P-1611

PARALLEL PARSING ALGORITHM FOR STATIC DICTIONARY COMPRESSION

A PROJECT PEPORT

Submitted By

DIVYA.R	**71202104008**
SUDHA.N	**71202104044**

in partial fulfillment for the award of the degree
of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

ANNA UNIVERSITY: CHENNAI 600025

APRIL 2006

EVALUATION CERTIFICATE

ANNA UNIVERSITY:CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report **"PARALLEL PARSING ALGORITHM FOR STATIC DICTIONARY COMPRESSION"** is the bonafide work of **"R.DIVYA(71202104008), N.SUDHA (71202104044)"**, who carried out the project work under my supervision.

SIGNATURE	SIGNATURE
Dr.S.Thangasamy	Ms.S.Rajini
DEAN OF THE DEPARTMENT	SUPERVISOR
	Senior Lecturer
Dept.of Comp.Sci&Engg.,	Dept.of Comp.Sci&Engg.
Kumaraguru College Of Tech.,	Kumaraguru College of Tech.,
Chinnavedampatti P.O.,	Chinnavedampatti P.O.,
Coimbatore-641006.	Coimbatore-641006.

Submitted for the Viva-voce Examination held on 2/5/06

Internal Examiner

External Examiner

EVALUATION CERTIFICATE

College : **KUMARAGURU COLLEGE OF TECHNOLOGY**

Branch : **COMPUTER SCIENCE AND ENGINEERING**

Semester : **EIGHTH SEMESTER (08)**

Name of the Students	Title of the Project	Name of the Supervisor
R.Divya N.Sudha	**Parallel Parsing Algorithm for Static Dictionary Compression**	Ms.S.Rajini,B.E., Senior Lecturer.

The report of the project work submitted by the above students in partial fulfillment for the award of BACHELOR OF ENGINEERING degree in COMPUTER SCIENCE AND ENGINEERING of Anna University were evaluated and confirmed to be the report of the work done by the above students and then evaluated.

Internal Examiner

External Examiner

DECLARATION

DECLARATION

We hereby declare that the project entitled "**PARALLEL PARSING ALGORITHM FOR STATIC DICTIONARY COMPRESSION**", is a record of original work done by us and to the best of our knowledge, a similar work has not been submitted to Anna University or any other institution, for fulfillment of the requirement of the course study.

This report is submitted in partial fulfillment of the requirements for the award of the Degree of Bachelor of Computer Science and Engineering of Anna University, Chennai.

Place: Coimbatore
Date :

(DIVYA.R)

(SUDHA.N)

ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

We would like to thank our principal, **Dr.K.K.Padmanabhan, B.Sc(Engg), M.Tech., Ph.D,** for the facilities provided to carry out this project.

We are very much indebted to **Dr.S.Thangasamy, B.E.(Hons), Ph.D** Dean of the Department of Computer Science and Engineering, for his support towards us and for his gracious permission to undertake this project.

We deem it a pleasure and privilege to feel indebted to our guide **Ms.S.Rajini, B.E.,** Senior Lecturer, who is always a constant source of inspiration and encouragement that helped us in the successful completion of the project.

We also thank our project coordinator, **Mrs.S.Devaki, M.S.,** Assistant Professor, Department of Computer Science and Engineering, for her support during the course of our project.

We extend our special thanks to our **family, friends and staff members** for all their support and help rendered to us.

ABSTRACT

ABSTRACT

The project entitled **'PARALLEL PARSING ALGORITHM FOR STATIC DICTIONARY COMPRESSION'** based on compressing the given input data in parallel using the dictionary entries.

The data compression based on dictionary techniques works by replacing phrases in the input string with indexes into some dictionary. The dictionary can be static or dynamic. In static dictionary compression, the dictionary contains a predetermined fixed set of entries.

We implemented parallel algorithms for the optimal parsing strategy with dictionaries by modifying the on-line algorithm using a pointer doubling technique which gives better result.

For this, we used Java Parallel Virtual Machine (JPVM) to connect the systems and to do the compression in parallel. JPVM is an interface used to connect the systems and is implemented in java.

The dictionary compression is done in parallel to increase the speed of data compression so that the data compression can be achieved with reduced time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

ABBREVIATIONS

LIST OF ABBREVIATIONS

1.LFF : Longest Fragment First strategy

2.JPVM : Java Parallel Virtual Machine

3.PVM : Parallel Virtual Machine

4.MPI : Message Passing Interface

5.JDK : Java Development Kit

INTRODUCTION

1.INTRODUCTION

1.1 OVERVIEW:

In this project we implemented parallel algorithm for the parsing strategy for static dictionary compression:

The optimal parsing strategy

The increasing speed of modern data processing systems, such as disk controllers and data transmission systems requires the data compression to be accomplished at a very high speed. Parallelism can help by increasing speed and compression effectiveness.

Parallel compression is therefore adopted in many applications for which sequential compression does not meet the performance criteria.

The data compression based on dictionary techniques is popular for desk-top compression utilities and fast hardware data compression systems. It works by replacing groups of consecutive symbols (phrases) in the input string with the corresponding dictionary indexes. The dictionary can be static or dynamic.

In static dictionary compression, the dictionary contains a predetermined fixed set of entries. The central part of dictionary techniques is the process of parsing the input string into phrases that are to be replaced with the corresponding dictionary indices.

In dynamic dictionary compression, the dictionary changes its entries during compression, reflecting newly encoded phrases in the input string.

There are different strategies for parsing. Many data compression programs are based on dictionary techniques used today, ex: UNIX compress, employ a greedy parsing strategy.

In the greedy parsing strategy, the input string is processed from left to right and, at each step, it is encoded as the longest phrase starting at the first symbol of the uncoded portion of the string that matches a dictionary entry and replaced by the corresponding index.

