Document classification in parallel environments using Java bindings in open MPI

Subhi Abdul-rahim Bahudaila and Waddah Ahmed Munasser
Department of Information Technology, Faculty of Engineering, Aden-University
(Email: sabhudail@yahoo.com, wmunassar@yahoo.com)
DOI: https://doi.org/10.47372/uajnas.2017.n2.a09

Abstract

This paper describes the high performance computing (HPC) of document search engines that are vectorizing each classified document. The parallelization is achieved by exploiting the parallel and distributed computing environments of collective multiple processes that are implemented by using Java message passing interface (MPI) bindings of openMPI. A parallelism model of manager/worker is implemented for obtaining the load balancing, as well as the analysis and benchmarking are achieved in our parallelism profiling model that is designed for the implementation. Two output of the experimental results are shown:- the parallel processing performance with the efficiency of 80%, and the profiling results that show the utilizations and overheads in our parallelism model.

Keywords: Java MPI bindings, Load-Balancing, Manager/Worker-style, Profiling parallelism, Vectoring classification

1. Introduction

In document classification problem, initial works occur in reading a dictionary and searching a directory structure for plain text files (such as .html, .xml, and .txt files). For each of these files, the program opens a file, reads its content, and generates a profile vector that indicates how many times the text document contains each word appearing in the dictionary. After that, the program writes a file containing the profile vectors for each of the plain text files it has examined.

High performance computing is implemented for parallelizing document classification using the functional decomposition with showing the drawbacks: tasks do not communicate to each other and the time needed to perform each task may vary widely because the documents may have different sizes and some of them may be more difficult to process than others.

The main contributions of this paper are the following:

- Enhancing the parallel performance with obtaining the load balancing in the experiment by designing the scheme of manager/worker and allocating small groups of tasks to workers (section 4).
- Nonblocking communication is implemented for improving the performance of function manager (section 4). It is due to allow the system to overlap communications and computations as quick as possible.
- Designing a new mechanism for profiling parallelism based on time indication of transitional states of each process that is structured in a log file containing of line messages (section 5).
- Implementing a profiler application of the proposed profiling parallelism (section 5.2), that estimates performance matrices such as execution time, speedup, efficiency and overheads of each process, as well as showing the results of profiling the computation-communication scenario (section 6.3).
- Providing the availability of the powerful MPI functionality of the latest MPI-3 features to Java programmers and applications by using Java MPI bindings of Open MPI environment in

the parallelization of document classification (section 6). Also, the MPI advance capabilities are used, such as: The *MPI.COMM_WORLD.split* function, spawning a new communicator, which facilitated the broadcasting of the dictionary among the workers. In addition to the *MPI.COMM_WORLD.probe* and *getCount* functions, checking the length of path names sent by the master before actually reading them.

2. Related Work

A number of research papers used the K-Nearest Neighbour (KNN), a classification technique in data mining, for document classification in parallel. The approach by Vyawahare et al. [8] viewed the mathematical model of the KNN for solving in serial and parallel executions. They used the multithreaded model of parallelization, in which a set of text documents are partitioned among the number of threads for handling their assigned documents for training, then they calculate the document vectors for testing. The implementation is executed on GPU. However, they did not consider the experimented results of the performance measurements. The performance measurements has here considered in our work and in a previous published one [1]. Lican Huang, Zhilong Li [4] presented the parallel GPU implementation of KNN with the performance results that exceeds 40 time, compared with CPU implementation.

Vyawahare et al. [8] achieved the accuracy results of the parallel KNN which is measured by precision (P), recall (R), and F-measure,as it is achieved by Polpinij et al. [5], who presented the accuracy results for sentiment classification based on support vector machine (SVM) algorithm in serial execution.

Subhi Bahudaila[2], in his work, addressed the profiling parallelism by using Intel VTune Thread Profiler. Considering the analyzing of a shared memory programming mode of hyper-threading platform that showed the utilization and overheads results of executing threads on logical processors of the hyperthreading technology by implementing the explicit parallelism of openMP, while our work has considered these investigations in distributed memory environments with open MPI in java binding [7] and [10] by implementing the computation-communication scenario for indicating the performance of each process, including its all states. So, the results of the profiling parallelism has been shown.

