Hindawi Publishing Corporation
Advances in Multimedia
Volume 2015, Article ID 575687, 9 pages
http://dx.doi.org/10.1155/2015/575687

Research Article

Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems

Sol Ji Kang, Sang Yeon Lee, and Keon Myung Lee

Department of Computer Science, Chungbuk National University, Cheongju, Chungbuk 361-763, Republic of Korea

Correspondence should be addressed to Keon Myung Lee; kmlee@cbnu.ac.kr

Received 29 August 2014; Accepted 8 November 2014

Academic Editor: Seungmin Rho

With problem size and complexity increasing, several parallel and distributed programming models and frameworks have been developed to efficiently handle such problems. This paper briefly reviews the parallel computing models and describes three widely recognized parallel programming frameworks: OpenMP, MPI, and MapReduce. OpenMP is the de facto standard for parallel programming on shared memory systems. MPI is the de facto industry standard for distributed memory systems. MapReduce framework has become the de facto standard for large scale data-intensive applications. Qualitative pros and cons of each framework are known, but quantitative performance indexes help get a good picture of which framework to use for the applications. As benchmark problems to compare those frameworks, two problems are chosen: all-pairs-shortest-path problem and data join problem. This paper presents the parallel programs for the problems implemented on the three frameworks, respectively. It shows the experiment results on a cluster of computers. It also discusses which is the right tool for the jobs by analyzing the characteristics and performance of the paradigms.

1. Introduction

We often happen to meet problems requiring heavy computations or data-intensive processing. Hence, on one hand, we try to develop efficient algorithms for the problems. On the other hand, with the advances of hardware and parallel and distributed computing technology, we are interested in exploiting high performance computing resources to handle them.

Parallel and distributed computing technology has been focused on how to maximize inherent parallelism using multicore/many-core processors and networked computing resources [1–6]. Various computing architectures and hardware techniques have been developed such as symmetric multiprocessor (SMP) architecture, nonuniform memory access (NUMA) architecture, simultaneous multithreading (SMT) architecture, single instruction multiple data (SIMD) architecture, graphics processing unit (GPU), general purpose graphics processing unit (GPGPU), and superscalar processor [1, 7].

A variety of software technology has been developed to take advantage of hardware capability and to effectively develop parallel and distributed applications [2, 8]. With the plentiful frameworks of parallel and distributed computing, it would be of great help to have performance comparison studies for the frameworks we may consider.

This paper is concerned with performance studies of three parallel programming frameworks: OpenMP, MPI, and MapReduce. The comparative studies have been conducted for two problem sets: the all-pairs-shortest-path problem and a join problem for large data sets. OpenMP [9] is the de facto standard model from shared memory systems, MPI [10] is the de facto standard for distributed memory systems, and MapReduce [11] is recognized as the de facto standard framework intended for big data processing. For each problem, the parallel programs have been developed in terms of the three models, and their performance has been observed.

The remainder of the paper is organized as follows: Section 2 briefly reviews the parallel computing models and Section 3 presents the selected programming frameworks in more detail. Section 4 explains the developed parallel programs for the problems with the three frameworks.

Section 5 shows the experiment results and finally Section 6 draws conclusions.

2. Parallel Computing Models

In parallel computing memory architectures, there are shared memory, distributed memory, and hybrid shared-distributed memory [12]. Shared memory architectures allow all processors to access all memories as global memory space. They have usually been classified as uniform memory access (UMA) and NUMA. UMA machines are commonly referred to as SMP and assume all processors to be identical. NUMA machines are often organized by physically linking two or more SMPs in which not all processors have equal access time to all memories.

In distributed memory architectures, processors have their own memory, but there is no global address space across all processors. They have a communication network to connect processors' memories.

Hybrid shared-distributed memory employs both shared and distributed memory architectures. In clusters of multicore or many-core processors, cores in a processor share their memory and multiple shared memory machines are networked to move data from one machine to another.

There are several parallel programming models which allow users to specify concurrency and locality at a high level: thread, message passing, data parallel, and single program multiple data (SPMD) and multiple program multiple data (MPMD) models [12].

Thread model organizes a heavy weight process with multiple light weight threads that are executed concurrently. POSIX threads library (a.k.a. pthreads) [13] and OpenMP [9] are typical implementation of this model.

