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 $O(P)$
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:

- Parse Wikipedia XML Dump into plain text documents
- Split data equally among MPI processes

MapReduce-Based Approach:

- Map phase: Process individual documents locally
- Reduce phase: Distribute terms based on first letter

Communication Pattern

- Initial distribution: Process 0 assigns files to workers
- Independent processing: No communication during map phase
- Alphabet-based assignment: Each process handles specific letters
- Final merge: Process 1 combines all partial indices

MapReduce Pattern for Inverted Index Construction

1.1 Map Phase

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

```python
        word_count = 0
        for word in words:
            if word != '':
                word = word.lower()
                word_filename = f"word_{abs(hash(word)) % 100000}.txt"
                outFilePath = os.path.join(tempProcessPath, word_filename)
                word_index_path = os.path.join(tempProcessPath, "word_index.txt")
                with open(word_index_path, 'a') as index_file:
                    index_file.write(f"{word_filename}:{word}\n")

                with open(outFilePath, 'a') as outFile:
                    outFile.write(f'{data} ')
                word_count += 1
```

```python
if word and word[0] in data:
    try:
        with open(os.path.join(root, filename), 'r') as f:
            file_content = f.read()
            words = file_content.split()
            words.sort()

            result = {}
            for doc in words:
                if doc in result:
                    result[doc] += 1
                else:
                    result[doc] = 1

            outFilePath = os.path.join(outputPath, f"result_{abs(hash(word)) % 100000}.txt")
            with open(outFilePath, 'a') as outFile:
                outFile.write(f'{word}:')
                for key, count in result.items():
                    outFile.write(f'({key},{count}) ')
                outFile.write('\n')
```

1.2 Reduce Phase

Terms are redistributed based on their first letter, with each process handling specific alphabet sections.

Implementation

Reduce Phase Algorithm:

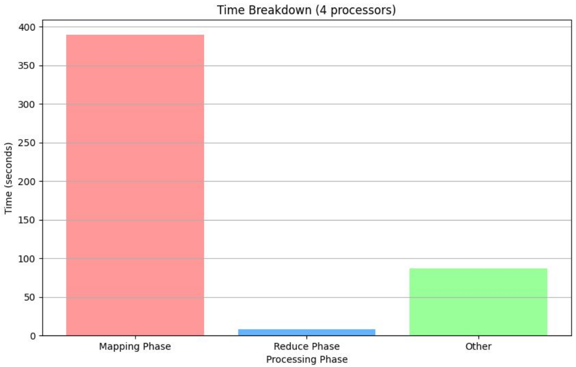
- Each process is responsible for specific alphabet subset
- Terms redistributed based on first letter
- Merges postings lists from all processes

Communication Pattern:

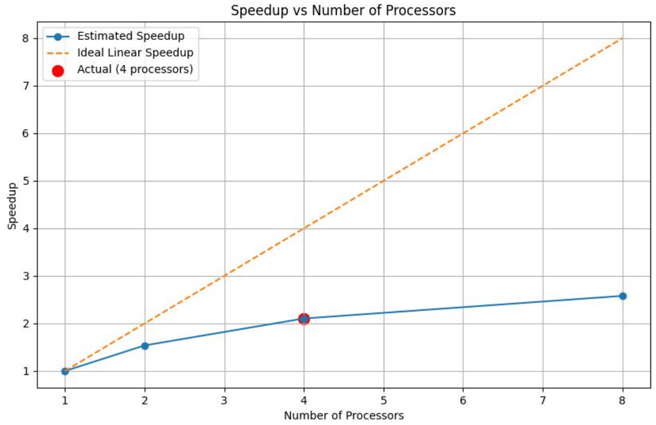
- File distribution from rank 0 process
- Independent processing during Map phase
- Term redistribution with consistent hashing
- Final merging coordinated through designated process

Results

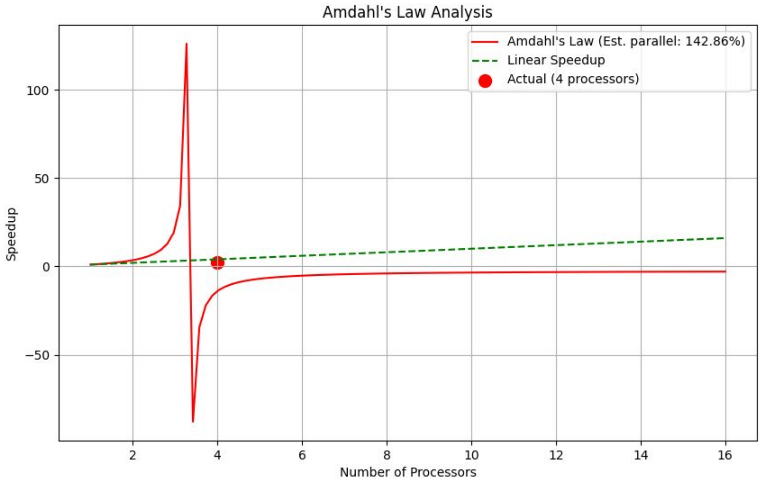
```
Single Run Performance Summary
==============================

Processors: 4
Total Words Processed: 1721178

Total Execution Time: 483.99 seconds
Mapping Phase: 389.71 seconds (80.52%)
Reduce Phase: 7.84 seconds (1.62%)
Other Time: 86.45 seconds (17.86%)
```

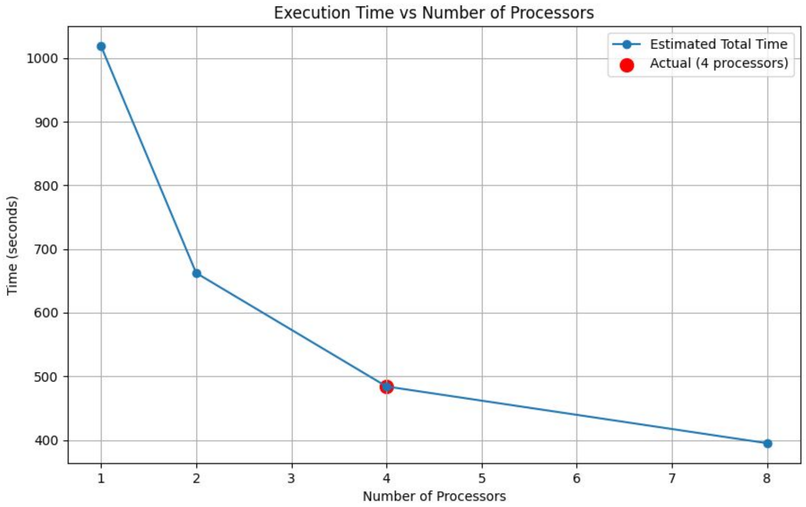



```
Performance for Different Processor Counts:
  1 processors: 1018.94s (Speedup: 1.00x)
  2 processors: 662.31s (Speedup: 1.54x)
  4 processors: 483.99s (Speedup: 2.11x)
  8 processors: 394.84s (Speedup: 2.58x)
```



Results


Amdahl's Law Analysis
Amdahl's Law (Est. parallel: 142.86%)
Linear Speedup
Actual (4 processors)
Speedup
100
50
0
−50
2
4
6
8
10
12
14
16
Number of Processors


Execution Time vs Number of Processors
Estimated Total Time
Actual (4 processors)
Time (seconds)
1000
900
800
700
600
500
400
1
2
3
4
5
6
7
8
Number of Processors

Comments?