Decentralized Parallel Inverted Index Construction with MPI

Ziming Yang

Background

Inverted Index:

- Key data structure for search engines, mapping terms to containing documents

  Example structure:

- `"computer" → [doc1, doc3, doc5]`
- `"science" → [doc1, doc7, doc9]`

- Essential for fast document retrieval and query processing

Challenges:

- Processing large text collections (Wikipedia XML Dump)
- Single machine approach too slow for real-time requirements
- Traditional master-worker architectures create bottlenecks

Project Goal:

- Implement a decentralized parallel inverted index system using MPI
- Distribute workload evenly without central coordination
- Verify Amdahl's and Gustafson's laws through performance analysis

```
undivided:(25.txt,1)
muffled:(15.txt,2) (18.txt,2) (20.tx
muffled:(10.txt,1) (16.txt,3) (19.tx
timbuctoo:(19.txt,1)
jutted:(15.txt,1) (18.txt,1)
pointing:(1.txt,7) (15.txt,8) (18.tx
derisive:(6.txt,1)
brightest:(10.txt,1) (19.txt,1) (7.t
mashed:(1.txt,3) (15.txt,2)
beam:(10.txt,1) (19.txt,3)
allergic:(15.txt,1)
sawhorses:(1.txt,1) (15.txt,2)
sawhorses:(13.txt,1)
tenets:(16.txt,4) (21.txt,1)
tenets:(11.txt,1)
pages:(1.txt,5) (12.txt,9) (15.txt,6
geological:(15.txt,1)
conservation:(14.txt,1)
burgen:(10.txt,1)
confusion:(16.txt,2)
confusion:(14.txt,2) (2.txt,4) (22.t
conviction:(19.txt,1)
conviction:(11.txt,6) (14.txt,2) (2.
expiation:(7.txt,1)
witte:(19.txt,1)
twa:(10.txt,1)
connective:(5.txt,2)
late:(11.txt,3) (14.txt,1) (17.txt,2
misfigured:(15.txt,1)
piled:(1.txt,1) (15.txt,1) (18.txt,1
wailing:(1.txt,1) (15.txt,1) (18.txt
wailing:(10.txt,2) (13.txt,2)
piled:(10.txt,2) (13.txt,2)
rethink:(17.txt,1)
complaints:(14.txt,1) (17.txt,1) (2.
virgin:(15.txt,1) (6.txt,1)
indictment:(3.txt,1)
indictment:(5.txt,1)
realm:(25.txt,1) (8.txt,11)
distinction:(6.txt,1)
distinction:(19.txt,1) (24.txt,1)
```

Time Complexity Analysis

1. Sequential Algorithm: O(N)

   N = total number of term occurrences

2. Parallel Algorithm: O(N/P + C)

   N = total number of term occurrences

   P = number of processes

   C = communication overhead, approximately O(P log P)

3. Expected Speedup:

   According to Amdahl's Law: S(P) = 1 / (s + (1-s)/P)

   s = sequential fraction (communication + I/O)

   (1-s) = parallelizable fraction (document processing)

Implementation

Data Preprocessing:

1. **Parse Wikipedia XML Dump into plain text documents**
2. **Split data equally among MPI processes**

MapReduce-Based Approach:

1. **Map phase: Process individual documents locally**
2. **Reduce phase: Distribute terms based on hash function**

Communication Pattern:

1. **Initial distribution: Process 0 assigns files to workers**
2. **Independent processing: No communication during map phase**
3. **Hash-based assignment: Each process handles specific term ranges**
4. **Global merge: Using MPI_Alltoallv collective communication**

MapReduce Pattern for Inverted Index Construction

Map Phase:Each process independently processes its assigned documents, extracting terms and building a local index.

Reduce Phase:Terms are redistributed based on their hash value, with each process handling specific term subsets.

Using MPI_Alltoallv for efficient all-to-all communication.