The greedy parsing strategy is simple and efficient in sequential environment, but it has been shown that non greedy parsing strategies can improve the performance of dictionary techniques. There are two non greedy parsing strategies.

1. The optimal parsing strategy with dictionaries
2. The Longest Fragment First strategy(LFF)

An optimal parsing of the input string is a short possible sequence of dictionary indices such that the concatenation of the corresponding dictionary entries forms the input string. If fixed length indices are used, an optimal parsing produces the minimum number of dictionary indices.

In general, a sequential optimal parsing encoder must be able to look at arbitrarily large prefixes of the input string.

However, if the dictionary has the property called prefix property, optimal parsing can be computed online, or in a one-pass left-to-right scan with limited look ahead.

A dictionary has the prefix property if all the prefixes of the phrase in the dictionary are also in the dictionary. We call a dictionary with this property a prefix property.

The key to our parallel algorithm is that we derive them by modifying the on-line algorithms using a pointer doubling technique. This derivation of parallel algorithms was used by Stauffer and Hirschberg for the greedy parsing algorithm and can be applied to other on-line algorithms.

Thus we used this on-line optimal parsing strategy with dictionaries to derive the parallel algorithm and we derived the algorithm and then implemented it by connecting four processors using JPVM.

The JPVM library is a software system for explicit message-passing based distributed memory MIMD parallel programming in Java. The data is divided among these four processors and sent to the processors. Then the calculation is done in parallel among these four processors and the result is written to the output file.

Thus the parallel parsing algorithm for static dictionary compression reduces the processing time for the compression.

1.2 EXISTING SYSTEM AND LIMITATIONS:

The limits of the processing power of single processor machines have already been reached. Parallel processing provides a very good solution for this.

- The speed of modern data processing systems are increased such as disk controllers and data transmission systems.
- This requires the data compression to be accomplished at a very high speed.
- But the existing sequential compression techniques requires more time for compression.

Existing PVM also does not support the features, such as thread safety, multiple communication end-points per task, and default-case direct message routing.

1.3 PROPOSED SYSTEM AND ADVANTAGES:

The speed of the data compression can be increased by introducing parallelism for the parsing technique in compression algorithm.

We implemented parallel parsing algorithm for static dictionary compression using optimal parsing strategy to reduce the time for compression.

We improved the runtime of the optimal parsing algorithm by using parallel parsing, so that the data compression using static dictionary can be done with high speed.

We used JPVM to connect the systems in parallel and to perform parallel processing. The JPVM library is a software system for explicit message-passing based distributed memory MIMD parallel programming in Java.

The advantages of JPVM include the following,

- Add/delete hosts from a virtual machine
- Spawn/kill tasks dynamically
- Message passing
- Blocking send, blocking and non blocking receive
- Dynamic tasks group
- Fault tolerance
- Process control
- Thread safety
- Multiple communication end point per task
- Default case directed message routing
- Portability

SYSTEM DESIGN

2. SYSTEM DESIGN

2.1 OVERVIEW:

Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for good design.

Design is the process through which the requirements are translated into blue print for constructing the software.

The design must have some characteristics:

1. The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all the implicit requirements.
2. The design must be a readable and understandable guide for those who generate code and for those who test & subsequently support the software.

The design should provide a complete picture of the software, functional and behavioral domains from an implementation perspective.

2.2 MODULES:

Module.1 (Dictionary interface):

In this module first we created the dictionary. The dictionary should be static and prefixed.

Static:

If the dictionary contains a predetermined fixed set of entries, then it is called as a static dictionary.

Prefixed dictionary:

A dictionary has the prefix property if all the prefixes of the phrases in the dictionary are also in the dictionary.

Once the dictionary is created we can use this dictionary for all. Next we have to read the dictionary entries and store those that in data structure and also indices are found.

Module.2 (Reading data):

Data is the given input file which has to be compressed. In this module we read the data and store them in corresponding data structure. They can be used later to find out the longest match length and to replace with the corresponding indices.

Module.3 (longest match):

This module is used to find out the longest match length at a given position. It takes the input as

- The data which has to be compressed
- The position for which we have to find the longest match
- The dictionary

and find out the longest match as the output and returns the same to the function which calls it.

Module.4 (next break point):

This module is used to find out the next break point from the given position. This can be obtained by using the comparison of *c[k]* values where *k* is between *i+1* and *c[i]* and *i* is the current breakpoint. Thus it produces the next break point after *i* as the current breakpoint and returns that position to the function which calls it.

Optimal parsing algorithm:

This is the main module for the dictionary compression which makes use of the above modules sequentially, so that this is used for sequential dictionary compression. The dictionary compression time for this sequential optimal parsing algorithm is more.

Parallel optimal parsing algorithm:

This is the main module for the parallel dictionary compression which makes use of the above four modules for computing the values in parallel. This module is used for parallel parsing in static dictionary compression. For this we connected four systems and in one system we divided the process into those four systems. Then the system computes the values and the results are sent to the main module. For this we have master module and slave module.

In master module, we have to split up the process into four and the corresponding data packed into buffer and sent to all the systems that are connected in parallel. When the output is received from the slave processes we have to unpack those data and then write to the output file.

In slave module we have to get the data from the master module and it has to be unpacked. Then the actual process is carried out in the slave systems in parallel. After the completion of processing, the results should be packed and returned to the master.

WORKING ENVIRONMENT

3. WORKING ENVIRONMENT

The development environment gives the minimum hardware and software requirements

3.1 HARDWARE SPECIFICATION

Processor	:Pentium III
RAM	:64MB
Cache	:128KB
Hard disk	:10GB
Floppy drive	:1.44FDD
Monitor	:14"Monitor

3.2 SOFTWARE SPECIFICATION

Operating System	: Windows XP
Language	: Java
Software Package	: JPVM

4.DETAILED DESIGN

4.1 ONLINE OPTIMAL PARSING ALGORITHM

The greedy parsing strategy is simple and efficient in sequential environment, but it has been shown that non greedy parsing strategies can improve the performance of dictionary techniques. There are two non greedy parsing strategies.