In the message passing model, an application consists of a set of tasks which use their own local memory that can be located in the same machine or across a certain number of machines. Tasks exchange data by sending and receiving messages to conduct the mission. MPI [10] is the de facto industry standard for message passing.

Data parallel model, also referred to as partitioned global address space (PGAS) model [14], provides each process with a view of the global memory even though memory is distributed across the machines. It makes distinction between local and global memory reference under the control of programmer. The compiler and runtime take care of converting remote memory access into message passing operations between processes [7]. There are several implementations of the data parallel model: Coarray Fortran, Unified Parallel C, X10, and Chapel [12].

SPMD model is a high level programming paradigm that executes the same program with different data multiple times. It is probably the most commonly used parallel programming model for clusters of nodes [12]. MPMD model is a high level programming paradigm that allows multiple programs to run on different data. With the advent of general purpose graphical processing unit (GPGPU), hybrid parallel computing models have been developed to utilize the many-core GPU to perform heavy computation under the control of the host thread running on the host CPU [15].

When data volume is large, demanding memory capacity may hinder its manipulation and processing. To deal with such situations, big data processing frameworks such as Hadoop and Dryad have been developed which exploit multiple distributed machines. Hadoop MapReduce is a programming model that abstracts an application into two phases of Map and Reduce [16]. Dryad structures the computation as a directed graph in which vertices correspond to a task and edges are the channels of data transmissions [17].

3. OpenMP, MPI, and MapReduce

3.1. OpenMP. OpenMP is a shared-memory multiprocessing application program inference (API) for easy development of shared memory parallel programs [9]. It provides a set of compiler directives to create threads, synchronize the operations, and manage the shared memory on top of pthreads. The programs using OpenMP are compiled into multithreaded programs, in which threads share the same memory address space and hence the communications between threads can be very efficient.

Compared to using pthreads and working with mutex and condition variables, OpenMP is much easier to use because the compiler takes care of transforming the sequential code into parallel code according to the directives [12]. Hence the programmers can write multithreaded programs without serious understanding of multithreading mechanism. Its runtime maintains the thread pool and provides a set of libraries [7].

It uses a block-structured approach to switch between sequential and parallel sections, which follows the fork/join model. At the entry of a parallel block, a single thread of control is split into some number of threads, and a new sequential thread is started when all the split threads have finished. Its directives allow the fine-grained control over the threads. It is supported on various platforms like UNIX, LINUX, and Windows and various languages like C, C++, and Fortran [12].

3.2. MPI. MPI is a message passing library specification which defines an extended message passing model for parallel, distributed programming on distributed computing environment [10]. It is not actually a specific implementation of the parallel programming environment, and its several implementations have been made such as OpenMPI, MPICH, and GridMPI [7]. In MPI model, each process has its own address space and communicates other processes to access others' address space. Programmers take charge of partitioning workload and mapping tasks about which tasks are to be computed by each process.

MPI provides point-to-point, collective, one-sided, and parallel I/O communication models [10]. Point-to-point communications enable exchanging data between two matched processes. Collective communication is a broadcast of message from a process to all the others. One-sided communications facilitate remote memory access without matched process on the remote node. Three one-sided libraries are available for remote read, remote write, and remote update [9]. MPI provides various library functions to

coordinate message passing in various modes like blocked and unblocked message passing. It can send messages of gigabytes size between processes.

MPI has been implemented on various platforms like Linux, OS X, Solaris, and Windows. Most MPI implementations use some kind of network file storage. As network file storage, network file system (NFS) and Hadoop HDFS can be used. Because MPI is a high level abstraction for parallel programming, programmers can easily construct parallel and distributed processing applications without deep understanding of the underlying mechanism of process creation and synchronization. To order to exploit the multicore of processors, the MPI processes can be organized to have multiple threads in themselves. MPI-based programs can be executed on a single computer or a cluster of computers [18].

3.3. MapReduce. MapReduce is a programming paradigm to use Hadoop which is recognized as a representative big data processing framework [11]. Hadoop clusters consist of up to thousands of commodity computers and provide a distributed file system called HDFS which can accommodate big volume of data in a fault-tolerant way [19]. The clusters become the computing resource to facilitate big data processing [20, 21].

MapReduce organizes an application into a pair (or a sequence of pairs) of Map and Reduce functions. It assumes that input for the functions comes from HDFS file(s) and output is saved into HDFS files. Data files consist of records, each of which can be treated as a key-value pair. Input data is partitioned and processed by Map processes, and their processing results are shaped into key-value pairs and shuffled into Reduce tasks according to key. Map processes are independent of each other and thus they can be executed in parallel without collaboration among them. Reduce processes play role of aggregating the values with the same key.