3. Document classification in Parallel

There are five steps that determine the data dependency of the document classification algorithm as follows:

Step 1 and 2 are used to read dictionary and to identify documents, step 3 is used to read documents, step 4 is used to generate document vectors and step 5 is used to write documents vectors in one 2D matrix.

The parallel algorithm was designed using a functional decomposition for document classification. Let's assume the reading documents and generating the profile vectors that consume the vast majority of the execution time. This gives us the sense, to generate two tasks for each document; one to read the document file and another to generate the vector. Both tasks can be assigned to one processing element per document, while identifying document and reading dictionary can be done concurrently is shown in Figure 1.

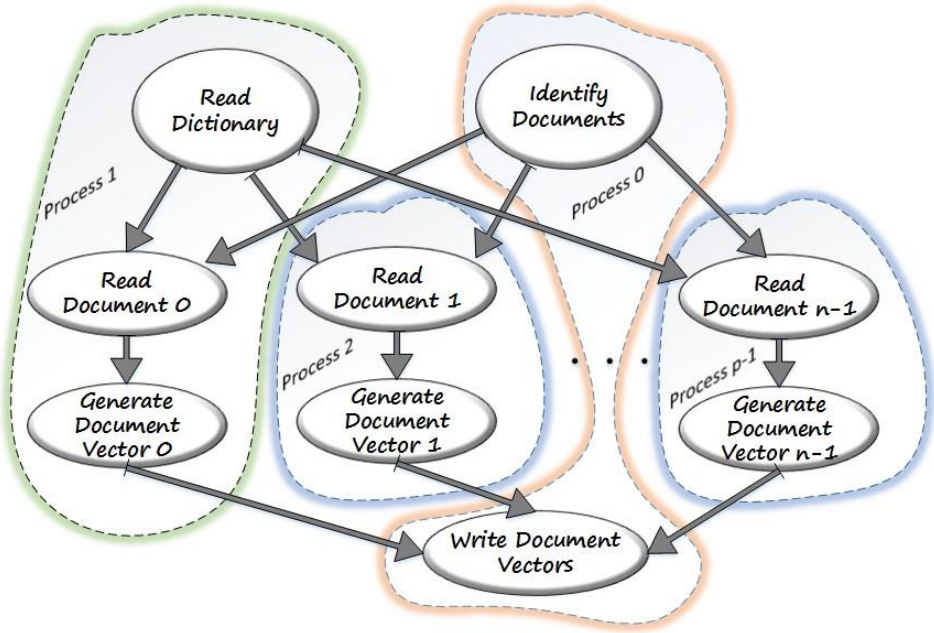

Figure 1: Data dependencies of parallel document classification algorithm

Figure 1 illustrates the task partitioning and the assignments of each process for the algorithm Parallelization.

4. Manager/Worker model

The Manager/Worker model is a type of dynamic load balancing. The number of tasks is not known at compile time and the tasks are allocated to processors during the execution of the program. To support practical dynamic allocation, Manager/Worker style parallel program should be constructed [9]. There is one process called manager with the process's rank = 0. The master is responsible of keeping track of assigned and unassigned tasks. It assigns tasks to other processes called workers with the process's ranks =1 to (p-1), and retrieves results back from them. The model is shown in Figure 2.

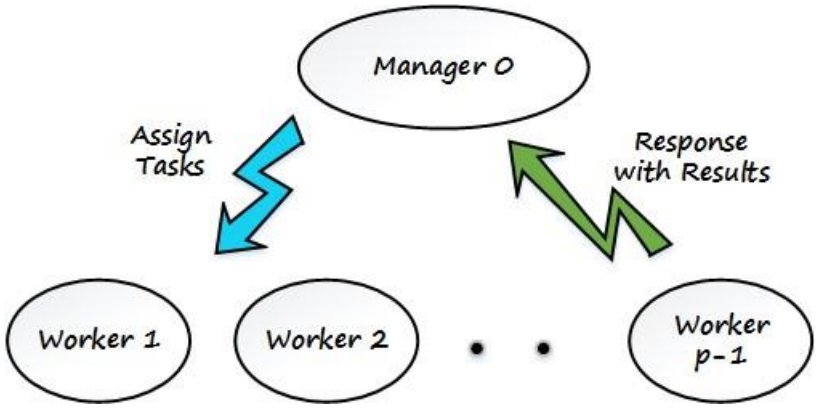

Figure 2: Manager/Worker Model

Table 1: Local variables of the Master/Worker style with their descriptions

Local variable	description
a	Array showing document assigned to each process
d	Documents assigned
f	filename
j	ID of worker representing document
k	Document vector length
n	Number of documents
p	Total number of processes (1 master and $p-1$ workers)
s	Storage array containing document vectors
t	Terminated workers
v	Individual document vector

