PROJECT - TRADE RECONCILIATION PROCESSING THROUGH HPC
COURSE: HIGH-PERFORMANCE COMPUTING (ENGR-E517)

Abhinav Bajpai (abbajpai@iu.edu)

Indiana University – Bloomington

1. INTRODUCTION

1.1 Topic and Motivation

Topic: Trade Reconciliation Processing through HPC

In 2017, one of our bank clients was spending close to $2M annually on exception processing of unreconciled trades. Their process was time intensive and error-prone with the risk of missing potential trade matches as it could only reconcile 1 to 1 matches like $10 with -$10. The reconciliation of divisible trades, such as -$8 and -$2, with $10 was not possible in their system and had to be sent to an exception resolution team for processing.

We deployed a combinatorial algorithm-based solution for them to reconcile 1 to 1, 1 to M, M to 1, and M to M trades which partially automated their process. However, our solution worked well only for up to 250 trades, after that, the number of combinations was too large for a single processor to handle. Based on my newly acquired knowledge of parallel programming, I would like to review my previous algorithm and rework it using the parallel constructs we learned in class.

1.2 Problem Description

In the trade reconciliation process, a list of trade amounts (credits and debits) is compiled and checked to see if there are any combinations whose sum is between -1 and 1.

Example: If we have a list of 5 numbers then possible combinations can be expressed as: Combination ([list of 5 numbers], k) where k can range between one to five.

Table 1: Combinations of List of Five Numbers

K	Combination (N, k)	Total
1	Combination (5,1)	5
2	Combination (5,2)	10
3	Combination (5,3)	10
4	Combination (5,4)	5
5	Combination (5,5)	1
	Total	**31**

The above table has 31 possible combinations to check if their sum is between -1 and 1. Upon finding the right combination, all numbers in the combination are labeled as "Matched" and removed from the number list.

The total number of combinations for a given **N** number list is **2^N - 1,** and as **N** increases, the possible combinations may scale into billions, making solving this problem computationally challenging.

1.3 Data (Input/Output)

The underlying data is a list of 846 trades grouped into 155 masked trade accounts provided by the bank client. The largest trade account has 354 trades, while other trade accounts have smaller trades ranging from 2 to 42.

The algorithm will return all possible combinations of trades from each trading account whose sum is between -1 and 1.

1.4 Proposed Parallelization Strategy

Below is a high-level view of the parallelization strategy that was proposed to solve the problem:

Algorithm Design – Original Proposal

```
1. Initialize the list of N elements and P processors (MPI)
2. k = [1, 2, 3, ….,N]
3. Assign N/(P-1) workload to each processor [different values of k]
4. for each processor $P_r$ do in parallel
5. Share all elements in the trade list with the processors
6. Create OpenMP parallel region to iterate over assigned values of k
7. If the sum of the combination is between -1 and 1, send it to the master processor with the "Matched" label
8. After all iterations of k are done, return the list of numbers with "Matched" and "Unmatched" labels
```

In addition to the above algorithm design, we had proposed to explore MPI and OpenMP communication protocols to dynamically update the trade list as we find target combinations so that we can reduce the trade list and the combination search space. Given the complexity of the task, we have developed 3 separate applications using Python, OpenMP, and MPI. The algorithm design of each of the approach is discussed in detail in the following section.

1.5 Algorithm Design

For this project, we have explored the following techniques to find out which of the following performs better in terms of speed and memory utilization on the IU Carbonate HPC system.