MapReduce runtime launches Map and Reduce processes with consideration of data locality. The programmers do not have to consider data partitioning, process creation, and synchronization. The same Map and Reduce functions are executed across machines. Hence, MapReduce paradigm can be regarded as a kind of SPMD model.

MapReduce paradigm is a good choice for big data processing because MapReduce handles data record by record without loading whole data into memory and in addition the program is executed in parallel over a cluster [20]. It is very convenient to develop big data handling programs using MapReduce because Hadoop provides everything needed for distributed and parallel processing behind the scene which program does not need to know.

4. Parallel Programs for the All-Pairs-Shortest-Path Problem and a Join Problem

This paper is concerned with comparative performance studies of OpenMP, MPI, and MapReduce. OpenMP and MPI have been compared earlier in [3, 22], but they have not been usually compared with MapReduce because MapReduce does not assume any special memory architecture. With increasing interest in big data, many practitioners are interested in using MapReduce to handle big volume of data and sometimes computation-intensive problems [20].

$n \leftarrow$ size of rows
$D^{(0)} \leftarrow$ input distance matrix
$\Pi^{(0)} \leftarrow$ calculate precedence matrix
for $k \leftarrow 1$ to n
  do for $i \leftarrow 1$ to n
    do for $j \leftarrow 1$ to n
      do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

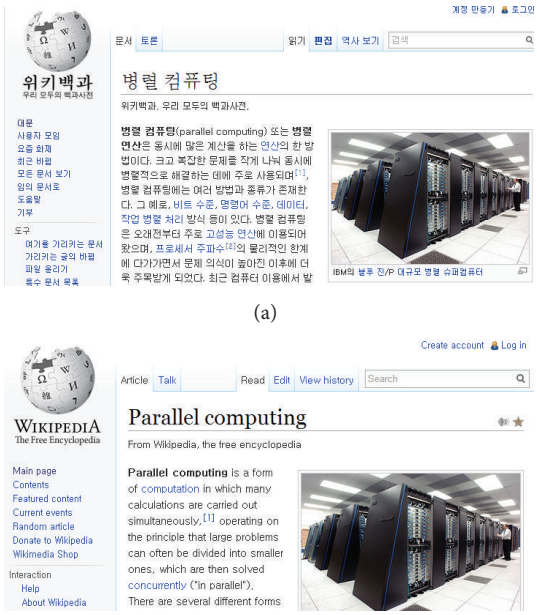
return $D^{(n)}$ and $\Pi^{(n)}$

Algorithm 1: Floyd-Warshall algorithm for the all-pairs-shortest-path problem.

Once you understand their design principle and underlying mechanism of the frameworks, you might guess which seems to be better in which situation. This study is intended to get quantitative figures about their performance. Here we choose as benchmark problems two problems: the all-pairs-shortest-path problem as a computation-intensive one and a join problem as a data-intensive one.

4.1. Benchmark Problems. The all-pairs-shortest-path problem is to find the shortest path between all pairs of nodes in a graph. The problem occurs in domains of communication networking, logistics planning, layout design, navigation, and so on. Floyd-Warshall algorithm is one of the best known algorithms to this problem, which iteratively searches the shortest paths by considering the intermediate nodes one by one [23]. Algorithm 1 shows Floyd-Warshall algorithm, where $D = (d_{ij})$ denotes the matrix containing the distance d_{ij} between nodes i and j, $\Pi = (\pi_{ij})$ is the precedence matrix containing the information about the shortest paths between nodes, $d_{ij}^{(k)}$ is the distance found which is obtained by using the node set $\{1, 2, \ldots, k\}$ as intermediate nodes, and $\pi_{ij}^{(k)}$ is the predecessor of node j on the shortest path from node i to j obtained by using the node set $\{1, 2, \ldots, k\}$ as intermediate nodes.

As the second benchmark problem, we choose the task to find associated English Wikipedia pages with English keywords appearing in Korean Wikipedia pages and then place the hyperlinks from the English words of Korean Wikipedia pages to the found English Wikipedia pages. Wikipedia is a free-access Internet encyclopedia which has 287 language editions (as of August 2014). Figure 1(a) shows the Korean Wikipedia page for the word "parallel computing" of which content is mostly written in Korean and some notable terms like "parallel computing" are put into the parenthesis in English. Figure 1(b) shows the corresponding English Wikipedia page for the term "parallel computing." We want to place a hyperlink from the word "parallel computing" of the page of Figure 1(a) to the page of Figure 1(b).



