18.696
2 processes per node	61.605	38.083	16.870	10.151
4 processes per node	32.871	13.687	8.291	4.454
8 processes per node	12.582	6.675	3.189	2.741
16 processes per node	7.969	4.171	1.913	1.453
32 processes per node	7.785	10.488	1.899	1.248

n = 131072	1 node	2 nodes	4 nodes	16 nodes
1 process per node	688.629	252.152	141.801	55.104
2 processes per node	297.432	110.267	72.259	21.820
4 processes per node	103.746	56.540	33.545	8.433
8 processes per node	71.412	35.107	18.204	4.486
16 processes per node	41.633	20.688	10.468	2.776
32 processes per node	27.355	17.537	9.060	2.648

Comparing the Results from the HW 3b Code and BLAS Code

To begin, the tables for $n = 1024, 2048, 4096$, and 8192 are very similar, which is expected because these are smaller calculations. The runtime was already around 0 to 1 seconds, so we expected that implementing BLAS would have no significant difference.

However, for $n = 32768, 65536$, and 131072, there are noticeable differences. For example, in the 2 nodes column, we are seeing speedup from 23.401, 8.867, and 4.976 (from the old results) to 10.978, 5.797, and 2.815 (from the new results). Similarly, for $n = 65536$, we are observing speedup in the 2 nodes column. Throughout the table, we also see that the times are slightly faster.

For $n = 131072$, we are seeing the most noticeable results. In the old results, we had 1263.125 seconds for the serial run, and in the new run, we had 688.629 seconds, which is almost half the time. Also, for 2 nodes and 1 process per node, we are seeing a speedup from 603.240 seconds to 252.152 seconds. We are not seeing runtimes in the 500 and 600-second range as we did in the old table. So, we can see that the BLAS implementation did have an impact, specifically for the larger n-values.

Speedup and Efficiency

These tables provide additional analysis related to the timing results. Specifically, the observed speedup (S_p) and observed efficiency (E_p). The equation for the observed speedup is $S_p = T_1/T_p$, and the equation for the observed efficiency is $E_p = S_p/p$. First, we will show the tables from HW 3b that were produced using the old code. Then, we will show new tables that were produced using the updated code:

Results from HW 3b:

(a) Wall clock time T_p in seconds (rounded to 3 decimal places)

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	0.029	0.005	0.002	0.002	0.002	0.003
2048	0.067	0.069	0.011	0.007	0.004	0.004
4096	0.326	0.154	0.075	0.039	0.023	0.016
8192	1.461	1.760	0.358	0.175	0.093	0.075
16384	6.500	3.240	3.817	1.903	0.392	0.318
32768	34.347	16.955	8.035	3.892	1.925	1.492
65536	143.611	77.718	38.190	47.108	9.734	7.759
131072	1263.125	501.062	199.273	144.364	45.672	34.680

(b) Observed speedup $S_p = T_1/T_p$

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.00	5.80	14.50	14.50	14.50	9.67
2048	1.00	0.97	6.09	9.57	16.75	16.75
4096	1.00	2.12	4.35	8.36	14.17	20.38
8192	1.00	0.83	4.08	8.35	15.71	19.48
16384	1.00	2.01	1.70	3.42	16.58	20.44
32768	1.00	2.03	4.27	8.83	17.84	23.02
65536	1.00	1.85	3.76	3.05	14.75	18.51
131072	1.00	2.52	6.34	8.75	27.66	36.42

(c) Observed efficiency $E_p = S_p/p$						
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	1.00	2.90	3.62	1.81	0.91	0.30
2048	1.00	0.49	1.52	1.20	1.05	0.52
4096	1.00	1.06	1.09	1.04	0.89	0.64
8192	1.00	0.42	1.02	1.04	0.98	0.61
16384	1.00	1.00	0.43	0.43	1.04	0.64
32768	1.00	1.01	1.07	1.10	1.12	0.72
65536	1.00	0.92	0.94	0.38	0.92	0.58
131072	1.00	1.26	1.58	1.09	1.73	1.14

Results Using BLAS:

(a) Wall clock time T_p in seconds (rounded to 3 decimal places)						
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	0.047	0.013	0.021	0.031	0.036	0.018
2048	0.062	0.042	0.036	0.639	0.020	0.030
4096	0.259	0.201	0.072	0.050	0.056	0.032
8192	1.166	0.585	0.308	0.167	0.107	0.106
16384	4.915	2.544	1.254	0.670	0.412	0.350
32768	23.330	12.100	5.549	3.170	1.797	1.513
65536	129.117	61.605	32.871	12.582	7.969	7.785
131072	688.629	297.432	103.746	71.412	41.633	27.355