1.The optimal parsing strategy with dictionaries

2.The Longest Fragment First strategy(LFF)

An optimal parsing of the input string is the shortest possible sequence of dictionary indices such that the concatenation of the corresponding dictionary entries forms the input string. If fixed length indices are used, an optimal parsing produces the minimum number of dictionary indices.

In general, a sequential optimal parsing encoder must be able to look at arbitrarily large prefixes of the input string. However, if the dictionary has the property called prefix property, optimal parsing can be computed online, or in a one-pass left-to-right scan with limited look-ahead.

A dictionary has the prefix property if all the prefixes of the phrase in the dictionary are also in the dictionary. We call a dictionary with this property a prefix property.

Algorithm for this on-line optimal parsing for static dictionary compression which explains the overall parsing of the compression is given in the figure 1.

Algorithm:

```
i:=1;
 m(1):=longest match length at position 1;
c(1):=1+m(1)
  while c(i)<=n do
          Begin
                for k:=i+1 to c(i) do
                begin
                    m(k):=longest match length at position k;
                    c(k):=k+ m(K)
                end
                determine k' that satisfies c(k')=max{c(k)|i+1<=k<=c(i)};
                encode phrase x_i.... x_k'-1;
                i:=k';
          end
  encode phrase x_i..... x_n;
```

Figure 1:On-line optimal parsing algorithm

In the on-line algorithm shown in fig. 1, variable *i* is used to keep track of the current break point, *m(i)* and it receives the length of the longest match at position *i* (*c(i)=i+m(i)*). At first we initialize *i* and find values for *m(1)* and *c(1)*.

In the next while loop, *i* advances through the input string from left to right. If *c(i)* is found to be greater than n, that means position *i* is the last break point in the input string, and the operation terminates after the last phrase $x_i...x_n$ is encoded.

In each iteration of the while loop, we encode one phrase at the current break point *i*, and advance *i* to the next break point. At the beginning of each iteration of the loop, we know the values of *i*, *m(i)*, and *c(i)*. In the internal for loop , we go through *m(i)* positions, from position *i+1* to the position just after the longest match at position *i*(position *i+m(i)* or *c(i)*).

We use variable *k* to indicate one of these positions. At each of these positions, we find the longest match length (*m(k)*) and the position just after the longest match (*c(k)=k+m(k)*).

After the for loop, we compare the *m(i)* values *c(i+1),c(i+2),....c(c(i))* and find the maximum among them. In effect, we determine the position *k'* at which the longest match has the right most ending point among the *m(i)* longest matches. This position *k*, will be the next break point.

Thus, we encode the phrase starting from the current break point *i* and ending at the position just before the next break point *k'-1*(x_i.... $x_{k'-1}$), and prepare for the next iteration by assigning *k'* to *i*. During the current iteration of the while loop, we find the longest match length at the break point and the position just after the longest match for the next iteration.

4.2 PARALLEL OPTIMAL PARSING ALGORITHM WITH PREFIX DICTIONARIES

Parallel optimal parsing algorithm with prefix dictionaries is derived from the on-line algorithm in fig.1. This parallel algorithm is used to do the static dictionary compression in parallel which reduces the running time of the process. This can be derived from the following way from the on-line algorithm depicted in fig.1.

Since we use the pointer doubling technique, we determine *next(i)* for each input position *i* in the parallel algorithm. We assign the different part of the input different processors working on the iteration of the while loop in the on-line algorithm in fig.1.

We divide our parallel optimal parsing algorithm with prefix dictionaries into three steps:

In step 1, we find the longest match length and the position just after the longest match at each input position in parallel. We trace down the dictionary (D) to find the longest match at each input position *i* in parallel and put the length of the longest match in *m(i)*. This process runs in O(L) time for each input position in different processors.

After that, we find the position just after the longest match at each input position *i* (*i+m(i)*) in parallel and put the value in *c(i)*.

In step 2, we determine *next(i)* for each input position *i*. At each input positon *i*, we compare *m(i)* values *c(i+1),c(i+2),...,c(c(i))* sequentially and find the maximum among them. If *c(k')* is the maximum, then *k'* is the *next(i)*. Since *m(i)<=L*, we compare at most L values for each input position in different processors.

In step 3, we determine all the true break points in the input string using a pointer doubling technique

The steps in the on-line optimal parallel parsing algorithm is given below briefly ,

- **Step 1:** To find the longest match length (*m(i)*) and the position just after the longest match at input position (*c(i)*) in parallel
- **Step 2:** Determine *next(i)* for each input position *'i'*. At each input position '*i*', we compare *m(i)* values with *c(i+1),c(i+2),....,c(c(i))*
- **Step 3:** Determine all the true breakpoints in the input string

Thus we derived parallel algorithm from the sequential on-line optimal parsing algorithm. Thus we can work with different input positions in different processors simultaneously which leads to parallelism and reduces the compression time.

Example of the parallel optimal parsing algorithm with prefixed dictionaries is given in fig.2.

Example

Dictionary entries :a, b, ba , bab, baa , baaa

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Input	**b**	**a**	**b**	**a**	**a**	**a**	**b**	**a**	**b**	**a**	**a**	
m(i)	**3**	**1**	**4**	**1**	**1**	**1**	**3**	**1**	**3**	**1**	**1**	
c(i)	**4**	**3**	**7**	**5**	**6**	**7**	**10**	**9**	**12**	**11**	**12**	
next(i)	**3**	**3**	**7**	**5**	**6**	**7**	**9**	**9**	**12**	**11**	**12**	
break points	●		●				●		●			

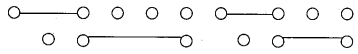

Figure: 2 Example for parallel optimal parsing algorithm

We have 11 symbols in the input string, and position 12 is the dummy input position.