(a)

(b)

Figure 1: A join problem to insert a hyperlink from English words "parallel computing" of Korean Wikipedia pages to the corresponding page of English Wikipedia. (a) Korean Wikipedia (http://ko.wikipedia.com/) and (b) English Wikipedia (http://en.wikipedia.com/).

The hyperlink construction problem can be regarded as a join problem: Korean Wikipedia pages are reviewed as a table of the pairs of English word(s) and the corresponding page contents, and English pages are also reviewed as a table of the pairs of their title and their URL. The inner join between two tables gives all needed information for hyperlink construction. In the benchmark problem, we have an English Wikipedia XML archive with 4,664,819 articles of size 4.04 gigabytes and a Korean Wikipedia XML archive with 529,997 articles of size 1.35 gigabytes. Hence the problem is data-intensive one to deal with large volume of data.

4.2. Distributed Programs for the All-Pairs-Shortest-Path Problem. To compare the performance of OpenMP, MPI, and MapReduce models for the all-pairs-shortest-path problem, the distributed programs are developed based on each model as follows. All the programs in fact implement the Floyd-Warshall algorithm in parallel executing codes.

Algorithm 2 shows the pseudocode for solving the all-pairs-shortest-path problem which is based on the OpenMP. In the code, "parallel start" indicates the parallel execution directive which spawns a team of threads as needed to take care of the loop in a partitioned manner. "Parallel end" indicates the place at which all threads join together into a single thread.

Algorithm 3 shows the MPI-based pseudocode for the all-pairs-shortest-path problem. In the code, "MPI Init" indicates the place to load and initialize the MPI library, and "MPI Finalize" is the part to wrap up the MPI processing. According to the process ID "pid," each process takes care

$n \leftarrow$ size of rows
$D^{(0)} \leftarrow$ input distance matrix
$\Pi^{(0)} \leftarrow$ calculate precedence matrix
tN $\leftarrow$ number of threads
for $k \leftarrow 1$ to n
  parallel start
    tid $\leftarrow$ id of thread
    for $i \leftarrow \dfrac{\text{tid} * n}{\text{tN}}$ to $\dfrac{(\text{tid}+1) * n}{\text{tN}} - 1$
      for $j \leftarrow 1$ to n

$$d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

  parallel end
return $D^{(n)}$ $\Pi^{(n)}$

Algorithm 2: OpenMP pseudocode for the all-pairs-shortest-path problem.

MPI Init
$n \leftarrow$ size of rows
pid $\leftarrow$ id of process
pN $\leftarrow$ number of processes
$D^{(0)} \leftarrow$ input distance matrix
$\Pi^{(0)} \leftarrow$ calculate precedence matrix
for $k \leftarrow 1$ to n
  do for $i \leftarrow \dfrac{\text{pid} * n}{\text{pN}}$ to n
    do for $j \leftarrow 1$ to n

$$\text{do } d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{dj}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

    send i'st row to another processes
    receive updated rows from another processes
return $D^{(n)}$ $\Pi^{(n)}$
MPI Finalize

Algorithm 3: MPI pseudocode for the all-pairs-shortest-path problem.

of the assigned part and sends and receives the intermediate results to and from other processes.

Algorithm 4 shows the pseudocode based on MapReduce model. MapReduce model abstracts the job with Map and Reduce functions. Map function is applied to each record of the input file where a line contains a record. Reduce function takes the whole outputs of Map function and aggregates them into final results. The implemented application iteratively repeats these MapReduce phases as many as the number of nodes in the input graph. In the code, "Driver( )" plays the role of a coordinate to invoke MapReduce cycles iteratively. An input record [key = $(i \quad j)$, val = $(d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)})$] of Map function means that the so far shortest distance from node i to j is $d_{ij}^{(k-1)}$ and the preceding node of node j in the shortest path is $\pi_{ij}^{(k-1)}$, and "write($i \quad j, \quad i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}$)" indicates

input: $\left[i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)} \right]$
Map(Object key = $(i \quad j)$, Value val = $\left(d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)$)
  if $i == k$ or $j == k$ then
    for $m \leftarrow 1$ to n
      if $j == k$ then write$(j \quad m), \left(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)$
      if $i == k$ then write$(m \quad i), \left(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)$
    else then write$(i \quad j), \left(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)$
Reduce(Object key = $(i \quad j)$, Value val = $\left(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)$)
  $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
  $\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$
  write$(i \quad j), \left(d_{ij}^{(k)} \quad \pi_{ij}^{(k)}\right)$
Driver( )
  $n \leftarrow$ size of rows
  for $k \leftarrow 1$ to n
    Map$\left((i \quad j), \left(d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)\right)$
    Reduce$\left((i \quad j), \left(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)}\right)\right)$