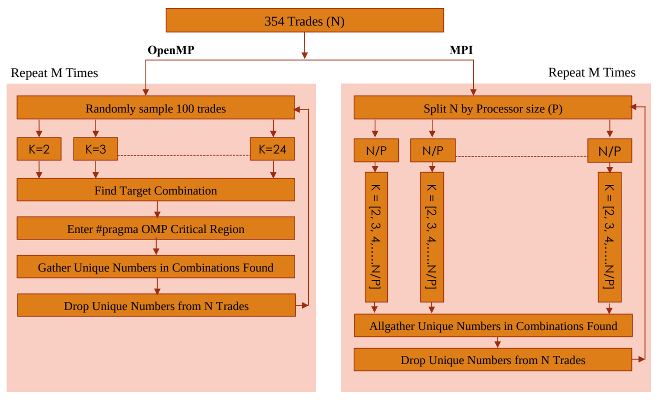
Sequential Python Programming – A brute force approach that sequentially creates combinations from k = 2 to k = N, then checks if the combination is between -1 and 1 and updates the counter for the unique and total combinations found accordingly. We compare the remaining two parallel programming approaches against this sequential program.

Algorithm Design – Sequential Python

```
1. Load the list of N (354) trades, initialize a set of unique numbers, total and unique combination counter
2. for k in range 2 to N
3. for each combination in (N, k)
4. Check if the sum of the combination is between -1 and 1
5. If True, update the total combination counter by 1
6. Check if the new combination numbers exist in the set of unique digits
7. If yes discard the combination else add the number in the set of unique combinations and update the unique combination counter
8. After all iterations are done, print the total combination and total unique combinations
```

The next two approaches follow parallel processing techniques. A high-level view of OpenMP and MPI implementation can be seen in the graph below:

Figure 1: OpenMP and MPI Approaches to Finding Target Combinations


354 Trades (N)
OpenMP
MPI
Repeat M Times
Repeat M Times
Randomly sample 100 trades
K=2
K=3
K=24
Find Target Combination
Enter #pragma OMP Critical Region
Gather Unique Numbers in Combinations Found
Drop Unique Numbers from N Trades
Split N by Processor size (P)
N/P
N/P
N/P
K = [2, 3, 4.....N/P]
K = [2, 3, 4.....N/P]
K = [2, 3, 4.....N/P]
Allgather Unique Numbers in Combinations Found
Drop Unique Numbers from N Trades

OpenMP Parallel Programming – With this programming construct, we convert a sequential Python program into a parallel one that uses multiple threads to benefit from parallel processing. The Python program searched for k sequentially, but in an OpenMP construct, multiple threads are launched so that different values of k [1, 2, 3, etc.,] are searched in parallel, with the expectation that it will save processing time.

Algorithm Design – OpenMP

```
1. Load the vector of N (354) trades as Amount, and initialize the total combination counter and unique value vector [to save unique numbers found in combinations]
2. Create an omp parallel region to launch multiple threads (#pragma omp parallel)
3. Create parallel for nowait
4. for k in range 2 to 24 do in parallel [can be extended up to N]
5. for J in range 1 to M [where M is a hyperparameter that can be tuned for combination search]
6. Randomly shuffle the trade list and extract 100 elements
7. Generate combinations of (100, k)
8. Check if the sum of the combination is between -1 and 1
9. If True, update the total combination counter by 1
10. Create a #pragma critical region
11. Check if the numbers in the new combination found are in the unique value vector
12. if yes, discard the combination
13. if no, insert the numbers of the combination in the unique value vector and drop the number from the Amount vector
14. Print the new size of the Amount vector and the unique combination found
15. Exit #pragma critical region
16. Repeat steps 6 – 15 M times
17. Print the total combination found by each thread
```

Each thread shares 354 trades, but we sample only 100 trades in each iteration to manage the number of combinations generated by the algorithm. With just k = 5, there are approximately 45 billion combinations possible, and this number increases as k increases. Due to this large number of combinations, we sample from the file randomly and let each thread handle a smaller set of numbers to generate combinations. In theory, if we shuffle and sample from the list enough times (Step 5: M is very large), we should be able to catch every possible target combination for k.

For the same hyperparameter setting (M is not very large), we may or may not get the same number of combinations given random shuffling and sampling, which is a limitation of this approach. We evaluate the performance of this algorithm in the next section.