(b) Observed speedup $S_p = T_1/T_p$						
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	1.00	3.62	2.24	1.52	1.31	2.61
2048	1.00	1.48	1.72	0.10	3.10	2.07
4096	1.00	1.29	3.60	5.18	4.62	8.09
8192	1.00	1.99	3.79	6.98	10.90	11.00
16384	1.00	1.93	3.92	7.34	11.93	14.04
32768	1.00	1.93	4.20	7.36	12.98	15.42
65536	1.00	2.10	3.93	10.26	16.20	16.59
131072	1.00	2.32	6.64	9.64	16.54	25.17

(c) Observed efficiency $E_p = S_p/p$						
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
1024	1.00	1.81	0.56	0.19	0.08	0.08
2048	1.00	0.74	0.43	0.01	0.19	0.06
4096	1.00	0.64	0.90	0.65	0.29	0.25
8192	1.00	1.00	0.95	0.87	0.68	0.34
16384	1.00	0.97	0.98	0.92	0.75	0.44
32768	1.00	0.96	1.05	0.92	0.81	0.48
65536	1.00	1.05	0.98	1.28	1.01	0.52
131072	1.00	1.16	1.66	1.21	1.03	0.79

Comparisons

By comparing tables (a), it is clear that there is some speedup for $n = 65536$ and 131072, as we discussed in the previous section. By looking at tables (b), we see a mixture of results. For example, at $n = 131072$ with $p = 16$ and 32, we see observed speedup. However, in the middle of both tables, we see some larger values, which means there was not a significant difference across every single trial.

However, for observed efficiency (table (c)), we see that all of the values are close to 1 or 0 in the new code. The old code produced results that were slightly less efficient. So, we can conclude that BLAS contributed to improving efficiency, at least to an extent.

Plots for Speedup and Efficiency

Now that we have produced the tables, here are plots showing the speedup and efficiency data along with an optimal line on both graphs. Additionally, we used MATLAB to produce these graphs, and the full MATLAB script is provided in the Appendix.

The first plot is "Observed Speedup (Sp) vs Number of Parallel Processes (p)" using the results from HW 3b:

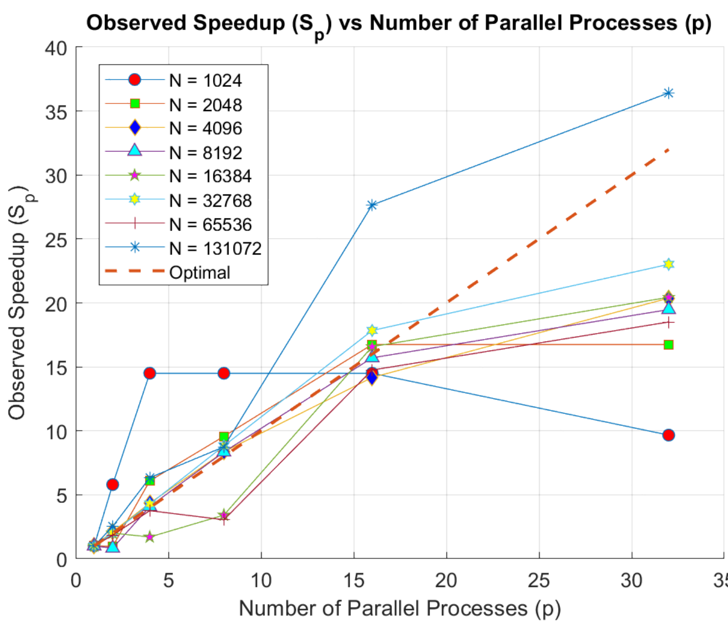

Figure 1: Observed speedup S_p

The second plot is “Observed Efficiency (Ep) vs Number of Parallel Processes (p)” using the results from HW 3b:

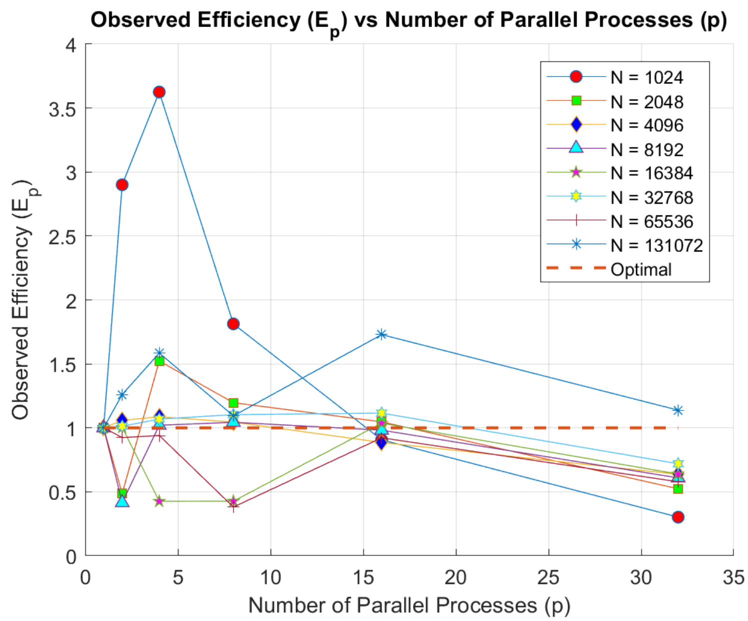

Figure 2: Observed efficiency E_p

Timing Results Using BLAS

The first plot is “Observed Speedup (Sp) vs Number of Parallel Processes (p)” using the results from the BLAS implementation:

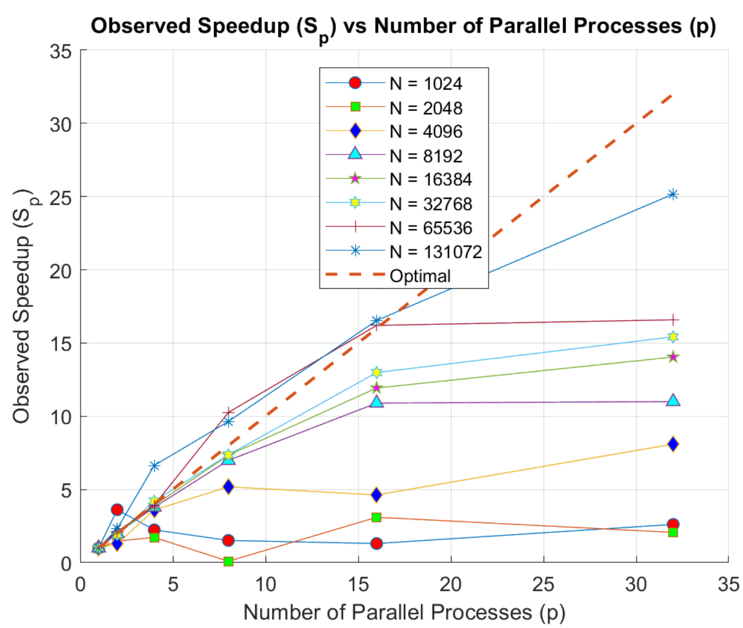

Figure 3: Observed speedup S_p

The second plot is "Observed Efficiency (Ep) vs Number of Parallel Processes (p)" using the results from the BLAS implementation:

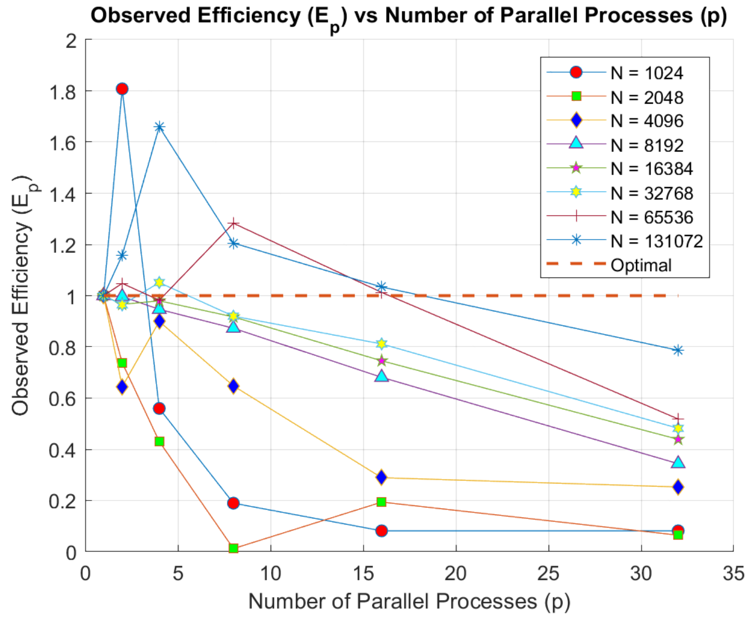

Figure 4: Observed efficiency E_p

Comparisons

By looking at both observed speedup graphs, it is clear that the curves appear more horizontal in the new plot. Since the runtimes for the first few n-values were around 0 to 1 seconds, it makes sense that the difference between the trials is not that large. However, we do see that for larger values of n, the curves more closely align with the optimal line even though they level off. Since the differences are more noticeable for larger n-values, these results make sense.

When comparing the observed efficiency plots, we see more variety. We see that the curves for $n = 1024$ and 2048 are fairly different from the optimal line. We had some values, such as 0.08, that were close to 0, which is why the curves appear this way. But, we do see that the larger values of n resemble the optimal line more closely. In conclusion, using BLAS led to some improvements, but outcomes varied depending on the trials.

Conclusion

By implementing BLAS in our code, we did notice speedup, especially for larger values of n. However, for the speedup and efficiency plots, we saw varied results. But, overall, using BLAS can make a difference when running code with very large n-values, since the differences become more evident with those larger values.

Acknowledgments

Department of Mathematics and Statistics: UMBC

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258, CNS-1228778, OAC-1726023, and CNS-1920079) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

Appendix

utilities.c file with the BLAS implementation:

```
#include "utilities.h"
#include "memory.h"

double serial_dot (double *x, double *y, long n)
{
  double dp;
#ifndef BLAS
  long i;
#endif

#ifdef BLAS
  // printf("BLAS for serial_dot\n");
  dp = cblas_ddot(n, x, 1, y, 1);
#else
printf("No Blas");
  dp = 0.0;
  for(i = 0; i < n; i++)
    dp += x[i] * y[i];
#endif

  return dp;
}

double parallel_dot(double *l_x, double *l_y, long l_n, MPI_Comm comm) {

  double l_dot=0.0;
  double dot=0.0;

  l_dot=serial_dot(l_x,l_y,l_n);

#ifdef PARALLEL
  MPI_Allreduce(&l_dot, &dot, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
#else
  dot = l_dot;
#endif

  return dot;

}

double parallel_norm2 (double *l_x, long l_n, MPI_Comm comm) {

  return sqrt(parallel_dot(l_x, l_x, l_n, comm));

}
/*
// New function to set up matrices A and B
void setup_matrices(double *A, double *B, int m, int k, int n) {

  // Set up matrix A (m x k) with example values
  for (int i = 0; i < m; i++) {
    for (int q = 0; q < k; q++) {
```

```
      A[i + q * m] = (double)(i + 1);
      // Example: A(i,q) = i + 1
    }
  }

  // Set up matrix B (k x n) with example values
  for (int q = 0; q < k; q++) {
    for (int j = 0; j < n; j++) {
      B[q + j * k] = (double)(10 * (q + 1) + j + 1);
      // Example: B(q,j) = 10*(q+1) + j+1
    }
  }
}
*/


void matvec (double *l_y, double *l_A, double *l_x,
             int n, int l_n, int id, int np, double *partial_y, double *y) {

  int i, l_j;

  /* Step 1: local matrix-vector product partial_y = l_A * l_x: */
#ifdef BLAS
  // printf("Using BLAS for matvec\n");
  cblas_dgemv(CblasColMajor, CblasNoTrans,
              n, l_n, 1.0, l_A, n, l_x, 1, 0.0, partial_y, 1);

#else
  for (i = 0; i < n; i++)
    partial_y[i] = 0.0;
  for (l_j = 0; l_j < l_n; l_j++) {
    for (i = 0; i < n; i++) {
      partial_y[i] += l_A[i + n * l_j] * l_x[l_j];
    }
  }
#endif

  /* Step 2: reduce all partial_y to y with MPI_SUM: */
  MPI_Allreduce(partial_y, y, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  /* Step 3: scatter y to l_y on each process: */
  // MPI_Scatter(y, l_n, MPI_DOUBLE, l_y, l_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for (int l_j=0; l_j < l_n; l_j++) {
    l_y[l_j] = y[id*l_n+l_j];
  }
}


void print_vector (double *l_x, int n, int l_n, int id, int np) {
  int i;
  double *x;

  if (id == 0) {
    x = allocate_double_vector(n);
  }
```

```
  MPI_Gather(l_x, l_n, MPI_DOUBLE, x, l_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  if (id == 0) {
    for (i = 0; i < n; i++) {
      printf("%.6f\n", x[i]);
    }
    free(x);
  }
}

void print_matrix (double* l_A, int n, int l_n, int id, int np) {
  double* A;

  if (id == 0) {
    A = allocate_double_vector(n * n);
  }

  MPI_Gather(l_A, l_n * n, MPI_DOUBLE, A, l_n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  if (id == 0) {
    printf("Matrix A = \n");
    for (int j = 0; j < n; j++) {
      for (int i = 0; i < n; i++) {
        printf("%.6f ", A[i + n * j]);
      }
      printf("\n");
    }
    free(A);
  }
}

double normE (double *l_x, int l_n, MPI_Comm comm) {
  return sqrt(parallel_dot(l_x, l_x, l_n, comm));
}
```