Algorithm 4: MapReduce pseudocode for the all-pairs-shortest-path problem.

the operation to write out a record with key $(i \quad j)$ and value $(i \quad j \quad d_{ij}^{(k-1)} \quad \pi_{ij}^{(k-1)})$ to a HDFS file.

4.3. Distributed Programs for the Join Problem. In the join problem, there are two XML files: a Korean Wikipedia XML file and an English Wikipedia file. First, we need to extract the English terms from Korean Wikipedia file and create a table of records, each of which consists of English term(s) and the original page content for the later use. The English Wikipedia file is transformed into a table of records consisting of the page title and the page URL. After that, the two tables are joined to obtain the information about hyperlinks to be added.

Algorithm 5 shows the pseudocode for the join problem to obtain the hyperlink information by using the OpenMP construct. The two files are too large to be fit into the main memory. Hence, they are partitioned and the larger file (i.e., English Wikipedia file) is sequentially read into the memory one by one just once, and the partitions of the smaller file are read sequentially as many as the number of partitions of the larger files.

Algorithm 6 shows the MPI pseudocode for the join problem. In the code, "send" and "receive" denote the MPI communication APIs for data exchange. Each process takes care of the partitioned job and files are read line by line which is supported for large volume of data by the operating system like Linux.

Algorithm 7 shows the MapReduce codes of two Map functions for both Korean Wikipedia articles and English Wikipedia articles and a Reduce function. The Korean Wikipedia Map function extracts English word(s) from each Korean page and produces records of word(s) and the entire article. The English Wikipedia Map function extracts the title and URL of an English page and generates a record with them. The Reduce function receives the outputs of both Maps and finds matched pairs to build the information of hyperlinks from words in Korean pages to English pages. Figure 2 shows the configuration of Map and Reduce functions to extract the hyperlink information from the two files.

5. Experiments

For the two benchmark problems, the parallel programs have been implemented using OpenMP, MPI, and MapReduce, respectively. In the experiments, a cluster of 5 PCs was used, each of which has Intel Core i7-4770 3.40 GHz CPU and 16 GB RAM, and was installed with CentOS-6.4 LINUX 64 bits. For MapReduce applications, Hadoop HDFS and YARN were installed over the cluster where HDFS is the distributed file system in Hadoop and YARN is a resource and application manager. MPICH, which is an implementation of MPI, was installed at each machine on YARN and HDFS. OpenMP was installed on a single node because it supports only shared memory model but not distributed memory model.

For the all-pairs-shortest-path problem, three sample graphs were randomly generated of which the numbers of nodes were 10, 100, and 1000, respectively. When they were generated, each node was set to be linked to half of the other nodes. When final results are written, there can exist a bottleneck if a single file is used as the output. Hence, for the fair comparisons, each process or thread is allowed to write its output into its own out-file.

Table 1 shows the execution time obtained in the all-pairs-shortest-path problem experiments. For this computation-intensive problem, the OpenMP program gave the best performance where 10 threads were used. The MPI program was executed on a single machine and on the cluster of 5 machines with total of 10 processes. Due to the computational

```
OMP_JOIN_PAGE(char* koFile, char* enFile, char* outFile)
  tN ← number of threads
  eS ← block size of English Wikipedia file to be read at a time
  eN ← number of blocks in English Wikipedia file, enFile
  kS ← block size of Korean Wikipedia file to be read at a time
  kN ← number of blocks in Korean Wikipedia file, koFile
  for i ← 1 to eN
    eBlock ← ith block of enFile
    for k ← 1 to kN
      kBlock ← kth block of koFile
      parallel-start
        tid ← id of thread
        for j ← (tid * eS)/tN to ((tid + 1) * eS)/tN − 1
          text ← eBlock.nextLine()
          title ← searchTitle(text)
          url ← searchUrl(text)
          foreach text ∈ kBlock
            content ← searchContent(text)
            koUrl ← searchUrl(text)
            wordList[] ← extractEngKeyword(content)
            foreach word ∈ wordList[] do
              if isMatched(word, title) then
                enUrl ← url
              else
                enUrl ← φ
              if enUrl ≠ φ
                write(title, koUrl, word, enUrl)
      parallel-end
```

Algorithm 5: OpenMP pseudocode for the join problem.