A circle or two circles connected with a line show the longest match at each input position. The two rows with the marks *m(i)* and *c(i)* show the results of Step 1.

For example, at position 7, the longest match is "bab" with length 3 and the position just after this longest match is 10.

Thus,

m(7)=3 and

c(7)=10.

The row marked with next(i) in the figure shows the result of step 2. For example, at position 7, we compare three values c(8), c(9) and c(10) sequentially because m(7)=3.

The maximum among them is

c(9)=12,

at position 9.

Therefore, we determine that

next(7)=9.

The filled circles in the bottom row of the fig.2, show the true break points determined in step 3. Position 1 is always a true break point for all input data.

Positions 3, 7, and 9 are also determined to be the true break points because,

next(1)=3,

next(3)=7 and

next(7)=9.

Thus the given example for the parallel parsing algorithm shows the actual working of the algorithm

4.3 JPVM (JAVA PARALLEL VIRTUAL MACHINE)

4.3.1 OVERVIEW:

The JPVM library is a software system for explicit message-passing based distributed memory MIMD parallel programming in Java. The library supports an interface similar to the C and Fortran interface provided by the PVM system, but with syntax and semantics modifications afforded by Java and better matched to Java programming styles.

The similarity between JPVM and the widely used PVM system supports a quick learning curve for experienced PVM programmers, thus making the JPVM system an accessible, low-investment target for migrating parallel applications to the Java platform. At the same time, JPVM offers novel features not found in standard PVM such as thread safety, multiple communication end-points per task, and default-case direct message routing.

JPVM is implemented entirely in Java, and is thus highly portable among platforms supporting some version of the Java Virtual Machine. This feature opens up the possibility of utilizing resources commonly excluded from network parallel computing systems such as Macintosh and Windows-NT based systems. Initial applications performance results achieved with a prototype JPVM system indicate that the Java-implemented approach can offer good performance at appropriately coarse granularities.

4.3.2 INTRODUCTION

The use of heterogeneous collections of computing systems interconnected by one or more networks as a single logical computational resource has become a wide-spread approach to high-performance computing. Network parallel computing systems allow individual applications to harness the aggregate power of the increasingly powerful, well-networked, heterogeneous, and often largely under-utilized collections of resources available to many users.

In this project, we used a network parallel computing software system for use with and implemented in the Java language. Because of its mandated platform independence and uniform interface to system services, Java provides an attractive environment for the implementation of both network parallel applications and the system software needed to support them.

Although numerous software systems support some form of network parallel computing, the majority of use has thus far been based on a small set of popular packages that provide an explicit message-passing, distributed memory MIMD programming model such as PVM and the MPI .

These software systems support simple, portable library interfaces for typical high-performance computing languages such as C and Fortran. For example, PVM provides the programmer with library routines to perform task creation, data marshalling and asynchronous message passing.

In addition, PVM provides tools for specifying and managing a collection of hosts on which applications will execute. Results obtained with network parallel computing systems have been encouraging. For example, a performance study of the NAS benchmark suite implemented in PVM demonstrated that relatively small clusters of workstations could provide performance comparable to significantly more expensive supercomputers.

However, the utilization of distributed, heterogeneous, shared resources connected by commodity networks as a single, virtual parallel computer poses serious problems for both the application and system software programmer. For example, from the application perspective, it has been found that successful network parallel programs will almost always exhibit medium to coarse granularity and will be tolerant of network latency, For example through the use of a send ahead programming style. These attributes can be difficult to achieve within some applications. From the system programmer perspective, heterogeneity results in difficult problems such as task to platform matching and system portability.

The Java language provides a number of features that appear to be promising tools for addressing some of the inherent problems associated with network parallel programming. For example, from the application perspective, Java provides a portable, uniform interface to threads.

From the system implementation perspective, Java supports a high degree of code portability and a uniform API for operating system services such as network communications.

4.3.3 SETTING UP JPVM

Step 1 - Start up a *jpvm daemon* on each host we want to use:

o First, make sure that the *jpvm* distribution directory (i.e. the directory in which this README lives) is in the CLASSPATH environment variable.

o Next, run the daemon using the standard JDK interpreter. For example, on Unix , we run the command:

S java jpvm.jpvmDaemon

o On each host on which the daemon runs, it will print out a port number that identifies the daemon. For example, when we run the command above on the local machine, it prints out the following message:

jpvm daemon: pac-man.cs.Virginia.EDU, port #63981

At this point, the daemons are running, but are not communicating.

Step 2 - Connect the *jpvm daemons* using the *jpvm console*:

o First, we need to run a *jpvm console*. Pick any host that has a daemon running on it, and run the *jpvmConsole* class.

For example, in Unix we run:

S java jpvm.jpvmConsole

This should bring up the *jpvm console command prompt:* **"jpvm>".**

o The console command to connect a new host to the host on which we are running the console is "add". After we enter the command "add", the console will ask for the name of the host you wish to add and the port of the daemon on that host.

An example session might look like:

jpvm> add
Host name : augusta.cs.virginia.EDU
Port number : 40305
jpvm>

o Every time we add a host from the console, it becomes part of the Java Parallel Virtual Machine and can be used by *jpvm* programs. we can check the available host set from the *jpvm console* prompt using the "conf" command. For example, after the above session, conf would print out the following:

jpvm> conf
2 hosts:
pac-man.cs.Virginia.EDU
augusta.cs.Virginia.EDU
jpvm>

Step 3 - Run a JPVM program:

o After the JPVM net is set up, we can try out a *jpvm* program. There is a simple example program called "hello" . The hello command line task needs to spawn objects of the class "hello other". This means that the JPVM daemons will need to be able to find the file "hello_other.class". The simplest thing to do to ensure that they can find it is to copy it into the directory in which the daemons were started (i.e. their current working directory).

o After the "hello_other.class" file is visible to all daemons,we can run the hello example.