MATLAB Code for Plots (MATH447_HW4b.m):

```
% Processes per node
p = [1, 2, 4, 8, 16, 32];
N_1024 = [0.047, 0.013, 0.021, 0.031, 0.036, 0.018];
N_2048 = [0.062, 0.042, 0.036, 0.639, 0.020, 0.030];
N_4096 = [0.259, 0.201, 0.072, 0.050, 0.056, 0.032];
N_8192 = [1.166, 0.585, 0.308, 0.167, 0.107, 0.106];
N_16384 = [4.915, 2.544, 1.254, 0.670, 0.412, 0.350];
N_32768 = [23.330, 12.100, 5.549, 3.170, 1.797, 1.513];
N_65536 = [129.117, 61.605, 32.871, 12.582, 7.969, 7.785];
N_131072 = [688.629, 297.432, 103.746, 71.412, 41.633, 27.355];

% Calculate Sp for each N
Sp_1024 = N_1024(1) ./ N_1024;
Sp_2048 = N_2048(1) ./ N_2048;
Sp_4096 = N_4096(1) ./ N_4096;
Sp_8192 = N_8192(1) ./ N_8192;
Sp_16384 = N_16384(1) ./ N_16384;
Sp_32768 = N_32768(1) ./ N_32768;
Sp_65536 = N_65536(1) ./ N_65536;
Sp_131072 = N_131072(1) ./ N_131072;

% Calculate Ep for each N
Ep_1024 = Sp_1024 ./ p;
Ep_2048 = Sp_2048 ./ p;
Ep_4096 = Sp_4096 ./ p;
Ep_8192 = Sp_8192 ./ p;
Ep_16384 = Sp_16384 ./ p;
Ep_32768 = Sp_32768 ./ p;
Ep_65536 = Sp_65536 ./ p;
Ep_131072 = Sp_131072 ./ p;

% Print Sp values with & separator (to 2 decimal places)
fprintf('Sp values:\n');
fprintf('N = 1024: '); fprintf('%.2f & ', Sp_1024(1:end-1));
fprintf('%.2f\n', Sp_1024(end));
fprintf('N = 2048: '); fprintf('%.2f & ', Sp_2048(1:end-1));
fprintf('%.2f\n', Sp_2048(end));
fprintf('N = 4096: '); fprintf('%.2f & ', Sp_4096(1:end-1));
fprintf('%.2f\n', Sp_4096(end));
fprintf('N = 8192: '); fprintf('%.2f & ', Sp_8192(1:end-1));
fprintf('%.2f\n', Sp_8192(end));
fprintf('N = 16384: '); fprintf('%.2f & ', Sp_16384(1:end-1));
fprintf('%.2f\n', Sp_16384(end));
fprintf('N = 32768: '); fprintf('%.2f & ', Sp_32768(1:end-1));
fprintf('%.2f\n', Sp_32768(end));
fprintf('N = 65536: '); fprintf('%.2f & ', Sp_65536(1:end-1));
fprintf('%.2f\n', Sp_65536(end));
fprintf('N = 131072: '); fprintf('%.2f & ', Sp_131072(1:end-1));
fprintf('%.2f\n', Sp_131072(end));

% Print Ep values with & separator (to 2 decimal places)
fprintf('Ep values:\n');
fprintf('N = 1024: '); fprintf('%.2f & ', Ep_1024(1:end-1));
fprintf('%.2f\n', Ep_1024(end));
```

```
fprintf('N = 2048: '); fprintf('%.2f & ', Ep_2048(1:end-1));
fprintf('%.2f\n', Ep_2048(end));
fprintf('N = 4096: '); fprintf('%.2f & ', Ep_4096(1:end-1));
fprintf('%.2f\n', Ep_4096(end));
fprintf('N = 8192: '); fprintf('%.2f & ', Ep_8192(1:end-1));
fprintf('%.2f\n', Ep_8192(end));
fprintf('N = 16384: '); fprintf('%.2f & ', Ep_16384(1:end-1));
fprintf('%.2f\n', Ep_16384(end));