After the presentation, I will make my code public on GitHub

```
// MPI_Alltoallv - Core Communication Pattern
// ================================================

// Step 1: Exchange data sizes
MPI_Alltoall(send_counts, 1, MPI_INT,
             recv_counts, 1, MPI_INT, MPI_COMM_WORLD);

// Step 2: Calculate displacements
for (int i = 1; i < size; i++) {
    send_displs[i] = send_displs[i-1] + send_counts[i-1];
    recv_displs[i] = recv_displs[i-1] + recv_counts[i-1];
}

// Step 3: All-to-all data exchange (KEY OPERATION)
MPI_Alltoallv(send_buffer, send_counts, send_displs, MPI_BYTE,
              recv_buffer, recv_counts, recv_displs, MPI_BYTE,
              MPI_COMM_WORLD);

// Replaces O(P^2) point-to-point with O(P log P) collective
```

Results

Single Run Performance Summary

====Processors: 32

- Total Documents Processed: 800
- Total Execution Time: 1.45 seconds
- Mapping Phase: 0.26 seconds (17.9%)
- Reduce Phase: 1.17 seconds (80.7%)

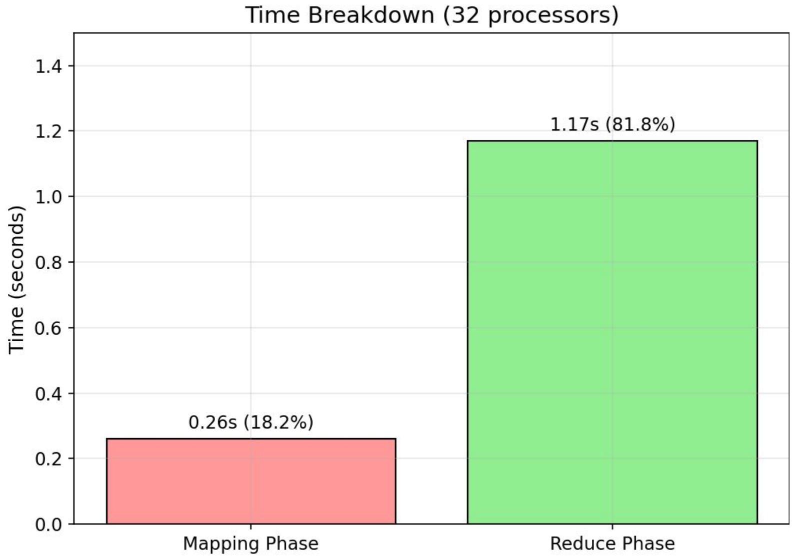
Performance for Different Processor Counts:

- 1 processors: 226.40s (Speedup: 1.00x)
- 2 processors: 108.22s (Speedup: 2.09x)
- 4 processors: 35.30s (Speedup: 6.41x)
- 8 processors: 13.69s (Speedup: 16.54x)
- 16 processors: 5.00s (Speedup: 45.28x)
- 32 processors: 1.45s (Speedup: 156.14x)

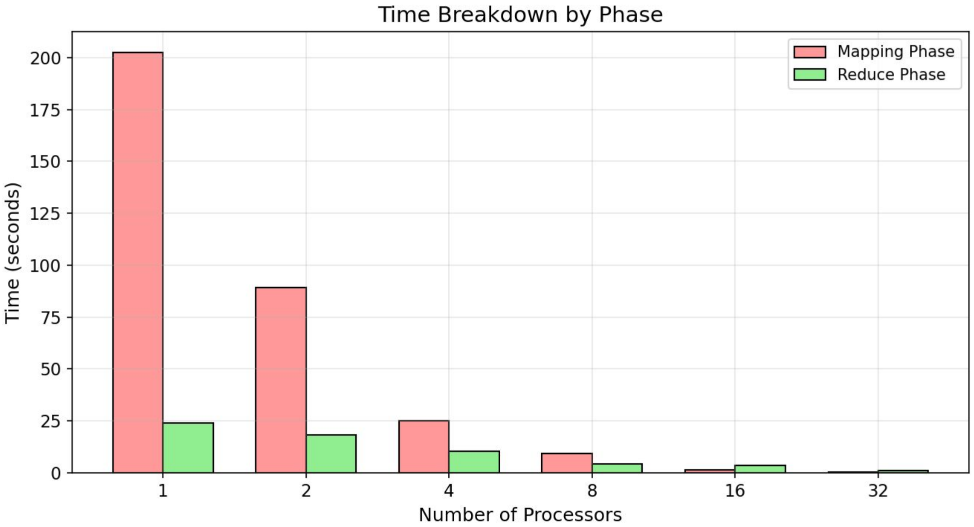
Performance for Different Processor Counts