For example, on Unix we ran the command:

> java hello

The output is reproduced below:

Task Id: pac-man.cs.Virginia.EDU, port #64118
Worker tasks:
augusta.cs.Virginia.EDU, port #40327
pac-man.cs.Virginia.EDU, port #64121
augusta.cs.Virginia.EDU, port #40331
pac-man.cs.Virginia.EDU, port #64124
Got message tag 12345 from augusta.cs.Virginia.EDU, port #40327
Received: Hello from jpvm task, id: augusta.cs.Virginia.EDU, port #40327
Got message tag 12345 from pac-man.cs.Virginia.EDU, port #64121
Received: Hello from jpvm task, id: pac-man.cs.Virginia.EDU, port #64121
Got message tag 12345 from augusta.cs.Virginia.EDU, port #40331
Received: Hello from jpvm task, id: augusta.cs.Virginia.EDU, port #40331
Got message tag 12345 from pac-man.cs.Virginia.EDU, port #64124
Received: Hello from jpvm task, id: pac-man.cs.Virginia.EDU, port #64124

Figure 3:Output of the example program

3. Known Problems

- In some environments, the *jpvmDaemon* objects may not find the "java" executable that the need to start new tasks, even though "java" is in the PATH environment variable when the daemon is started. The symptom of

this problem is the inability to spawn any new tasks (i.e. the demo freezes). The quick fix for this problem is to set the "jpvm_exec" member variable of the *jpvmDaemon* class in the file *jpvmDaemon.java* in the *jpvm* directory to be the full path for our local "java" executable. On Unix systems, this path can be found out by running "which java". After this change is made to the daemon code, the daemon needs to be recompiled by running the command *"javac jpvmDaemon.java"* in the *jpvm* directory. Any running daemons will need to be restarted for the fix to take effect.

4.3.4 INTERFACE

The programming interface provided by the JPVM system is intentionally similar to that supported by the PVM system, with the addition of enhancements to better exploit the potential benefits of Java as a language for implementing network parallel applications.

As in PVM, the programmer decomposes the problem to be solved into a set of cooperating sequential task implementations. These sequential tasks execute on a collection of available processors and invoke special library routines to control the creation of additional tasks and to pass messages among tasks. In JPVM, task implementations are coded in Java, and support for task creation and message passing is provided by the JPVM library.

The central interface through which most JPVM interaction takes place is exported by the `jpvmEnvironment` Java class. Instances of this class are declared by JPVM tasks to connect to and interact with the JPVM system and other tasks executing within the system.

Objects of this class represent communications end-points within the system, and are identified by system-wide unique identifiers of the opaque type *jpvmTaskId* (analogous to a PVM task identifier). Whereas standard PVM restricts each task to having a single communications end-point (and correspondingly a single task identifier), JPVM allows tasks to maintain a logically unlimited number of communication connections simply by allocating multiple instances of *jpvmEnvironment*.

The ability to contain multiple communication end-points in a single task simplifies the process of developing separate linkable modules that need to perform communication. First-class separation of communication streams eliminates the need to artificially distinguish between messages intended for different modules.

After an instance of *jpvmEnvironment* is allocated, its containing task can invoke basic JPVM services such as task creation and message passing. For example, a task can determine its identity for JPVM communication by invoking the *pvm_mytid()* method. A task can detach a *jpvmEnvironment* from the JPVM system by invoking the `pvm_exit()` method.

4.3.5 TASK CREATION

The first action performed by a typical JPVM program is the creation of additional tasks to achieve parallel execution. Task creation in JPVM is supported by the *pvm_spawn()* method, which takes a string parameter indicating the name of a valid Java class visible in the **CLASSPATH** environment variable, as well as the number of tasks to spawn and an array

into which the *jpvmTaskIds* of the newly created tasks will be placed on successful return.

Each task created through *pvm_spawn()* executes in its own instance of the Java Virtual Machine, avoiding issues such as conflicting usage of system services among tasks. These newly created instances of the Java Virtual Machine are placed throughout the set of processors available to JPVM, and each runs an object of the specified Java class.
Classes in JPVM:

Constructor registers task with the JPVM system
public jpvmEnvironment();
Task creation:
public int pvm_spawn(String task_name, int num, jpvmTaskId tids[]);
Send messages:
public void pvm_send(jpvmBuffer buf, jpvmTaskId tid, int tag);
Receive messages, blocking (non-blocking versions not depicted):
public jpvmMessage pvm_recv(jpvmTaskId tid, int tag);
System configuration and control:
public jpvmConfiguration pvm_config();
Figure 4: Different classes in JPVM

The identity of newly spawned tasks gives rise to a basic problem for JPVM. In PVM, the identity of a task is clear since tasks have a single communication end-point and thus a single task identifier.

In JPVM, tasks can have any number of communication end-points, and thus it is unclear what identity for a newly spawned task should be returned to the parent (i.e. the spawning task). One simple solution would be to remove the return of task identifiers from the spawn interface. Spawned tasks could communicate their identities to their parents explicitly to enable communication.

In order to retain an interface similar to PVM, and to avoid additional application code in the common case of a single identity per task, JPVM instead addresses this issue by returning the identity of the first *jpvmEnvironment* allocated in each newly spawned task. For standard, single-identity tasks, this provides the familiar PVM style of identifying newly spawned tasks.

4.3.6 MESSAGE PASSING

Message passing in JPVM is performed using the *pvm_send()* and *pvm_recv()* methods of the *jpvmEnvironment* class. However, before data can be sent, it must be collected into a *jpvm-Buffer* object. Analogous to PVM buffers, *jpvmBuffer* objects are the message content containers of JPVM. The *jpvmBuffer* interface contains two basic groups of methods: those to pack data into a buffer, and those to extract data from a buffer. Where possible, overloading is used in the interface to simplify application code.