The master and the worker algorithms are shown in Algorithm 1 and 2. In these algorithms, there are local variables *a*,*d*,*f*,*j*,*k*,*n*,*p*,*s*,*t*,*v*as described in Table 1.

Algorithm 1 Manager algorithm

```
1: function Manager
2:     local variables: a,d,j,k,n,p,s,t,v
3:     n ← identified documents number in user specified directory
4:     receive dictionary size k from the 1st worker
5:     allocate s with dimension n ∗k to store document vectors
6:     d,t← 0  ⇨to initialize nothing assignments and termination
7:     repeat
8:     receive message from worker j
9:     if message contains document vector v then s[a[j]] ← v
10:    else message is first request for work, do nothing
11:    end if
12:    if d < n then send name of document d to worker j
13:    a[j] ← d; d ← d + 1
14:    else send termination message to worker j; t ← t + 1end if
17:    until t = p − 1
18:    write s to output file
19: end function
```

Algorithm 2 Worker algorithm

```
1: function Worker
2:     local variables: f,k,v
3:     send first request for work to manager
4:       if wrank= 0 then Read dictionary from file ⇨ the 1st worker
5:     end if
6:     Broadcast dictionary around workers
7:     build hash table from dictionary
8:     if wrank= 0 then send dictionary size k to manager end if
10:    while true do
11:    receive file name f from manager
12:    if f indicates termination then Exit loop
13:    else
14:    read document from file f
15:    generate document vector v
```

16: **send** *v* to manager
17: **end if**
18: **end while**
19:**endfunction**

5. Designing scheme for profiling parallelism

5.1 Profiling scheme

For the sake of profiling parallelism, the structure of the log file is proposed for this article. The logging messages are lined messages, and each message contains three parts: process RANK, TIME in milliseconds, and message type SIGNAL. The signal message type takes one of five integer values as follows:

1BeginComp, **2**BeginWait, **3**BeginComm, **4**ResumeComp, and **5**EndComp The signal values are designed for indicating the five-state transition of each process, as shown in Figure 3.

In Figure 3, there is an associated signal message needed for each transition. Each signal indicates the transition state of a process five-state model. By using these messages, the parallel program transits from one state to another.

