CPS343 Parallel and High-Performance Computing **Project 3**

Introduction

In order to improve performance of a Power Method algorithm for finding the dominant eigenvalue of an $n \times n$ matrix we can split the work of the across multiple nodes in a SMP cluster. This project sought to utilize MPI to communicate between nodes running small pieces of the algorithm. Each individual node also used OpenMP to parallelize their portion of the work.

The bulk of the Power Methods computation takes the form of a matrix vector multiplication, computed one time each loop. Because this function can be parallelized with relative ease, we find reasonable speedup can be achieved by parallelizing the Power Method with OpenMP. However, as the size of the matrix increases this speedup is quickly overshadowed by the overhead required to read more data into the array after cache misses. Therefore, it is crucial that a way is found to prevent this performance degradation for large matrices.

In order to improve performance, we need to limit the number of cache misses. One way to do this is to compute the matrix-vector product in chunks, giving each chunk to a SMP cluster and using MPI for communication between the nodes at several key points. For testing, the C++ code was run on two different clusters; the Canaan cluster and the Minor Prophets cluster. Slurm (Simple Linux Utility for Resource Management) was used to allocate processors on which to run the program.

Tests were run on a 10000x10000 matrix. While the serial version of the program took approximately 30 seconds on the Canaan cluster, utilizing MPI to run the program on all 180 clusters resulted in a massive drop to 0.477 seconds, a 97% improvement. The Minor Prophets cluster took 14 seconds to finish the iterations in the serial implementation, but only took 2.6 seconds on its 44 processors, an 85% improvement.

These tests prove that proper utilization of MPI can result in many less cache misses, drastically improving performance for large matrices. If these methods were to be implemented in a production system, a serial or OMP version of the program should be used for small matrices, and this approach should be used for larger matrices so that the cache can be utilized more efficiently.

Overview

The specific algorithm that was parallelized is as follows:
If A_r is a process's piece of the $n \times n$ matrix, $\epsilon > 0$ is a tolerance, $M > 0$ is the maximum allowed number of iterations, λ is the eigenvalue estimate, λ_0 is the previous eigenvalue estimate, y_r is a process's part of the eigenvector estimate, and x is the full previous eigenvector estimate, then the Power Method algorithm proceeds as follows:

```
$A_r, f_r, l_r \leftarrow read\ matrix\ from\ HDF5\ file$
$n_r := l_r - f_r + 1$
$F \leftarrow allgather(f_r)$    $array\ of\ offsets\ to\ first\ rows\ in\ each\ process$
$N \leftarrow allgather(n_r)$    $array\ of\ number\ of\ rows\ assigned\ to\ each\ process$
$x := (1, 1, 1, \ldots, 1)^T$
$x := x/||x||$
$\lambda := 0$
$\lambda_0 := \lambda + 2\epsilon$
$k := 0$
while $|\lambda - \lambda_0| \geq \epsilon$ and $k \leq M$ do
    $k := k + 1$
    $y_r := A_r x$
    $\lambda_0 := \lambda$
    $\lambda_r := x_r^T y_r$
    $\alpha_r := y_r^T y_r$
    $\lambda \leftarrow allreduce(\lambda_r)$
    $\alpha \leftarrow allreduce(\alpha_r)$
    $y_r := y_r / \sqrt{\alpha}$
    $x \leftarrow allgatherv(y_r, N, F)$
end while
```

Because each process is running its own version of the program, all processes need to call the function to read their chunk of the matrix A. The function that reads this file also returns the first and last rows of the original matrix that this process is using; f_r and l_r, respectively. We can then compute the number of rows that this process is using, storing that value in n_r. Later in the program, we will need to gather each process's piece of the eigenvector (y_r)and store it in the shared vector *x*, since the entire *x* vector is required for the matrix-vector multiplication. To prepare for this, we create two arrays: one containing each process's offset to the first row of the matrix, and the other containing the number of rows assigned to each process. Since every process needs to use the entire *x* vector, we initialize and normalize it the same way as in the serial version. After entering the while loop, we compute y_r, which is this process's chunk of the new eigenvector estimate. To complete this, we multiply A_r, our section of the matrix, with the full *x* vector. We can then find the partial sum $\lambda_r$by computing the dot product of $x_r$and y_r, and the partial normalization factor by finding the dot product of y_r with itself. In order to use these values, we need to sum each process's $\lambda_r$and $\alpha_r$values. Finally, we normalize $y_r$using our completeαvalue. Finally, we combine each process's $y_r$values into the single *x* vector, which is shared to all processes.