Scalar and vector pack and unpack operations are provided for all basic Java types as well as *String* and *jpvmTaskId* objects. An important difference between JPVM buffers and standard PVM buffers is the explicit nature of the JPVM buffer data structure.

public jpvmBuffer();
public void pack(int v[], int n, int stride);
...
public void unpack(int v[], int n, int stride);

Pack and unpack operations manipulate the buffer object on which they are invoked, leading to a simplified interface for dealing with threaded tasks. This is one example of JPVM's changes to the standard PVM interface to better support common Java programming styles such as the use of threads.

After the contents of a message have been marshalled into a *jpvmBuffer* object, the buffer can be sent to any task in the JPVM system using the *pvm_send()* method of the `jpvmEnvironment` class. Besides taking the buffer to be sent, *pvm_send()* also requires the identity of the task to which to the message should be delivered (in the form of a *jpvmTaskId*) and an integer identification number for the message called the message tag.

The send operation is asynchronous—the sending task proceeds immediately after the send is initiated, regardless of when (or if) the message is received.

To receive messages, tasks must execute the *pvm_recv()* method of the *jpvmEnvironment* class. Using the various versions of *pvm_recv()* depicted in Figure 4, tasks can request messages based on the identity of the message sender, the identification number (tag) of the message, both of these, or neither (i.e. receive any message). Receive operations block until a message of the requested type is available, at which point a *jpvmMessage* structure is returned containing the received message.

4.3.7 SYSTEM CONFIGURATION

The JPVM library routines require run-time support during execution in the form of a set of JPVM daemon processes running on the available collection of processors. These daemon processes are required primarily to support task creation (unlike in standard PVM, JPVM daemon processes do not perform message routing). JPVM system configuration is a two-phase process. First, *daemon* processes must be started manually on all hosts of interest.

The daemon program is provided in the form of a java class *(jpvmDaemon)*, so daemon startup simply involves running Java Virtual Machine processes to execute the daemon class instances. After the daemons are started, they must be notified of one another's existence. This is accomplished through an interactive JPVM console program.

Again, the console program is a java class (*jpvmConsole*) and should be started in a Java Virtual Machine process. Besides adding hosts, the JPVM console can be used to list the hosts available to the system and

JPVM tasks running in the system. An example *console* session is depicted below.

```
$ java jpvm.jpvmConsole
jpvm> conf
4 hosts:
stonesoup00.cs.virginia.edu
stonesoup01.cs.virginia.edu
stonesoup02.cs.virginia.edu
stonesoup03.cs.virginia.edu

jpvm> ps
stonesoup00.cs.virginia.edu, 2 tasks:
(command line jpvm task)
jpvm console
stonesoup01.cs.virginia.edu, 1 tasks:
mat_mult
stonesoup02.cs.virginia.edu, 1 tasks:
mat_mult
stonesoup03.cs.virginia.edu, 1 tasks:
mat_mult
jpvm>
```

Figure 5:Example console session

SYSTEM TESTING

5.SYSTEM TESTING

5.1 INTRODUCTION

Testing is an activity to verify that a correct system is being built and is performed with the intent of finding faults in the system. Testing is an activity, however not restricted to being performed after the development phase is complete.

But this is to be carried out in parallel with all stages of the system development, starting with requirement specification. Testing results once gathered and evaluated provide a qualitative indication of software quality and reliability and serve as a basis for design modification if required.

System testing is a process of checking whether the developed system is working according to the original objectives and requirements. The system should be tested experimentally with test data so as to ensure the system works according to the required specification. When the system is found working, we test it with actual data and check the performance.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding. The increasing visibility of software as a system element and the attendant "cost" associated with a software failure are the motivation forces for a well planned, thorough testing.

5.2 TESTING OBJECTIVES

The testing objectives are summarized in the following three steps. Testing is the process of executing a program with the intent of finding an error. A good test case is one that has high probability of finding an error. A successful test is one that uncovers as-yet-undiscovered errors.

5.3 TESTING PRINCIPLES

All tests should be traceable to customer requirements. Tests should be planned long before testing begins, that is, the test planning can begin "in the small" and progress towards resting "in large".

The focus of testing will shift progressively from programs to individual modules and finally to the entire project. Exhaustively testing is not possible. To be more effective, testing should be one, which has highest probability of finding errors.

The following are the attributes of good tests:

- A good test has a high probability of finding errors.
- A good test is not redundant
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

6. RESULTS AND COMPARISONS

In this project we improved the runtime of the static dictionary compression techniques by using the parallel parsing algorithm for optimal parsing strategy.

Comparison of the execution time for the sequential and parallel dictionary compression algorithms is given in the following chart,

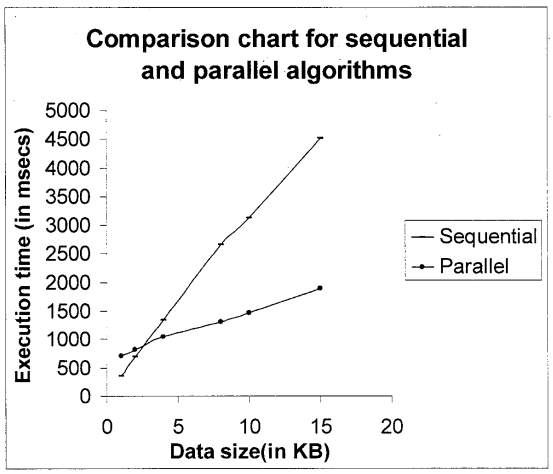

Figure 6: Comparison chart

7. CONCLUSION

The increasing speed of modern data processing systems, such as disk controllers and data transmission systems, requires the data compression to be accomplished at a very high speed. Parallelism can help with increasing speed and compression effectiveness.

