Parallel and Distributed Computing - Recommender System

1. Introduction

The purpose of this project is to simulate a recommender system, computing results across a network of computers using the OpenMPI Message Passing Interface implementation.

In section 2, we start by describing our approach to applying Foster's Methodology to the problem. In section 3, we touch on communication channels and load balancing concerns. In section 4, we go over the OpenMPI implementation and, in section 5, we try to analyze its scalability. Finally, we test this version and compare empirical results to the previous serial implementation.

2. Foster's Methodology Approach

2.1 Partitioning

We considered each primitive task in this problem to be the computations necessary for each non-zero entry of domain matrix A. This partition model defines a number of tasks at least an order of magnitude higher than there are target processes, avoids redundant computations and also reduces redundant data storage to a solid number - as, in turn, it allows us to segment matrices L and R in rows and columns respectively, one of each necessary for said non-zero entries.

2.2 Communication

When distributing the initial values the communication is simple: each node receives their respective chunk of non-zero entries and their initial L and R matrices. During the factorization, given that each row of L and column of R in their entirety are updated by their respective non-zero entries, local communication in our approach consists of reducing these matrices using their respective row-wise and column-wise communicators in order update them in preparation for the next iteration. When generating the output, the communication consists of reducing each local output and printing it out.

2.3 Agglomeration & Mapping

We decided to follow a **checkerboard decomposition** of matrix A. What this implies is that we group blocks of non-zero entries in a grid-like fashion and give one block to each process. This helps decrease communication costs and guarantees that the number of tasks can still scale with the problem size.

3. Overview

3.1. Communicators

Three communicators were created to handle all communication throughout execution:

- *Cartesian communicator* - Establishes each process in the grid, creating a virtual topology which allows for communication along grid dimensions and facilitates the creation of the following communicators;
- *Row & Column communicators* - Communicators used when the only necessary communication is between processes in the same grid row/column respectively (e.g. reducing the chunks of L and R during the matrix factorization section).

3.2. Load Balancing

On most instances, and assuming non-zero entries in matrix A are somewhat evenly distributed, the workload is uniformly divided among processes. Unfortunately, there are a couple of instances that, when ran with certain numbers of processes, leave some of these work-free. To improve load balancing even further, a balanced grid creation algorithm was developed, mentioned further ahead.

3.3. Memory Management

Given the chosen decomposition, we need to send chunks of matrices L and R to every process. Additionally, during the matrix factorization part of the algorithm, we need a copy of each of these as well. To minimize memory usage when distributing matrix A, we send one small chunk at a time while still complying with the idea that each grid entry must have a disjoint set of non-zero entries.

3.4. Balanced Grid Creation

Sometimes the default grid is not good enough in regards to performance or memory usage, so we came up with an algorithm that tries to balance the grid dimensions by taking into consideration the relation between the number of lines and number of columns of matrix A.

4. MPI Implementation

4.1. Grid Creation

Let P be the number of processors, F be the number of features, i be the number of items and u be the number of users. We compute the following ratio $R = min(P, max(i,u)/min(i,u))$.

The balanced grid is created taking the default grid $rows \times columns$ returned by `MPI_Dims_create` as the starting point and modifying its dimensions so that $rows = min(f(rows, columns, P, R))$ and $columns = max(g(rows, columns, P, R))$ where:

$$f(r,c,P,R) = |R - r|, 0 < r \leq R \wedge r \times c = P \quad (1)$$

$$g(r,c,P,R) = |R - c|, 0 < c \leq R \wedge r \times c = P \quad (2)$$

In the end, if $items > users$ then we rotate the resulting grid, i.e. $rows \times columns$ becomes $columns \times rows$

Example: $P = 16$, $i = 1600$, $u = 160$, $R = 10$. Default grid is 4×4. L size is $400 \times F$ and R size is $40 \times F$. This means every process has to store and reduce a matrix L that is much bigger than its counterpart R. The resulting grid is 8×2 which means the sizes of L and R become more even, reducing the impact of storing and reducing matrices with unbalanced sizes.