Implementation

To allow communications between discrete processes we use the Message Passing Interface (MPI), a standardized way to communicate between processes running a program across several nodes. For C++, we include the `<mpi.h>` header file. The program is then compiled with mpic++. We begin by making the following three function calls:

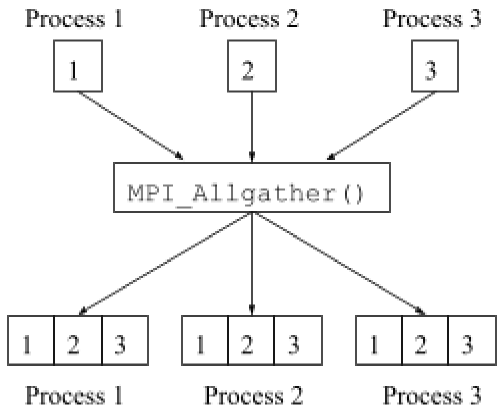
```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &num_blocks);
```

The first call initializes the MPI environment, the second gets the current processor's number (its rank), and the third returns the total number of processes that are running in this environment. Each process needs part of the HDF5 data file that holds the matrix *A*. Each process begins by calling the function `readMatrixMPI()`[1], providing its rank and the total number of processes. The function then handles the reading of the matrix, reading in only the data required for the process that called it. In this way we ensure that each process has smaller blocks of data to work with, and thus can drastically reduce the number of cache misses.

The first two times we are combining a single integer held by each process into an array of integers that is then shared with all processes. To complete this task we use MPI's `Allgather`

[1] Provided by Dr. Senning

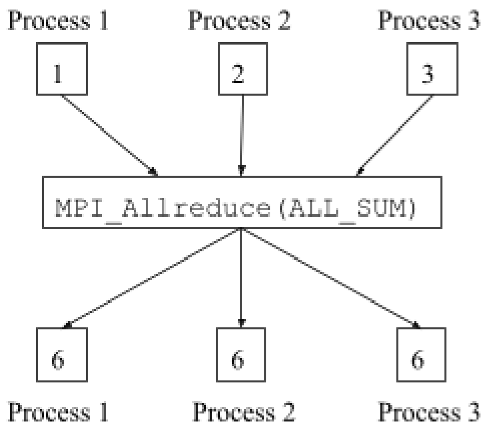
function. We set the parameters of the function such that every process will send one integer variable to every other process. After this single function call, every process will have every other process's value along with its own, and thus every process holds an identical array.



MPI's Allgather function.

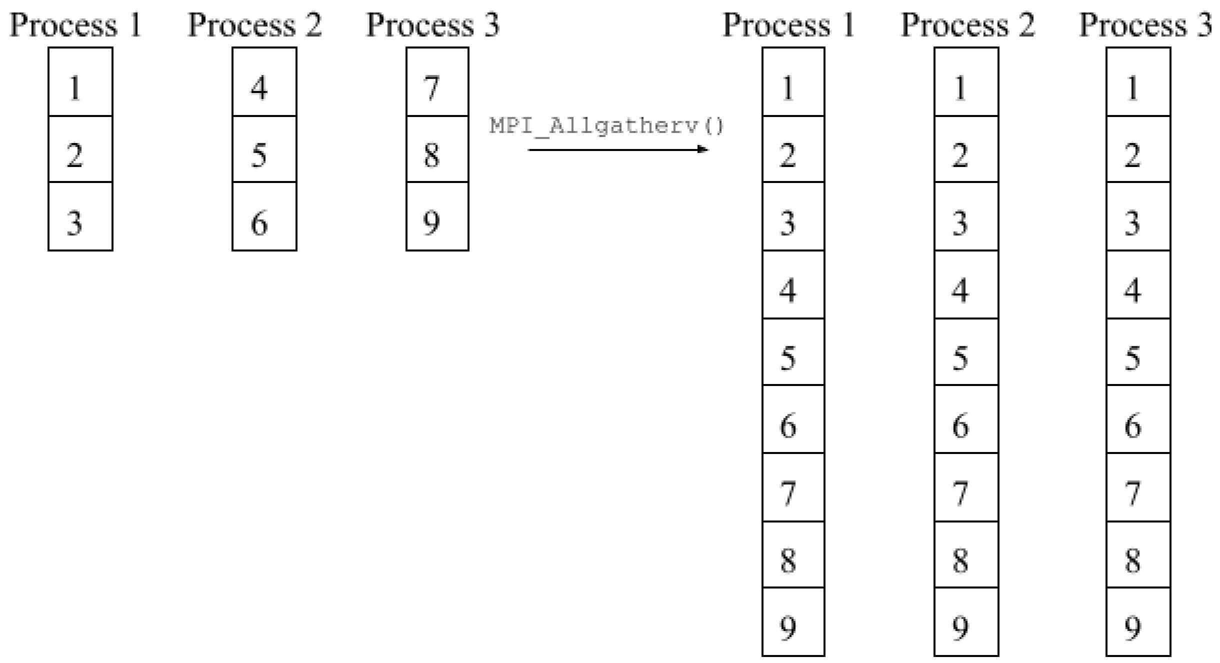
Because the order that processes call this function is non-deterministic, this function also ensures that a value it receives will be put in that process's index in the array, determined by the process's rank. We call this function to accumulate each rank's f_r and n_r into the shared arrays F and N, respectively.