To conclude, in this project we improved the runtime of the static dictionary compression techniques by using the parallel parsing algorithm for optimal parsing strategy.

We used JPVM to connect the system in parallel and to do the parallel processing which has good features to support parallel processing with multiple systems.

8. FUTURE ENHANCEMENTS

Due to the short duration that we had with us for the completion of this project, there have been some features that could not be implemented. These features are essentially enhancements to the proposed parallel processing model.

This parallel processing is only implemented for optimal parsing strategy for static dictionary compression.

This parallel parsing technique can be introduced for other static dictionary compression algorithms as well as dynamic dictionary compression algorithms.

This parallel parsing technique can also be introduced for other image compression and video compression methods.

We intend to carry out these enhancements whenever we find time to concentrate once again on parallel processing,

9. REFERENCES

- H.nagumo, M.Lu and K.Waston "Parallel parsing algorithms for static dictionary compression", *IEEE transactions on parallel and distributed systems*, vol. 10, pp. 1241 to 1251, December 1999.
- Patrick Naughton and Herbert Schildt, *JAVA2- The Complete Reference*, 3rd ed., Tata Mcgraw-Hill,1999
- Patrick Naughton ,*The JAVA Hand Book* , 2nd ed., Tata Mcgraw-Hill, 2001
- E.Balagurusamy ,*Programming with java*, 2nd ed., Tata Mcgraw-Hill, 2004

Websites:

- www.cs.virginia.edu/JPVM/
- www.ieee.org
- www.comp.parallel.pvm
- www.google.com