fprintf('N = 32768: '); fprintf('%.2f & ', Ep_32768(1:end-1));
fprintf('%.2f\n', Ep_32768(end));
fprintf('N = 65536: '); fprintf('%.2f & ', Ep_65536(1:end-1));
fprintf('%.2f\n', Ep_65536(end));
fprintf('N = 131072: '); fprintf('%.2f & ', Ep_131072(1:end-1));
fprintf('%.2f\n', Ep_131072(end));

% S_p = p represents optimal behavior according to HPCF{2019{1 (pg. 8)
optimal = p;

% Plot Observed Speedup (Sp) vs Number of Parallel Processes (p)
figure;
hold on;
plot(p, Sp_1024, '-o', 'DisplayName', 'N = 1024', 'MarkerFaceColor', 'r'); % Circle
plot(p, Sp_2048, '-s', 'DisplayName', 'N = 2048', 'MarkerFaceColor', 'g'); % Square
plot(p, Sp_4096, '-d', 'DisplayName', 'N = 4096', 'MarkerFaceColor', 'b'); % Diamond
plot(p, Sp_8192, '-^', 'DisplayName', 'N = 8192', 'MarkerFaceColor', 'c'); % Triangle
plot(p, Sp_16384, '-p', 'DisplayName', 'N = 16384', 'MarkerFaceColor', 'm'); % Pentagon
plot(p, Sp_32768, '-h', 'DisplayName', 'N = 32768', 'MarkerFaceColor', 'y'); % Hexagon
plot(p, Sp_65536, '-+', 'DisplayName', 'N = 65536', 'MarkerFaceColor', 'k'); % Plus
plot(p, Sp_131072, '-*', 'DisplayName', 'N = 131072', 'MarkerFaceColor', 'r'); % Star
plot(p, optimal, '--', 'DisplayName', 'Optimal', 'LineWidth', 1.5);  % Dashed line
xlabel('Number of Parallel Processes (p)');
ylabel('Observed Speedup (S_p)');
title('Observed Speedup (S_p) vs Number of Parallel Processes (p)');
legend('Location', 'best');
grid on;
hold off;

% Plot Observed Efficiency (Ep) vs Number of Parallel Processes (p)
figure;
hold on;
plot(p, Ep_1024, '-o', 'DisplayName', 'N = 1024', 'MarkerFaceColor', 'r'); % Circle
plot(p, Ep_2048, '-s', 'DisplayName', 'N = 2048', 'MarkerFaceColor', 'g'); % Square
plot(p, Ep_4096, '-d', 'DisplayName', 'N = 4096', 'MarkerFaceColor', 'b'); % Diamond
plot(p, Ep_8192, '-^', 'DisplayName', 'N = 8192', 'MarkerFaceColor', 'c'); % Triangle
plot(p, Ep_16384, '-p', 'DisplayName', 'N = 16384', 'MarkerFaceColor', 'm'); % Pentagon
plot(p, Ep_32768, '-h', 'DisplayName', 'N = 32768', 'MarkerFaceColor', 'y'); % Hexagon
plot(p, Ep_65536, '-+', 'DisplayName', 'N = 65536', 'MarkerFaceColor', 'k'); % Plus
plot(p, Ep_131072, '-*', 'DisplayName', 'N = 131072', 'MarkerFaceColor', 'r'); % Star
plot(p, optimal ./ p, '--', 'DisplayName', 'Optimal', 'LineWidth', 1.5);  % Dashed line
xlabel('Number of Parallel Processes (p)');
ylabel('Observed Efficiency (E_p)');
title('Observed Efficiency (E_p) vs Number of Parallel Processes (p)');
legend('Location', 'best');
grid on;
hold off;
```