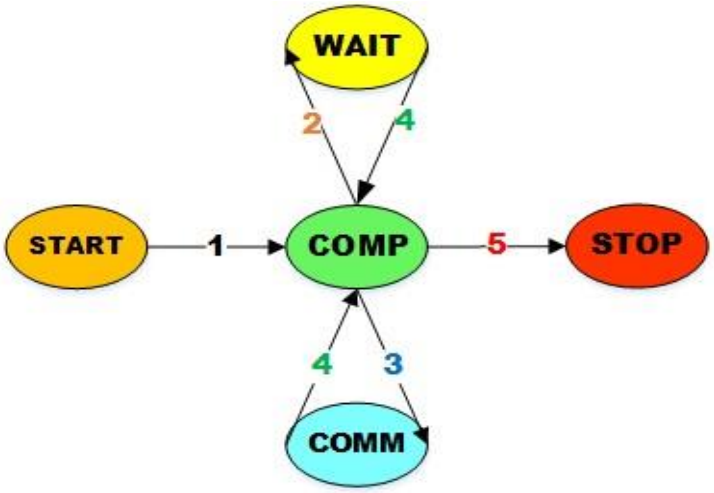

Figure 3: Five-state transition diagram

5.2 Profiling application

After the log is generated and MPI.csv file is formed, a MATLAB script is implemented to read MPI.csv file and estimates many of performance metrics which will be discussed later. The script is shown in Figure 4.

```
MPI = importfile('MPI.csv');
Num_Files = input('Enter the total number of files = ');
Num_Core = input('Enter number of Cores = ');
Num_Process = max(MPI[:,['rank']]) + 1;
Process_per_core = Num_Process ./ Num_Core;
Ref_time = min(MPI[:,['time']]);
max_time = max(MPI[:,['time']]);
MPI[:,['time']] = (MPI[:,['time']] - Ref_time) ./ Process_per_core;   %%Normalize time
wall_clock_time = (max_time - Ref_time)./ Process_per_core;
for i=0:Num_Process-1
    rows = MPI[:,['rank']] == i;
    subplot(Num_Process,1,i+1);
    stairs(MPI[rows,['time']],MPI[rows,['signal']]);
    grid on;
    axis([0 wall_clock_time 0 6]);
    xlabel('Time'),ylabel('State'),title(['Process' num2str(i)]);
end
% Calculating the total overhead
rank = (0:Num_Process-1)';
p_overhead = [];
for i=0:Num_Process-1
   rows  = (MPI[:,['rank']] == i) & (MPI[:,['signal']] == 2 | MPI[:,['signal']] == 3);
   start_margin = MPI[rows,['time']];
   rows  = (MPI[:,['rank']] == i) & (MPI[:,['signal']] == 4);
   stop_margin = MPI[rows,['time']];
   p_overhead = [p_overhead ; sum(stop_margin - start_margin)];

end
overhead_table = table(rank,p_overhead)
figure(2);
bar(rank,p_overhead),xlabel('Process ID'),ylabel('Overhead Time'),title('Overhead Time for each Process');
total_overhead = sum(p_overhead);
Seq_time = Num_Process .* wall_clock_time - total_overhead;
% Calculating the speedup and efficency
Speedup = Seq_time/wall_clock_time
efficiency = Seq_time ./ (Seq_time + total_overhead)
Report_Table = table(Num_Files,Num_Process,Seq_time,wall_clock_time,total_overhead,Speedup,efficiency)
```

Figure 4: MATLAB script for profiling

6. Experimental results and Evaluation

6.1 Experimental conditions

6.1.1 Practical parallel environment

In the implementation, a Java programming language is used. Since native MPI implementation hasn't supported Java Parallel programming, Java binding technique is integrated into Open MPI implementation project.

The Open MPI standard should be downloaded and compiled the source package. The Netbeans Java IDE is used for writing Java code for the parallel and sequential programs.

Converting into JAR file after compiling the source code. Using the Open MPI standard's capabilities of tools and programs to compile and run the parallel document classification program, such as mpirun command. The JAR file is run by the mpirun in the command line in the Ubuntu OS as follows:

/openmpi/bin/mpirun -np 4 –allow-run-as-root java - jar ./dist/ParDocClassify.jar ./data usa.txt output.csv > MPI.csv

The JAR file accepts three arguments: *./data* is the path of the directory containing the textual documents. *usa.txt* is the filename of the dictionary file, and *output.csv* is the output filename containing the results of document vectors.

The result of execution is redirected into CSV file such as *MPI.csv*.

6.1.2 Metrics

In order to determine the best algorithm, evaluation hardware platforms, and examining the benefits from parallelism, a number of metrics have been used on the desired outcome of performance analysis (see [3]).

In the experiment the following performance metrics are used as follows:

- **Wall-clock Time:** The parallel runtime is the time that elapses from the moment of a parallel computation starts to the moment the last processing element finishes execution. The serial and parallel runtime are denoted T_s and T_p respectively.

- **Total parallel overhead:** The total overhead of a parallel system as the total time collectively spent by all processing elements (pT_p) over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element (T_s). The overhead function (T_o) is given by Eq. (1):

$$T_o = pT_p - T_s \qquad (1)$$

- **Speedup:** Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with identical processing elements. Speedup is denoted by the symbol *S*, and it is calculated by Eq. (2).

$$S = \frac{T_s}{T_p} \qquad (2)$$

- **Efficiency:** Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements (p). In an ideal parallel system, speedup is equal to *p* and efficiency is equal to one. In practice, speedup is less than *p* and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. Efficiency is denoted by the symbol *E*. Mathematically, it is given by Eq. (3).

$$E = \frac{S}{p} = \frac{T_s}{pT_p} = \frac{1}{1 + \frac{T_o}{T_s}} \qquad (3)$$

6.2 Performance results

Sample results are reported in Table 2.