Table 1: Execution times for the all-pairs-shortest-path problem.

Node size	Framework			
	MapReduce	MPI		OpenMP
		Cluster	Single machine	
10	2 m 26 s	0.32 s	0.34 s	0.1 s
100	16 m 52 s	0.44 s	0.41 s	0.25 s
1000	4 h 4 m 39 s	4 m 48 s	24.14 s	8.03 s

Table 2: Execution time for the join problem.

Problem	Framework		
	MapReduce	MPI	OpenMP
The join problem	24 m 15 s	135 h 34 m	93 h 14 m

overhead, the cluster showed poor performance for the MPI program. In the experiment setting, the communication bottleneck was severe even though the machines were connected with a 1 Gbps switching hub. The performance of MPI on a single machine is not comparable to OpenMP, because OpenMP threads share the global address space but MPI processes communicate using the message passing protocol. If some application can be run on a high-end single machine, OpenMP is preferred to MPI. MapReduce is not a choice for computational-intensive and iterative computation problems like the all-pairs-shortest-path problem.

Table 2 shows the experiment results for the join problem. The execution time varies depending on the execution context like network bandwidth and resource management of operating systems, hence the same experiments have been conducted three times for each setting. For fair comparisons, no special indexing structures like B+ tree [23] were used in implementing the join operations. However, the same logic was implemented on each framework. The MapReduce-based program was the best one among the three models. In the join operation, we need only the title and URL of English Wikipedia pages and hence the implementations of MPI and OpenMP just scan the English pages as a stream and keep the whole set of the pairs of the title and the URL in the memory. Then, Korean Wikipedia pages are read one by one in order to examine which English terms appear. OpenMP gave better performance than MPI for the join problem because the curated information for English pages was loaded into the memory in a whole, and OpenMP threads have no communication overhead to access the memory. From these observations, we see that MapReduce is the best choice for data-intensive processing of big volume of data.

```
MPI_JOIN_PAGES(char* koFile, char* enFile, char* outFile)
  koFileLineSize ← getFileLines(koFile)
  MPI-Init
  nRank ← process ID number within communicator
  divLine ← the number of each process's allocated lines
  line ← move file pointer to the first of allocated lines
  if nRank = 0 then
    while true do
      send (NULL, NULL, NULL, NULL, true)
        to another processes
      receive (title, koUrl, word, enUrl, isLast)
        from another processes
      out.Write(title, koUrl, word, enUrl)
      if isLast(ALL) then break
  else
    for i = 0 to divLine do
      text ← enFile.nextLine(line + i)
      title ← searchTitle(text)
      content ← searchContent(text)
      koUrl ← searchUrl(text)
      wordList[] ← extractEngKeyword(content)
      foreach word ∈ wordList[] do
        if isExist(word, enFile) then
          enUrl ← getUrl(word, enFile)
        else
          enUrl ← φ
        send (title, koUrl, word, enUrl, isLast)
          to another processes
        receive (title, koUrl, word, enUrl, isLast)
          from another processes
      if isLast(ALL) then break
  MPI-Finalize
```

Algorithm 6: MPI pseudocode for the join problem.

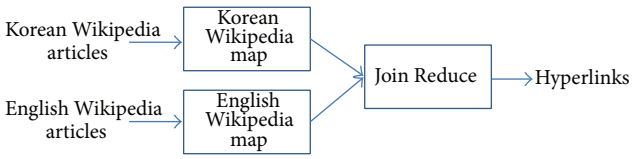


Figure 2: MapReduce pseudocode for the join problem.

6. Conclusions

OpenMP, MPI, and MapReduce are the most widely recognized parallel or distributed programming frameworks. Each one is said to be the de facto standard on its computing model. To evaluate their performance, we implemented the programs for both the all-pairs-shortest-path problem and the join problem for hyperlink extraction from two files using the three frameworks, respectively. The performance of each model was measured in terms of the execution time.