MPI Parallel Programming – Considering the large number of combinations that get generated by 354 trades under consideration, we decided to experiment with the MPI construct as well to take advantage of distributed memory. We don't take a brute-force approach in the MPI construct, instead, we divide the 354 trades within each rank (processor) and then make each rank search for valid combinations (sum between -1 and 1), thereby reducing the workload and possible combinations to generate and search. Once each rank finds unique combinations, we collect the numbers from all ranks and drop them from the original trades list. We randomly shuffle and redistribute the trades among the ranks for finding new combinations. This process is repeated multiple times to find as many combinations as possible.

This approach is based on the idea of Monte-Carlo method, which involves a large number of iterations in order to determine a stable price for options. Likewise, we believe that dividing the trades into smaller chunks will reduce the memory burden and allow deeper searches ($k > 5$). By continuously shuffling chunks, we create an opportunity to find new unique combinations faster, thereby shrinking the trade list for the next iteration.

Algorithm Design – MPI

```
1. Load the vector of N (354) trades as Amount and P processors (MPI)
2. Initialize Iterator J = 1 to M [where M is a hyperparameter that can be tuned for combination search]
3. for J in range 1 to M
4. Assign N/P workload to each processor [different trades to each processor]
5. Initialize unique number vector, total combination counter, and unique combination counter
6. for each processor $P_r$ do in parallel
7. for each combination in (N/P, k)
8. Check if the sum of the combination is between -1 and 1
9. If True, update the total combination counter by 1
10. Check if the new combination number exists in the vector of unique number
11. If yes discard the combination else add the number in the set of unique combinations and update the unique combination counter
12. Print the unique combination found
13. After steps 6 -12 are done by all processors, use the MPI Allgather operation to collect all numbers in each unique combination found by all processors
14. Use MPI Reduce to find the total of all combinations found by each processor
15. Drop these unique numbers from the Amount vector, it will reduce N
16. Randomly shuffle the new vector of trades and move back to step 3 until all iterations of J are complete
17. Print the total combinations found
```

The above algorithm was designed to overcome the challenge of generating a very large number of combinations by reducing the trade list at each iteration and finding the maximum possible combination in a limited run-time environment. However, one major limitation of this algorithm is that it searches for k = 2 to N/P instead of K =2 to N. We expect to overcome this limitation in future work by extending the algorithm to search for the remaining list of N once it has been reduced to a much smaller size that can be managed by brute-force search.

1.6 Performance Testing Results

We selected IU's **Carbonate HPC system** to implement all the performance testing. Specifically, we used the large memory cluster with 24 processors and 503GB memory allocation as this was the highest memory allocation available on this system. The IU Carbonate system has the GNU Compiler Collection (GCC), along with OpenMP and MPICH compilers for compiling parallel programs. We followed the specific directions to compile the C++ program provided on the IU knowledge base page and used the below mentioned commands to compile the mpi and omp code:

- mpicxx mpi.cpp -o mpi.exe
- g++ omp.cpp -fopenmp -o omp.exe

In order to find the most efficient implementation, we set a time constraint of 120 minutes for each algorithm runtime. In the following table, we provide the number of distinct combinations and the remaining numbers (unresolved trades) for each implementation. During parallel programming implementations, we also take into account the remaining numbers, since parallel implementations are capable of diving deeper ($k > 5$), thus reducing the number of unique combinations while still offsetting more numbers.

Table 2: HPC Implementation

HPC Implementations	Unique Combinations	Numbers Remaining	Runtime (mins)
Python – **Baseline**[1]	48	154	120
OpenMP (8 threads)*	29	224	120
MPI (16 processors)*	33	128	45

From the table above, MPI implementation performed the most efficiently, resolving 226 trades out of 354 trades in 45 minutes (completed after 10,000 iterations), leaving only 128 unresolved trades (354 - 226 =128). Sequential Python implementation performed the second best with 154 unresolved trades, while OMP implementation did the worst with 224 unresolved trades.