We also use MPI to sum the individual α_r and λ_r values held by each process to the shared $\alpha$and λvalues. To do this, we call MPI's `Allreduce` function, using the `ALL_SUM` operation parameter. This function gathers each process's individual value to one process, summing them all together. It then broadcasts this value out to each process, so they all have a copy.



MPI's Allreduce function using ALL_SUM.

Finally, in order to update the x vector at the end of the while loop we need a way to combine the individual y_r chunks that each process holds. We can do this by using the `Allgatherv` function.



MPI's Allgatherv function.

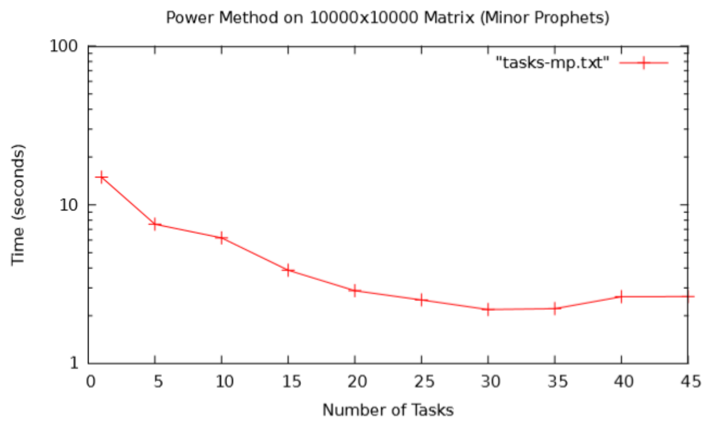
This function is essentially the same as `Allgather`, except it combines arrays into larger arrays, not elements into arrays. To use this function we input the two arrays that we created at the beginning of the algorithm, *F* and *N*. These arrays will tell MPI how long each process's arrays are, and also where they belong in terms of the *x* array.

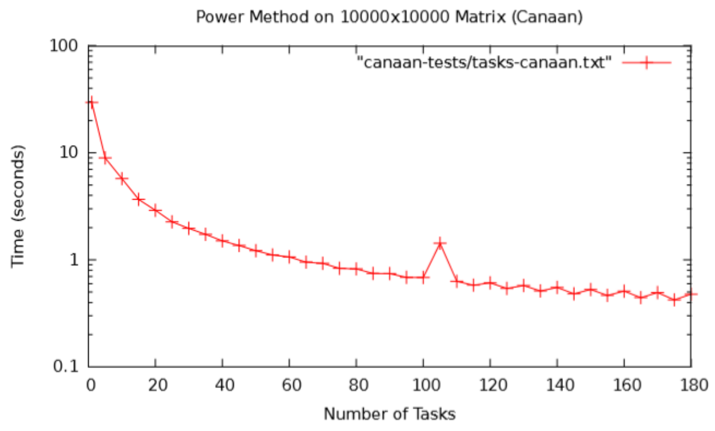
Finally, we need to ensure that we close the environment when the algorithm is finished. To do this we call `MPI_Finalize`. This step is crucial, without it the program will crash.

Discussion and Analysis of Results

The program was run on two different clusters: Minor Prophets (MP) and Canaan. The MP cluster consists of 12 workstations communicating over ethernet cables. The Canaan cluster is made up of 18 nodes (broken up into communicating with Infiniband cables. of To test the program, Slurm (Simple Linux Utility for Resource Management) was used to allocate jobs. Jobs were allocated using the `salloc` command with the `--exclusive` option to prevent multiple jobs from running at the same time on a node. This was done to ensure that the timing data was as unbiased as possible. Any time a specific configuration was run, it was run five times with the resulting computation time averaged over those five runs. Times were recorded using `MPI_Wtime`, called directly before and after the algorithm's while loop.