From the experiment results, we observed the following things. If a problem is small enough to be accommodated and the computing resources such as cores and memory are sufficient, OpenMP is a good choice. When data size is moderate and the problem is computation-intensive, MPI can be considered the framework. When data size is large and the tasks do not require iterative processing, MapReduce can be an excellent framework. OpenMP is the easiest to use because there is no special attention needed to be paid because we just need to place some directives in the sequential code. MapReduce is relatively easy to use once we can abstract an application into Map and Reduce steps. The programmers do not have to consider workload partitioning and synchronization. MapReduce programs, however, take considerable time for the problems requiring much iteration, like all-pairs-shortest-path problem. MPI allows more flexible control structures than MapReduce; hence MPI is a good choice when a program is needed to be executed in parallel and distributed manner with complicated coordination among processes.

This study did not consider the CUDA (compute unified device architecture) model which is a parallel computing platform and programming model for GPUs [24]. It would be interesting to compare CUDA with the discussed programming models for some additional practical problem sets.

```
MAP for Korean Wikipedia:
  Input Korean Wikipedia XML File
  MAP(Object key = null, Page input = Korean Wikipedia file)
    line ← input.readLine()
    title ← searchTitle(line)
    content ← searchContent(line)
    wordList[] ← extractEngKeyword(content)
    for all word ∈ wordList[] do
      write(word, input)
MAP for English Wikipedia File
  Input: English Wikipedia XML File
  MAP(Object key = null, Page input = English Wikipedia file)
    line ← input.readLine()
    title ← searchTitle(line)
    url ← searchUrl(line)
    Page out(title:title, url:url)
    write(title, out)
REDUCE(Text key, Page [p1, p2, ...])
  word ← ∅
  url ← ∅
  list ← new List<Page>
  for all p ∈ [p1, p2, ...] do
    if IsEnglish(p) then
      word ← p.getTitle()
      url ← p.getUrl()
    else
      list.add(p)
  for all koreanPage ∈ list do
    koreanPage.setWord(word)
    koreanPage.setUrl(url)
    write(key, koreanPage)
```

Algorithm 7: MapReduce pseudocode for the join problem.

Conflict of Interests

No potential conflict of interests relevant to this paper was reported.

Acknowledgment

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support Program (NIPA-2013-H0301-13-4009) supervised by the NIPA (National IT Industry Promotion Agency).

References

[1] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/software Approach*, Gulf Professional, 1999.

[2] A. 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.

[3] 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.

[4] 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.

[5] 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.

[6] S. W. Lee, J. T. Kim, B. K. Sohn, K. M. Lee, J. W. Jeon, and S. Lee, *Real-Time System-on-a-Chip Architecture for Rule-Based Context-Aware Computing*, vol. 3681 of *Lecture Notes in Computer Science*, 2005.

[7] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[8] S. C. Ravela, "Comparison of shared memory based parallel programming models," Tech. Rep. MSC-2010-01, Blekinge Institute of Technology, 2010.

[9] OpenMP Architecture Review Board, "OpenMP Application Program Interface," 2008, http://www.openmp.org/mp-documents/spec30.pdf.

[10] W. Gropp, S. Huss-Lederman, A. Lumsdaine et al., *MPI: The Complete Reference, the MPI-2 Extensions*, vol. 2, The MIT Press, 1998.

[11] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, 2007, https://computing.llnl.gov/tutorials/parallel_comp/.

[13] POSIX-IEEE Standards Association, 2014, http://standards.ieee.org/develop/wg/POSIX.html.

[14] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey et al., "An evaluation of global address space languages: co-array fortran and Unified Parallel C," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming*, pp. 36–47, June 2005.

[15] M. Macedonia, "The GPU enters computing's mainstream," *Computer*, vol. 36, no. 10, pp. 106–108, 2003.

[16] A. 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.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Feterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[18] G. Jost, H. Jin, D. Mey, and F. Hatay, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," in *Proceedings of the 5th European workshop on OpenMP (EWOMP '03)*, 2003.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 29–43, October 2003.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA '07)*, pp. 13–24, Scottsdale, Ariz, USA, February 2007.

[21] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[22] M. Resch, B. Sander, and I. Loebich, "A comparison of OpenMP and MPI for the parallel CFD test case," in *Proceedings of the 1st European Workshop on OpenMP*, pp. 71–75, 1999.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2009.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.