The MPI implementation found only 33 unique combinations but it went into greater depths of K so it was

[1] Itertools Combination

able to resolve more trades. The below table showcases the distribution of the combinations by K:

Table 3: MPI Implementation - Combination breakup by K

K	Number of Combinations	Total Numbers
2	1	2
3	5	15
4	1	4
5	4	20
6	6	36
7	3	21
8	4	32
9	2	18
10	2	20
11	2	22
12	3	36
Total	**33**	**226**

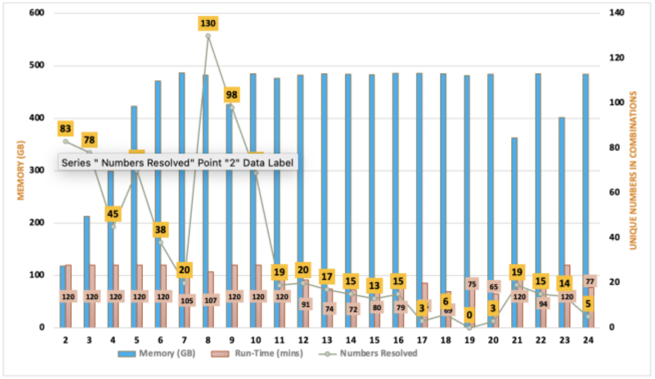
In comparison, both OpenMP and sequential Python code did not go beyond k = 5 search space in the allocated 120-minute runtime. A more detailed break-up of each implementation is provided in the Excel file "Implementation Results.xlsx"

1.7 Performance Evaluation

In our MPI and OpenMP implementations, we performed dynamic master list updates and found that the MPI implementation performed better than the OpenMP implementation. Strong scaling is performed for both implementations and following are the observations for each algorithm:

OpenMP – The OpenMP algorithm was executed over threads 2 to 24 and the below chart provides the total runtime, memory usage (GB), and count of trades resolved.

Figure 2: OpenMP Strong Scaling



From the charts, we can conclude that the algorithm did not scale very well. Initial jobs with 2 to 6 threads completed the allocated 120-minute runtime with low memory usage, but as we increase the number of threads, we observe heavy memory usage with jobs continuously touching 480 GB and failing due to memory errors. It is the inherent design of our algorithm, in which we attempt to search for target combinations for different values of k on a sample size of 100, that leads to failed jobs. The number of combinations for search and load imbalance increases as the thread count increases. With eight threads, the algorithm looks at +17 billion possible combinations, and it continues to grow as the number of threads increases. This is the primary reason for high memory usage and memory out errors. By reducing the sample size and increasing the repetition iteration parameter (M), we can experiment to improve this performance.

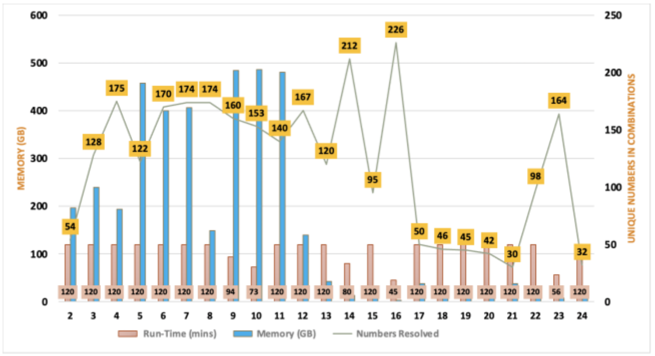
In addition, the overhead generated by each thread as it enters the critical region to drop the unique numbers from the shared "Amount" vector also impacts the performance of the OpenMP implementation. After 11 threads, we observe a sharp drop in the number of distinct numbers/target combinations due to this overhead. Our implementation uses the omp critical construct, so we can experiment with the omp atomic construct to see if it performs better.