Three different tests were run. First, the non-omp version of the program was run on the MP and Canaan clusters with a linearly increasing number of tasks. The program was run with 1 to 45 tasks on the MP cluster and 1 to 180 tasks on the Canaan cluster. A 10000x10000 matrix was used.

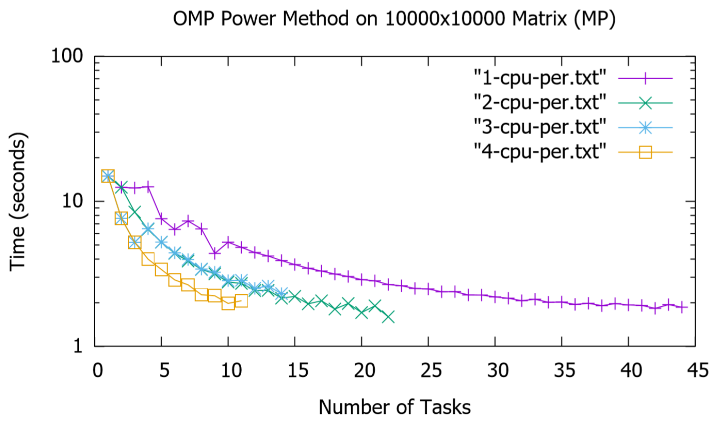


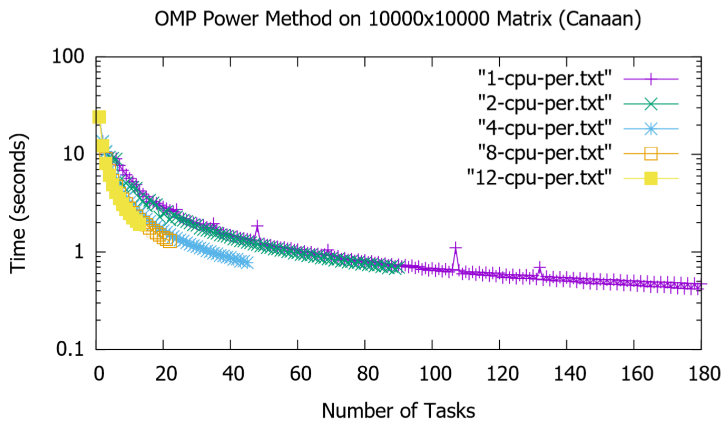


Test 1 results.

This test revealed a significant improvement over the serial version. While the Canaan version took 30 seconds with 1 task, 180 tasks returned a time of 0.477 seconds, a nearly 97% decrease. The minor prophets cluster went from 18 seconds with 1 task to 2.6 seconds with 44 tasks, a 85% decrease.

The second test revealed what would happen if multiple CPUs are assigned to each task. In the previous test, there were many tasks running at the same time, but a task's work was completed serially. Therefore, by using OMP and assigning multiple CPUs to each task, that workload can be done in parallel.

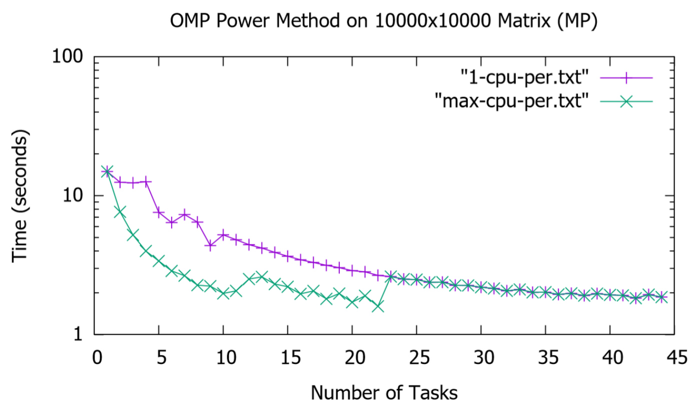


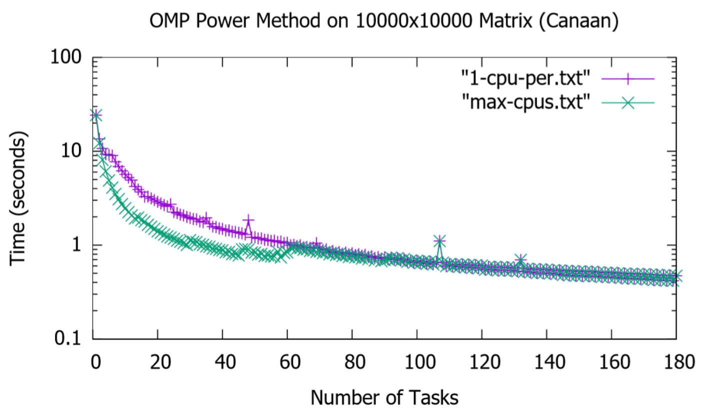


Test 2 results.