Table 2: Results of the Sequential and Parallel document classification

NFile	NProcess	Seqtime	WallCTime	Overhead	Speedup	Efficiency
534	4	3755	1720	3125	2.1831	0.5458

6.2.1 Speedup and efficiency evaluation

The parallel program is run on different processor chain to compute the speedup. The number of text files is maintained and remains fixed, while the number of processing elements increases. The profiled log file is analyzed and the performance parameters are recorded for each case, as they are described in Table 3.

The graph is plotted as shown in Figure 5.

Table 3: Results of performance evaluation

No.Files	No.Processors	Speedup	Efficiency
534	2	0.9672	0.4836
534	4	2.1831	0.5458
534	6	4.3546	0.7257
534	8	6.7848	0.8481

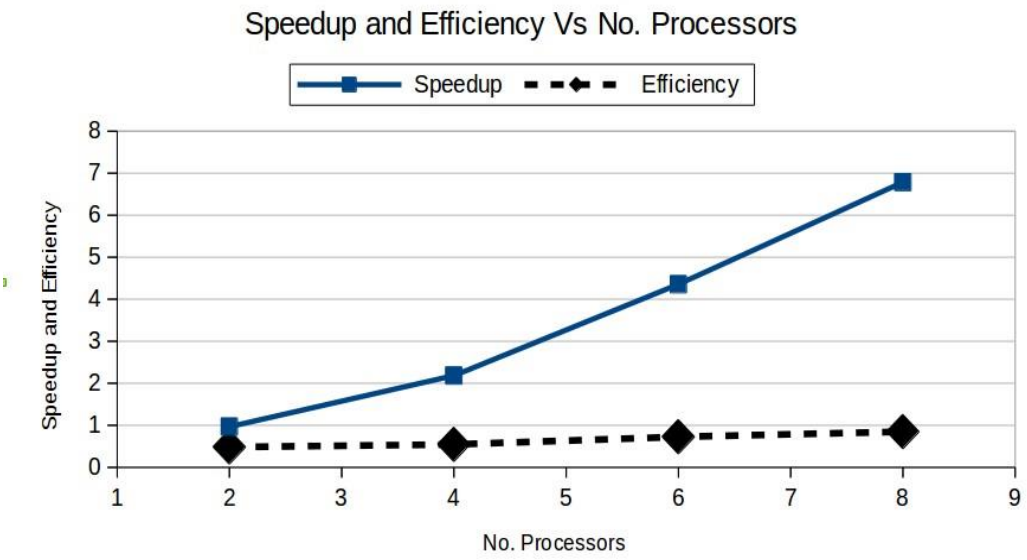

Figure 5: Speedup and Efficiency evaluation

6.2.2 Problem Size and Performance Evaluation

In this case, the performance of the parallel program is examined at different file pooling number. The examination takes place on 8 processors, while speedup and efficiency are recorded for a given number of files.

Table 4 shows the readings:

Table 4: Results of performance evaluation vs. the increasing number of files

No.Files	No.Processors	Speedup	Efficiency
100	8	6.7014	0.8377
200	8	6.7823	0.8478
300	8	6.7875	0.8484
400	8	6.7721	0.8465
500	8	6.7829	0.8479
600	8	6.8219	0.8527
700	8	6.7811	0.8476
800	8	6.7865	0.8483
900	8	6.796	0.8495
1000	8	6.8392	0.8549

6.3 Profiling Results

In the experiment, the proposed profiler program reports the performance results for the parallel execution of document classification algorithm, with running four processors handling classification of 534 text files. Here the profiler reports the following performance parameters:

- The serial runtime: T_s= 3755
- The parallel runtime: T_p= 1720
- The total parallel overhead: T_o = 3125
- The speedup: S = 2.1831
- The efficiency: E = 0.5458

In the profiler, there are graphical representations showing the performance results for each process and computation/communication scenarios. For example, Figure 6 shows the overhead representation for each of the four processors.

As shown in Figure 6, the master process has a long overhead time waiting for the workers to finish their assigned tasks. The situation is clearly shown on the scenario in Figure 7.