4.2. Distribution

We assumed the input file is accessible by a single process. On account of this, we start by broadcasting the input metadata (alpha plus the number of features and total users/items/non-zero entries) followed by reading non-zero entries by rows, iteratively computing each grid column frontier and sending each segment of non-zero entries to the respective process. L (R) is also sent row (column) by row (column), which means we only allocate enough memory for an entire row (column).

4.3. Matrix Factorization

For every iteration, processes start by either copying the previously computed chunk of the stable L/R matrix, if they are a root process for a row/column of the grid respectively, or zeroing the unstable matrices. After going through their own set of non-zero entries, processes converge by reducing their chunks of L/R into the respective stable versions of the matrices.

4.4. Output

Each process computes the maximum for every row they have been allocated with regards to the columns they store. Once that is done, they again converge by reducing the arrays of maximums into the output array of the respective row root process, obtaining the maximums for each chunk of rows. Afterwards, these are gathered in the main root process, which then outputs the values.

4.5. Partial Line Reduction

We implemented a different version of the matrix factorization method where we only reduced the rows of L and columns of R that had effectively been modified by each grid row and column. Ultimately, this was scrapped because in most instances and real-world examples, this hardly makes any difference.

4.6. Hybrid

We also implemented a hybrid approach which is very similar to our previous delivery but this time we decided to manually load balance the non-zero array by giving each thread a disjoint set of non-zero values. This improvement removes the need for synchronization, except for barriers and the reduction of L or R at the end of each iteration. This version provides a tangible upgrade in performance compared to one that employs **atomic** keywords, however, during evaluation the results turned out to be hit or miss compared to the raw OpenMPI implementation.

5. Evaluation

In the Appendix, we provide graphs and tables with speedups and execution times when running both raw and hybrid versions locally and on the cluster.

5.1. Speedup

Overall, the speedup was mostly good and almost perfect in some cases when the program was tested locally.

5.2. To Cluster, or not to Cluster

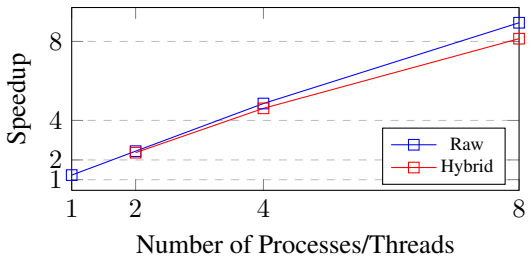
Unfortunately, when testing on the cluster the story is completely different. Results were all over the place and sometimes an instance which took a set amount of time to complete would, out of nowhere, take double or triple that time to run again. These slowdowns mostly manifested whenever there was some communication between different machines. It was a very frustrating and unfair experience. We believe this was some kind of cluster instability because we think our implementation is of good quality.

6. Conclusion

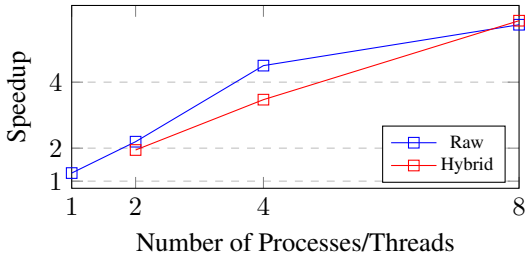
It was a good experience to learn to how use OpenMPI. It was also interesting to manage memory in a more conscious way to be able to run the biggest instances without exceeding capacity. The experience with the cluster could have been better.