APPENDICES

10. APPENDICES

10.1 SAMPLE CODE

PARALLEL PARSING:

```
//main module
public static void main(String args[])
{
    start = msecond();
    System.out.print("Compression going on...");
    try
    {
        System.out.println("i am in main");
        jpvm = new jpvmEnvironment();
        myTaskId = jpvm.pvm_mytid();
        masterTaskId = jpvm.pvm_parent();
        //if it is parent then call master
        if(masterTaskId==jpvm.PvmNoParent)
            master();
        //otherwise call slave process
        else
            slave();
        System.out.print("i am in the end");
        end=msecond();
        //to find out the execution time
        System.out.println("\nTotal time: "+(end-
        start)+"msecs");
        //exit from the jpvm
        jpvm.pvm_exit();
```



```
    }
    catch (jpvmException jpe)
    {
        System.out.println("Error - jpvm exception");
    }
}

//dictionary interface module
//to read the dictionary entries
    public static void readdict()
    {
        String tmp="";
        //to assign for enter
        words[1]+=(char)13;
        //to assign for space
        words[2]+=(char)32;
        int i=1;
        try
        {
            FileInputStream ins;
            ins=new FileInputStream("C:\\jpvm\\dict.txt");
            int ch;
            while((ch=ins.read())!=-1)
            {
                //System.out.print("reading dictionary:");
                //if it is a space put one word
                if(ch==32)
```

```
                {
                        tmp=tmp.toLowerCase();
                        words[nwords]=tmp;
                        d[i]=(char)ch;
                        i++;
                        //increase the number of words
                        nwords++;
                        tmp="";
                        while((ch=ins.read())!=13);
                                ins.read();
                }
                else
                {       d[i]=(char)ch;
                        tmp+=(char)ch;
                        i=i+1;
        }}
        mld=i-1;
        for(i=1;i<=nwords;i++)
                words[i]=words[i].trim();
        ins.close();
        readdata();
    }
    catch(IOException e)
    {   System.out.println(e);
        System.exit(-1);
    }
}
```



```
    //to read data
    public static void readdata()
    {
        int ch;
        int i,v;
        int f;
        int k,p=0,d,flag;
        String tmp="";
        try
        {
            FileInputStream dat;
            dat=new FileInputStream("E:\\jpvm\\data4.txt");
            i=1;
            System.out.print("Reading data:");
            while((ch=dat.read())!=46)
            {
                if(ch==32)
                {
                    tmp=tmp.toLowerCase();
                        words1[nwords1]=tmp;
                        nwords1++;
                        tmp="";
                        a[i]=(char)ch;
                        i=i+1;
                        ch=dat.read();
                        if((ch!=13)&&(ch!=32))
                        {
```



```
                        a[i]=(char)ch;
                        i=i+1;
                        continue;
                }
                if(ch==13)
                {   a[i]=(char)ch;
                    i=i+1;
                    while((ch)!=13)
                        ch=dat.read();
                    dat.read();
                }
            }
            else
            {
                a[i]=(char)ch;
                tmp+=(char)ch;
                i=i+1;
            }
        }
        ml=i-1;
        for(i=1;i<=ml;i++)
            {
                ad[i]=a[i];
            }
        dat.close();
    }
```



```
        catch(IOException e)
        {
            System.out.println(e);
            System.exit(-1);
        }
    }
//to find out the longest match length
public static int longestmatch(int i)
    {
        int j,c,k,t=0,flag;
        String tmp="";
        if(da[i]==32)
            return 1;
        if(da[i]==13)
            return 1;
        for(t=8;t>=0;t--)
        {   k=i;
            for(j=0;j<=t;j++)
            {
                tmp+=da[k];
                k++;
                if(k==ml)
                break;
            }
            for(c=2;c<nwords;c++)
            {
                flag=tmp.compareTo(words[c]);
```

```
                if(flag==0)
                {
                        return(tmp.length());
                }
            }
            tmp="";
        }
        return 0;
}

//to find out the next break point
public static int next(int s)
{
        int t=0,nt=0;
        int max=0;
        for(t=s+1;t<=c[s];t++)
        {
            if(max<=c[t])
            {
                max=c[t];
                nt=t;
            }
        }
        return nt;
}
```



```
//to find out the execution time
public static double msecond() {
        Date d = new Date();
        double msec = (double) d.getTime();
        return msec;
}

//to write into the output file
public static void write()
{
        int v,f,k,t,flag=0;
        String tmp="";
        i=1;
        try
        {   //open the output file
            File primitive =new File("C:\\jpvm\\out.txt");
            FileOutputStream writes;
            writes=new FileOutputStream(primitive);
            DataOutputStream dos=new DataOutputStream(writes);
            while(nt5[i]<ml-1)
            {
                for(v=i;v<nt5[i];v++)
                tmp=tmp+ad[v];
                for(f=1;f<=nwords;f++)
                {
                    if(ad[v-1]==32)
```



```
                    {
                        dos.write((byte)12);
                        break;
                    }
                    if(ad[v-1]==13)
                    {
                        dos.write((byte)11);
                        break;
                    }
                    flag=tmp.compareTo(words[f]);
                    if(flag==0)
                    {
                        dos.write((byte)f+10);
                        break;
                    }
                }
                t=nt5[i];
                i=t;
                tmp="";
            }
                for(v=i;v<=ml;v++)
                    tmp=tmp+ad[v];

                for(f=1;f<=nwords;f++)
                {
                    if(ad[v-1]==32)
                    {
```



```
                        dos.write((byte)12);
                        break;
                    }
                    if(ad[v-1]==13)
                    {
                        dos.write((byte)11);
                        break;
                    }
                    flag=tmp.compareTo(words[f]);
                    if(flag==0)
                    {
                        dos.write((byte)f+10);
                        //System.out.print(f);
                        break;
                    }
                }

            k=249;
            dos.write(k);
        }

        catch(IOException e)
        {
            System.out.println(e);
            System.exit(-1);
        }
}
```

```
//master process which divides the process
// and gives them to slave processes
public static void master() throws jpvmException
{
    System.out.print("I am in master\n");
    readdict();
    jpvmTaskId tids[] = new jpvmTaskId[4];
    int t=0;
    int k1=0,k2=0,k3=0,k4=0,j=1;
     jpvm.pvm_spawn("ps",4,tids);
    jpvmBuffer buf = new jpvmBuffer();
    //for first slave
    for(i=1;i<=ml/4;i++,j++)
        ts[j]=a[i];
    buf.pack(1);
    buf.pack(mld);
    buf.pack(d,mld,1);
    buf.pack(j-1);
    buf.pack(ts,j-1,1);
    jpvm.pvm_send(buf,tids[0],12345);
    //for second slave
    j=1;
    for(i=ml/4+1;i<=ml/2;i++,j++)
        ts[j]=a[i];
        jpvmBuffer buff = new jpvmBuffer();
    buff.pack(2);
    buff.pack(mld);
```



```
    buff.pack(d,mld,1);
    buff.pack(j-1);
    buff.pack(ts,j-1,1);
    jpvm.pvm_send(buff,tids[1],12345);
        System.out.println(" ");
    jpvmBuffer buf1 = new jpvmBuffer();
    //for third slave
    j=1;
    for(i=ml/2+1;i<=3*ml/4;i++,j++)
        ts[j]=a[i];
    buf1.pack(3);
    buf1.pack(mld);
    buf1.pack(d,mld,1);
    buf1.pack(j-1);
    buf1.pack(ts,j-1,1);
    jpvm.pvm_send(buf1,tids[2],12345);
    jpvmBuffer buf2 = new jpvmBuffer();
    //for fourth slave
    j=1;
    for(i=3*ml/4+1;i<=ml;i++,j++)
        ts[j]=a[i];
    buf2.pack(4);
    buf2.pack(mld);
    buf2.pack(d,mld,1);
    buf2.pack(j-1);
    buf2.pack(ts,j-1,1);
    jpvm.pvm_send(buf2,tids[3],12345);
```



```
for(t=0;t<4;t++)
{
        jpvmMessage message = jpvm.pvm_recv();
        System.out.println("Got message tag " +
        message.messageTag + " from "+
        message.sourceTid.toString());
        int f= message.buffer.upkint();
        if(f==1)
        {
            k1= message.buffer.upkint();
            message.buffer.unpack(nt1,k1,1);
        }
        if(f==2)
        {
            k2= message.buffer.upkint();
            message.buffer.unpack(nt2,k2,1);
        }
        if(f==3)
        {
            k3= message.buffer.upkint();
            message.buffer.unpack(nt3,k3,1);
        }
        if(f==4)
        {
            k4= message.buffer.upkint();
            message.buffer.unpack(nt4,k4,1);
        }
```



```
    }
    for(i=1;i<k1;i++)
        nt5[i]=nt1[i];
        nt5[k1]=k1+1;
    t=k1+1;
    for(i=1;i<k2;i++)
    {
            nt5[t]=nt2[i]+k1;
            t++;
    }
    nt5[t]=t+1;
    t=t+1;
    for(i=1;i<k3;i++)
    {
            nt5[t]=nt3[i]+k1+k2;
            t++;
    }
    nt5[t]=t+1;
    t=t+1;
    for(i=1;i<k4;i++)
    {
            nt5[t]=nt4[i]+k1+k2+k3;
            t++;
    }
    write();
}
```

10.2 SAMPLE OUTPUT

CASE 1:

Running sequential dictionary compression using optimal parsing algorithm,

```
C:\>java so
Compression going on.....
Total time: 3054 msecs
C:\>
```

CASE 2:

Running parallel dictionary compression using optimal parsing algorithm,

```
C:\>java po
Reading dictionary....
Reading data.....
Received message from jpvm task, id: sys21, port #4032
Received message from jpvm task, id: sys22, port #4034
Received message from jpvm task, id: sys24, port #4035
Received message from jpvm task, id: sys27, port #4037
Total time: 1406 msecs
C:\>
```