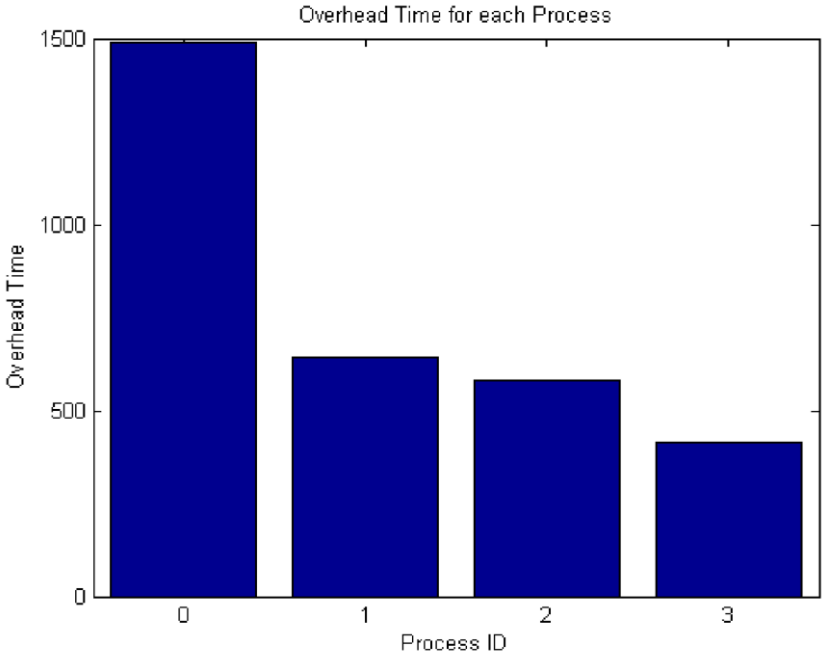

Figure 6: Overheads of parallel document classification

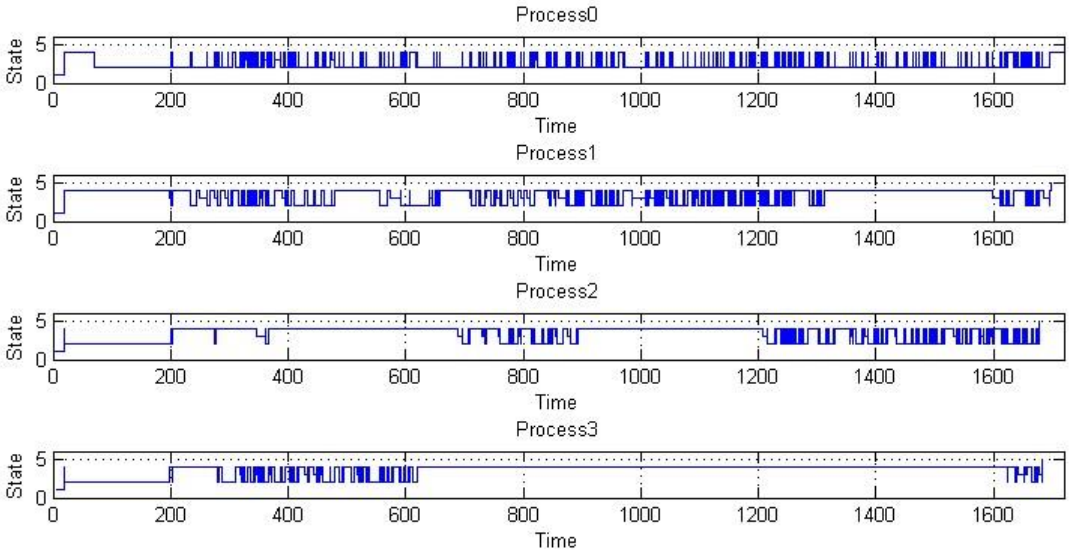

Figure 7: Profiling of computation-communication scenario

In Figure 7, there is a representation of the computation/communication scenario that illustrates the profiling of the five-state transition in each process of life-cycle. The performance evaluation of each process is obviously illustrated.

For comparing the performance of parallelism, each process has different amount of computation time (Tcomp) and communication time (Tcomm) as shown in Figure 7. The following is the overheads parallelism summary:

- P0: very long $T_{comm}$is due to highly passing of messages in the intra-communication environment with implementing the Manager/Worker-style in which the Manager distributes tasks among Workers that send the results info back to the manager.
- P1: long $T_{comm}$is due to processing of small-size files
- P2: T_{comp}= $T_{comm}$is due to the medium-size of files.
- P3: long $T_{comp}$is due to processing of big-size files.