Appendix

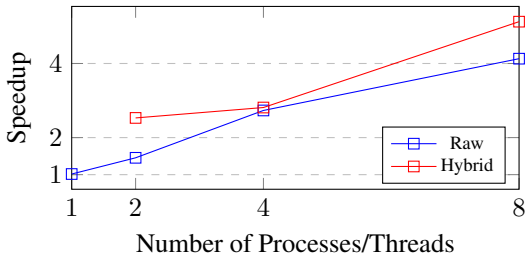
inst500-500-20-2-100



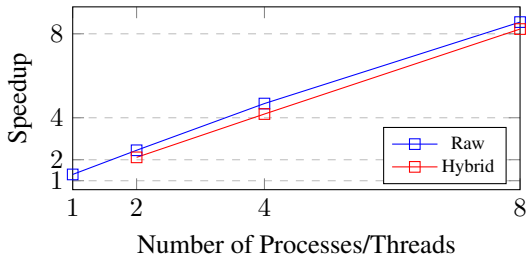
inst50000-5000-100-2-5



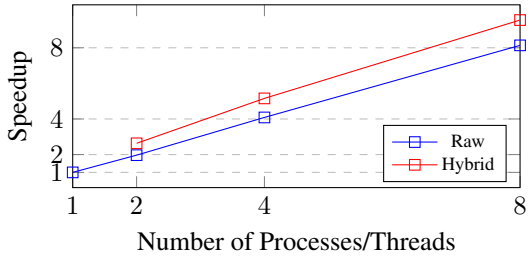
instML100k



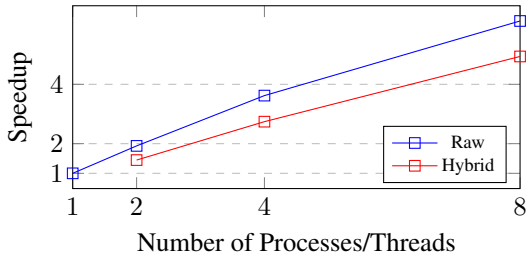
instML1M



inst1000-80000-20-10-1000



inst60000-2000-200-10-20



Instance	Exec. Time (s)							
	Serial	MPI-1	MPI-2	MPI-4	MPI-8	MPI-16	MPI-32	MPI-64
inst1000-1000-100-2-30	18.123	17.52	9.42	5.19	13.98	11.11	13.14	9.31
inst200-10000-50-100-300	24.711	28.49	14.08	6.92	5.04	3.94	4.08	31.74
inst400-50000-30-200-50	35.813	38.94	20.18	10.23	5.88	3.86	3.38	14.76
inst50000-5000-100-2-5	156.984	157.56	92.05	48.59	88.73	89.45	38.39	114.38
inst500-500-20-2-100	57.798	55.67	28.39	14.75	21.21	20.63	27.77	70.42
inst600-10000-10-40-400	83.490	84.36	42.19	22.12	21.78	31.91	161.74	42.02
instML100k	104.930	97.19	84.17	42.51	35.59	23.85	75.93	66.10
instML1M	125.201	103.68	51.77	31.83	20.57	12.25	8.60	15.69
inst60000-2000-200-10-20	-	-	-	-	-	27.57	16.83	11.05
inst20000-10000-40-2-50	-	-	-	-	-	56.07	74.11	169.83
inst1000-80000-20-10-1000	-	-	-	-	-	20.86	19.04	22.77
inst1000-1e6-1000-1-3	-	-	-	-	-	211.80	174.89	143.60
inst1e6-100-700-1-3	-	-	-	-	-	87.42	86.32	70.34

Table 1: Execution times for the serial and OpenMPI implementations when ran on the RNL cluster

Instance	Exec Time (s)			
	MPI-1	MPI-2	MPI-4	MPI-8
inst50000-5000-100-2-5	126.09	71.50	34.88	27.35
inst500-500-20-2-100	46.90	23.65	11.90	6.46
instML100k	103.16	72.13	38.40	25.41
instML1M	96.31	51.08	26.77	14.63
inst1000-80000-20-10-1000	321.61	163.45	78.71	39.53
inst60000-2000-200-10-20	232.13	120.62	64.14	37.84

Table 2: Execution times for the OpenMPI implementation when ran locally

Instance	Exec Time (s)		
	MPI-1	MPI-2	MPI-4
inst50000-5000-100-2-5	80.76	45.25	26.78
inst500-500-20-2-100	24.40	12.53	7.10
instML100k	41.46	37.32	20.46
instML1M	59.23	29.97	15.22
inst1000-80000-20-10-1000	122.15	62.37	33.65
inst60000-2000-200-10-20	160.45	84.68	47.00

Table 3: Execution times for the hybrid implementation when ran locally with **2 threads per process**