**MapReduce Implemented in C++ Using OpenMP and MPI Frameworks**

**Yashwanth Reddy Thoom, 50579138**

**Department of Computer Science, Arkansas State University, Jonesboro, US**

**Abstract:**

***An approach to implement MapReduce framework on MPI platform. In this approach, inter-process communication is achieved using MPI communication functions. There are some limitations when solving problems of both data intensive and computing intensive in Traditional MapReduce model. The message passing of MapReduce model is realized through the low layer of the distributed file system. It stores all information in disk, and then reads them from the disk while required.***

***In order to overcome the limitations of Traditional MapReduce approach, MPI (Message Passing Interface) based MapReduce model is proposed. In this project I used MPI communication for cluster level and OpenMp on node level. In this, the master processor takes an input file from the list of files provided as input & the other processors (workers) wait for the task from the master node. Once the data is reduced by the workers is sent back to the Master and then final reduced data is printed to the stdout.***

1. **Introduction:**

**MPI and OpenMP**:

MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.” MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

**Some of the functions used in the code are**:

\*) map<string,int> mapreduce\_omp(int num\_threads,int mpi\_size, int mpi\_rank);

It performs Reading and Mapping on the input Map data provided by the MapReduce\_mpi program, and then using a hash function's output to figure out which reducer will reduce a particular word. Therefore, as each reducer finishes it reduce work, the result for a word is updated in a final key-value pair map (e.g. <"word", count>) and after all of them are done, the map is returned.

Some of the MPI calls used in this are:

MPI\_Init() MPI\_Allreduce() MPI\_Finalize()

MPI\_Comm\_rank() MPI\_Barrier() MPI\_Wtime()

MPI\_Comm\_size() MPI\_Send()

MPI\_Status() MPI\_Receive()

Some of the Openmp directive used:

#pragma omp parallel #pragma omp barrier

#pragma omp critical #pragma omp parallel shared(List attributes)

1. **Project Overview:**

MapReduce is a programming model that involves two steps. The first, the map step, takes an input set I and groups it into N equivalence classes I0, I1, I2, ..., IN-1. I can be thought of as a set of tuples <key, data>, and the function map maps I into the equivalence classes based on the value of key. In the second reduce step, the equivalence classes are processed, and the set of tuples in an equivalence class Ij are reduced into a single value.

1. The project uses MPI and OpenMP frameworks by including 'mpi.h' and 'omp.h' header files.
2. There are 4 important files in this implementation. They are described below:

a) **list.txt:** The list of files to be read is stored here.

b) **mapreduce\_omp.cpp:** This defines a function that does initial mapReduce. This function is called in mapreduce\_mpi.cpp.

c) **mapreduce\_omp.h:** This header file declares the function defined in mapreduce\_omp.cpp.

d) **mapreduce\_mpi.cpp:** This uses the MPI libraries to parallelize MapReduce Operations on multiple nodes (processes). It finally prints the result to stdout.

1. **Implementation:**
2. **mapreduce\_omp.cpp:**

If there are N files to read and k processes, then (approx..) N/k files are read by each process. Each process further divides the workload amongst multiple threads.

There are 3 kinds of threads:

* **Reader threads (all even thread ids)**, which read files and put the data read into a work queue 1. Each work item will be a word.
* **Mapper threads (all odd thread ids)**, which execute in parallel with Reader threads, create combined records of words. That is, if one mapper thread dequeues 100 instances of "A" and 50 instances of "B" and so on., its output will have {<"A", 100>, <"B", 50>, ...}. Mappers and Readers work in parallel and interact by means of the work queue 1. The access to the work queue is controlled by locks. After Reader threads are finished and the work queue is empty, the Reader threads become Reducer threads.
* **Reducer threads** that operate on work queue entries created by mapper threads and combine (reduce) them to a single record. Thus, for every word, there is potentially a <“word”, count of word > record sent by every mapper thread in the system and it will sum all of the counts and place it on a work queue 2.

Note: For each word there is exactly one Reducer thread in the system that handles it. A hash function's output is used to figure out which reducer will reduce a particular word. Therefore, as each reducer finishes it reduce work, the result for a word is updated in a final key-value pair map (e.g. <"word", count>) and after all of them are done, the map is returned.

1. **mapreduce\_mpi.cpp:**
2. After each process obtains its rank, it calls the mapreduce\_omp function that performs the above-mentioned tasks.
3. Now, each process has an initial key-value map that contains set of all unique words encountered in the books that it read and their counts.
4. Now, for the final reduce step, all the instances of a particular word (let's say "Alpha"), must be sent to a single process to reduce.
5. As before, a hash function output is used to decide which process should receive the entry corresponding to a particular word.
6. Now, each of the k processes keep a set of k maps, one for each process and stores the key-value pair that it must send to that particular process in that particular map. For example, if "Alpha" should go to process 2, all process will store the <"Alpha", local count> in map 2.
7. Finally, all processes communicate with each other to send the words and counts.
8. Now, we are ready to perform the final reduction. As processes receive messages from others, they perform the final reduction.
9. The outputs are then sent to stdout.
10. **Number of threads**:

Number of threads (on each process) on which the algorithm must run can be set in

mapreduce\_mpi.cpp. At the beginning of the code, there is a #define directive to set this.

1. **Compile and Run:**

**Compilation:**

mpic++ mapreduce\_mpi.cpp mapreduce\_omp.cpp -o mapreduce\_mpi -fopenmp

**To run:**

mpiexec -n “# of processors” ./mapreduce\_mpi

1. **Outputs**

Screenshot explaining the processor number on which server it is running.

A screenshot of a computer screen

Description automatically generated

Screenshot explaining the output of the Final map which stores the word and its count.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

1. **References**

* E. Culler, J. P. Singh, and A. Gupta, Parallel Computer Architecture: A Hardware/software Approach, Gulf Professional, 1999.C. Sodan, “Message-passing and shared-data programming models—wish vs. reality,” in Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS '05), pp. 131–139, May 2005.

View at: Publisher Site | Google Scholar

* K. M. Lee and K. M. Lee, “Similar pair identification using locality-sensitive hashing technique,” in Proceedings of 6th International Conference on Soft Computing and Intelligent Systems, and 13th International Symposium on Advanced Intelligence Systems (SCIS/ISIS '12), pp. 2117–2119, 2012.

View at: Google Scholar

* H. Lee-Kwang, K. A. Seong, and K. M. Lee, “Hierarchical partition of nonstructured concurrent systems,” IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, vol. 27, no. 1, pp. 105–108, 1997.

View at: Publisher Site | Google Scholar

* S. W. Lee, J. T. Kim, H. Wang et al., “Architecture of RETE network hardware accelerator for real-time context-aware system,” Lecture Notes in Computer Science, vol. 4251, pp. 401–408, 2006.

View at: Google Scholar

* OpenMP Architecture Review Board, “OpenMP Application Program Interface,” 2008, <https://www.openmp.org/wp-content/uploads/spec30.pdf>

View at: Google Scholar

* B. Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory, 2007, <https://computing.llnl.gov/tutorials/parallel_comp/>
* POSIX-IEEE Standards Association, 2014,

<https://standards.ieee.org/project/index.html>

* M. Macedonia, “The GPU enters computing's mainstream,” Computer, vol. 36, no. 10, pp. 106–108, 2003.

View at: Publisher Site | Google Scholar

* Alexandrov, S. Ewen, M. Heimel et al., “MapReduce and PACT—comparing data parallel programming models,” in Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW '11), pp. 25–44, 2011.

View at: Google Scholar