Conclusion

Manager/Worker style is an efficient dynamic task allocation for parallel document classification programming since the size of the text file may be different and, furthermore, the type of html or xml document makes the operation more complicated than ordinary text files. This mechanism enhances the efficiency of the algorithm which operates around 70% − 80 %.

In the experimental results, a good scalability is shown as the gain in processing power increases linearly, when increasing the number of processing elements. Other obtained results show that there is no relation between the number of files getting classified and the performance of the system. Throughout the readings obtained from the profiling results, computation/communication scenario shows that the manager has a long overhead time waiting for tasks to finish and it has to be invested in that time specially for big text files. Enhancement procedures may take place such as sending group of files in one task rather than one only.

References

1. Bahudaila, S. A., & Haider, A. S. (2016). ”Performance Estimation of Parallel Face Detection Algorithm on Multi-Core Platforms”, Egyptian Computer Science Journal, Vol.40 No.2, pp. 65-76.
2. Bahudaila, S. A. (2007). ”A Comparison Study of OpenMP Loop Scheduling Methods on a Platform with Hyper-Threading Technology”, Journal of Computer Engineering, Vol.1, No.2, pp. 67-74.
3. Grama, A., & Gupta, A. (2003). ”Introduction to Parallel Computing”, 2nd Edition, (c) Pearson, pp. 195-226.
4. Huang, L., & Li, Z. (2013), ”A Novel Method of Parallel GPU Implementation of KNN Used in Text Classification”, 4th International Conference on Networking and Distributed Computing, pp. 6-8.
5. Polpinij, J., Srikanjanapert, N., Sopon, P. (2017). ”Word2Vec Approach for Sentiment Classification Relating to Hotel Reviews”, Recent Advances in Information and Communication Technology, pp. 308-316.
6. Quinn, M. (2004). ”Parallel Programming in C with MPI and OpenMP”, (c) McGraw-Hill, pp. 216-235.
7. Vega-Gisbar, O., & Roman, J. E. (2016). ”Design and Implementation of Java bindings in Open MPI”, © Journal of Parallel Computing, pp.1-20.
8. Vyawahare, N. R., & Lade, S. G. (2014). ”Document Classification using Parallel Processing”, International Journal of Engineering and Technology (IJERT), Vol.3 Issue 2, pp. 2665-2668.
9. Wilkinson, B., Allen, M. (2005). ”Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers,” 2nd Edition, Pearson Prentice Hall, pp. 204-210.
10. www.open-mpi.org

التصنيف الوثائقي في البيئات المتوازية عن طريق ربط جافا في واجهة مرور رسائل مفتوحة المصدر

صبحي عبدالرحيم باهذيلة و وضاح أحمد منصر
قسم تكنولوجيا المعلومات، كلية الهندسة، جامعة عدن
البريد الالكتروني: sabhudail@yahoo.com ، wmunassar@yahoo.com
DOI: https://doi.org/10.47372/uajnas.2017.n2.a09

الملخص

تصف هذه الورقة الحوسبة عالية الأداء محرك بحث الوثيقة التي تتجه كل وثيقة سرية. ويتحقق التوازي عن طريق استغلال بيئات الحوسبة الموازية والموزعة للعمليات المتعددة الجماعية التي يتم تنفيذها باستخدام ارتباطات واجهة مرور رسالة جافا مفتوحة المصدر. يتم تطبيق نموذج متوازية من مدير/ عامل للحصول على موازنة التحميل. وكذلك التحليل ويتم تحقيق القياس في نموذجنا لتشخيص التوازي وهذا مصمم للتنفيذ. تظهر اثنين من النتائج المخراجات التجريبية:أداء المعالجة المتوازية مع كفاءة 80٪، ونتائج التشخيص التي توضح اماكن الاستخدامات والنفقات العامة في نموذجنا التوازي.

الكلمات المفتاحية:ربط جافا بواجهة مرور رسائل، موازنة التحميل، النمط العلمي مدير/ عامل النمط، تشخيص التوازي، تصنيف المتجهات.