Because each task is run on 1 CPU, increasing the number of CPUs assigned to a task is only feasible when the product of CPUs and CPUs per task does not exceed the total number of CPUs in the cluster. For example, we cannot create 180 tasks and assign two processors to each task on the Canaan cluster, because there are only 180 total processors.

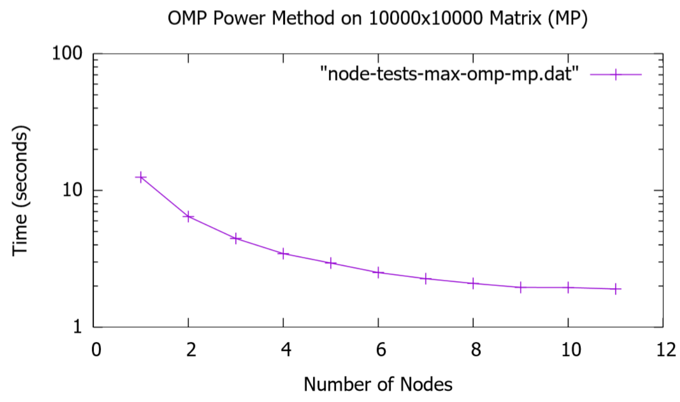
The results reveal that running with multiple CPUs assigned to each task is more efficient, this can be more clearly seen when comparing the times of a single CPU per task with the maximum number of CPUs that could be applied for that number of tasks. Notice that about halfway through both graphs both lines are the same, this is because the maximum amount of CPUs that can be assigned to each task at those higher levels is 1.

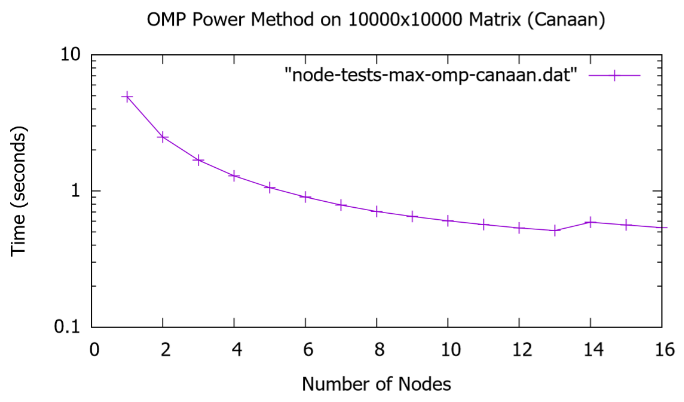




Comparing serial and the OMP versions.

The third test used nodes. In order to reduce the amount of time spent communicating data between tasks, we can group the tasks together on nodes. The tasks are ordinarily assigned to processors regardless of where those processors are physically located. Thus, in the case of the MP cluster, you could have four tasks running on four different workstations, which will require a significant amount of communication over ethernet to complete. By grouping the tasks by node, we can allocate a number of nodes (which will each have their own set of processors) and assign a number of tasks to those nodes. This theoretically can improve the speed because communication inside nodes is quicker than between nodes. Notice, however, that there is no difference between allocating the maximum amount of tasks and the maximum amount of nodes. Both will allocate every processor, so trying to group them will have no effect.





Test 3 results.

Interestingly, grouping by node seemed to have a very limited effect. It seems roughly equivalent to the results of test 1, and in fact the maximum speed for these tests is about the same. Further analysis may be required.

Conclusion

When computing the dominant eigenvalues and eigenvectors for a large matrix, MPI can be an incredibly powerful tool to gain a huge amount of performance with a (relatively) small amount of changes. This of course assumes you have access to a large amount of computing power. Utilizing OMP to further parallelize can yield more speedup, but this is only possible when there are at least half as many tasks as there are CPUs. In the future, implementing blocking techniques for the matrix-vector multiplication could yield more speedup. It may also be interesting to analyze the performance on matrices larger than 10000x10000.