MPI – In comparison with the OpenMP implementation, MPI performed better under strong scaling. Over two to 24 processors, we executed the MPI algorithm and observed that the number of trades resolved increased from 54 (2 processors) to 226 until 16 processors, and then declined to 32 with 24 processors. By dividing the workload of N trades by processor count, the search space shrinks from 177 per processor (2 processors) to 15 per processor (24 processors). In contrast to OpenMP with its constant sample size of 100, this division of workload reduces memory usage.

In the case of 16+ processors, performance declines because the sample size is too small (less than 20 per processor), so finding the ideal combination for each processor becomes less likely. Smaller sample sizes would require a lot more than 10,000 iterations to show any convergence. To test this hypothesis, we ran over 100k iterations for 12 hours on 24 processors and observed 48 unique combinations resolving 232 trades which is 6 more than our best MPI implementation using 16 processors (results available in Excel file). The result supports our initial theory that given enough iterations and runtime, we can find all possible combinations through this approach.

MPI implementation is also impacted by the implicit barrier encountered when calling MPI_Allgather and MPI_Reduce for collective communication. Synchronization costs are incurred when unique numbers are collected by processors and then dropped from the "Amount" vector before the next cycle.

Figure 3: Strong Scaling – MPI



A moderate sample size and a reasonable number of iterations (greater than 10k) are recommended for MPI implementation due to the above strong scale performance. Optimal performance can be achieved by tuning both of these parameters when given a file of size N.

Python – We executed the Python implementation with the sorted list of trade numbers to test if sorting can increase the likelihood of identifying target combinations.

The sorted list improved the performance of sequential python implementation. We saw the count of resolved numbers increase from 200 to 225 which is a 12.5% improvement in performance over unsorted input.

The results are provided in the table below:

Table 4: Sorted vs. Unsorted Input

HPC Implementations	Unique Combinations	Remaining Numbers	Runtime (mins)
Python – Unsorted Input	48	154	120
Python – Sorted Input	53	129	120

While the sorted input result is almost as good as the MPI output, it is less efficient as it resolves 225 trades in 120 minutes. In contrast, MPI resolves 226 trades in 45 minutes. Understanding Itertools' underlying combination algorithm will help us explain the difference in results for sorted and unsorted input to Python sequential implementation.

Both OpenMP and MPI algorithms perform random shuffling of the trade list hence we did not perform the sorted input test on these implementations.

The implementation code and Excel with a summary of results are available on **IU GitHub** - abbajpai/HPC_Project at main · engr-hpc-fa22/abbajpai (iu.edu)

1.8 Future Work

Considering our limited knowledge of C++ programming language, both OpenMP and MPI applications will need to be thoroughly tested and their hyperparameters tuned. On the IU Carbonate system, profiling libraries like PAPI were not available, so we were unable to profile the jobs and analyze cache usage and function calls more deeply. It is our intent to continue these aspects of the project in the future and to improve upon our implementations.

1.9 Acknowledgment

This work aimed to solve a real-world problem encountered in the past. According to the references provided in the next section, we have not found a similar approach to our MPI or OpenMP implementation. In our code, we use a recursion function to generate combinations because C++ libraries do not provide a built-in function for it. The code has relevant references for all custom functions used by us.

1.10 References

- Generating all K-combinations - Algorithms for Competitive Programming (cp-algorithms.com)
- math - Optimize C++ function to generate combinations - Stack Overflow
- Combination Algorithm: Print all Possible Combinations of R (guru99.com)
- Key to Parallel Combinations: Enumeration (vlkan.com)
- Subset sum problem - Wikipedia
- Subset Sum Problem - Dynamic Programming Solution | Techie Delight
- Subset Sum Problem (Recursion) - YouTube
- np complete - How to reduce SUBSET-SUM with integers to SUBSET-SUM with non-negative integers? - Computer Science Stack Exchange
- Rearrange positive and negative numbers in O(n) time and O(1) extra space - GeeksforGeeks
- Subset Sum Problem using Backtracking – Pencil Programmer
- d99kris/rapidcsv: C++ CSV parser library (github.com)