Processors	1	2	4	8	16	32
Time (s)	226.40	108.22	35.30	13.69	5.00	1.45
Speedup	1.00x	2.09x	6.41x	16.54x	45.28x	156.14x

Results



Time Breakdown (32 processors)
Time (seconds)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1.17s (81.8%)
0.26s (18.2%)
Mapping Phase
Reduce Phase

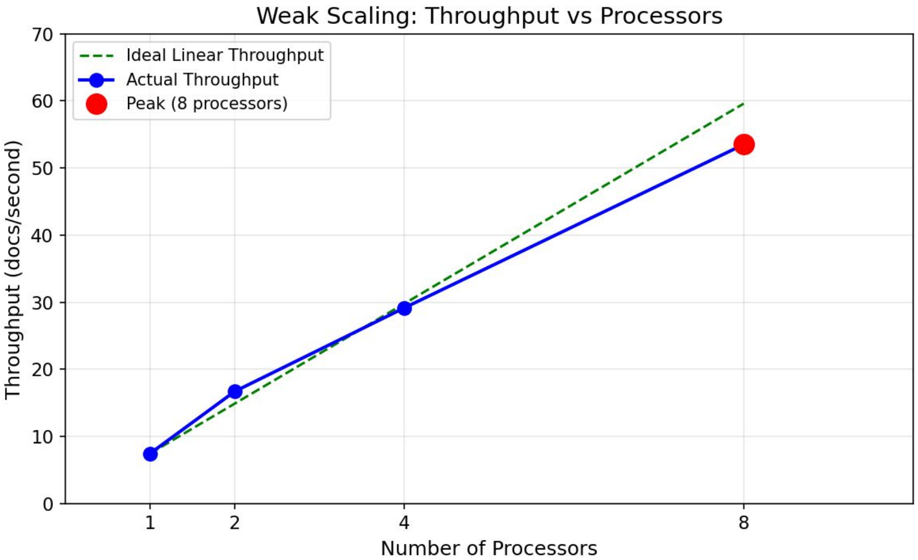


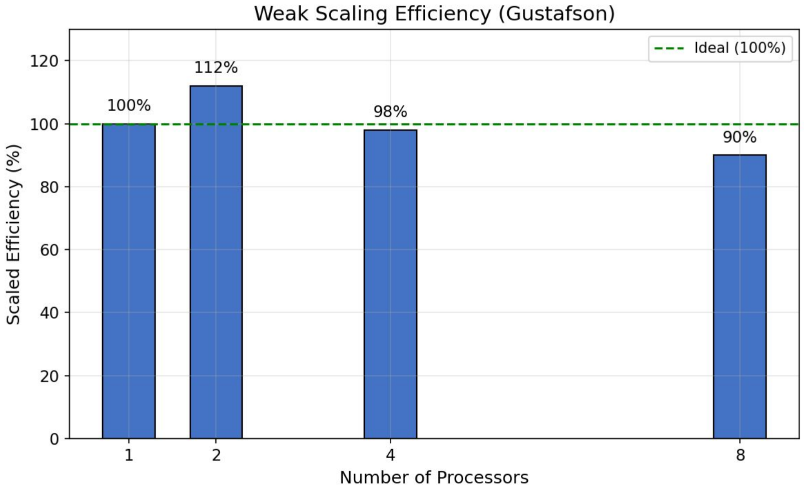
Time Breakdown by Phase
Mapping Phase
Reduce Phase
Time (seconds)
200
175
150
125
100
75
50
25
0
1
2
4
8
16
32
Number of Processors

Results - Weak Scaling (Gustafson's Law)

Fixed workload per process: 100 documents

Total work scales with processor count





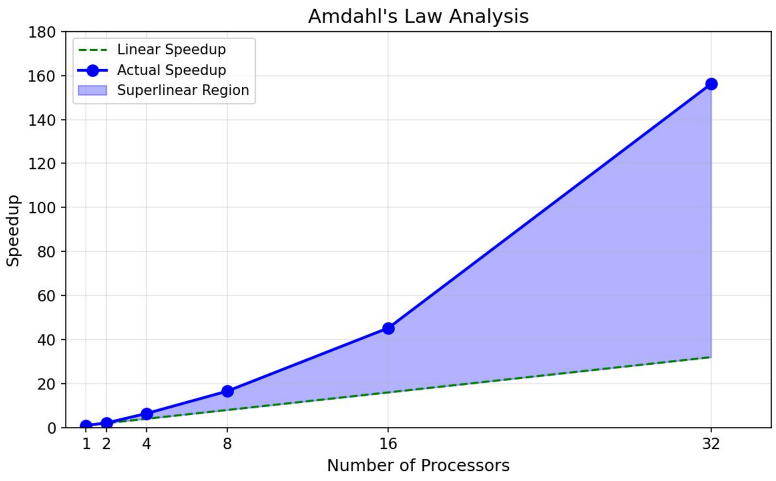
Processors:	1	2	4	8
Total Docs:	100	200	400	800
Time (s):	13.43	11.97	13.71	14.95
Throughput:	7.45	16.71	29.17	53.51 docs/s
Efficiency:	100%	112%	98%	90%

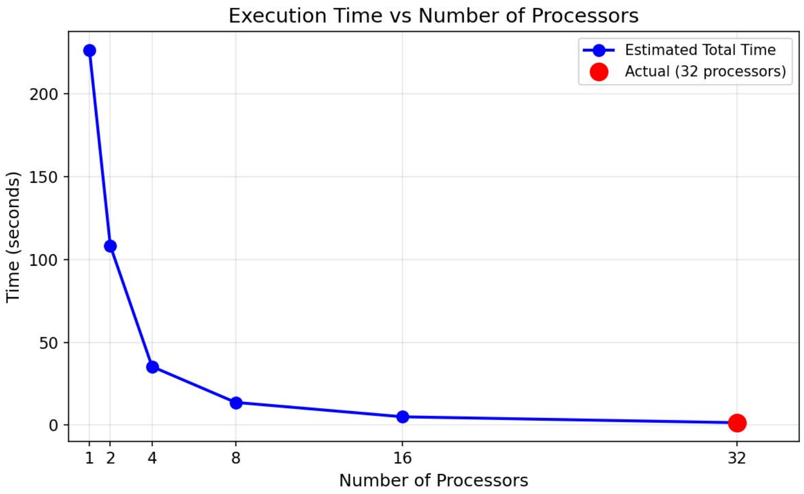
Results - Strong Scaling Analysis

Superlinear speedup observed (efficiency > 100%)

156x speedup achieved with 32 processors

Execution time reduced from 226s to 1.45s





Why Superlinear speedup

The main misunderstanding issue may be Cache Effects (Primary Cause). When running with a single process, the working set (local ndex, hash tables, document buffers) exceeds CPU cache capacity. This forces frequent main memory accesses. With multiple processes, each process handles a smaller subset of data (800/P documents), allowing the working set to fit entirely within L1/L2/L3 cache. This dramatically reduces memory access latency.I think the main misunderstanding issue may be Cache Effects (Primary Cause). When running with a single process, the working set (local index, hash tables, document buffers) exceeds CPU cache capacity. This forces frequent main memory accesses. With multiple processes, each process handles a smaller subset of data (800/P documents), allowing the working set to fit entirely within L1/L2/L3 cache. This dramatically reduces memory access latency.

Evidence: The Map phase shows 778x speedup (from 202.45s to 0.26s) with 32 processors-far exceeding the theoretical 32x linear speedup. Since Map phase involves only local computation with no communication, thisl acceleration can only be attributed to improved cache utilization.Evidence: The Map phase shows 778x speedup (from 202.45s to 0.26s) with 32 processors-far exceeding the theoretical 32x linear speedup. Since Map phase involves only local computation with no communication, this acceleration can only be attributed to improved cache utilization.

Comments?