Lossless Compression for Large Scale Cluster Logs

Raju Balakrishnan[1], Ramendra K. Sahoo[2]

[1]India Software Laboratory, IBM
Koramangala Ring Road,
Bangalore, India-560071
rajubala@in.ibm.com

[2]IBM T J Watson Research Center
19 Skyline Drive, Hawthorne
New York, USA-10532.
rsahoo@us.ibm.com

Abstract

The growing computational and storage needs of several scientific applications mandate the deployment of extreme-scale parallel machines, such as IBM's Blue Gene/L which can accommodate as many as 128K processors. One of the biggest challenges these systems face, is to manage generated system logs while deploying in production environments. Large amount of log data is created over extended period of time, across thousands of processors. These logs generated can be voluminous because of the large temporal and spatial dimensions, and containing records which are repeatedly entered to the log archive. Storing and transferring such large amount of log data is a challenging problem. Commonly used generic compression utilities are not optimal for such large amount of data considering a number of performance requirements. In this paper we propose a compression algorithm which preprocesses these logs before trying out any standard compression utilities. The compression ratios and times for the combination shows 28.3% improvement in compression ratio and 43.4% improvement in compression time on average over different generic compression utilities. The test data used is log data produced by 64 racks, 65536 processor Blue Gene/L installation at Lawrence Livermore National Laboratory.

1. Introduction

Recently developed massively parallel systems far exceed the computational power and data handling capabilities of systems available before. High performance systems running many applications may require to transfer and store petabytes of data [21]. In addition to high performance network and storage subsystems to handle such large amount of data, fast data compression schemes with high compression ratios can help data handling without incurring hardware cost and additional network bandwidth. IBM Blue Gene/L [3] is a recently developed large scale cluster to meet the computational and data handling demands of the scientific and industrial applications. The Blue Gene/L system installed in Lawrence Livermore National Laboratory has 128 K processors and 280 teraflops of peak computing power, which is going to be doubled in size and computing power by the end of the year 2005. The system is capable of running hundreds of parallel jobs in spatial partitions [20] and transferring many gigabytes of data per second.

A supercomputer of this scale will produce large amount of system logs. There are many alternatives to manage large volumes of system logs generated at a very non-uniform rate. Redundant log filtering is one of the popular methods [1]. Filtering is removal of records containing redundant or unimportant information which can be distracters rather than information providers during analysis. Filtering may cause a good reduction in log size by removing majority of the records in system logs. The other alternative is to use event compression and decompression techniques which has no requirement of analyzing the characteristics of the events even before they are archived.

The filtering and compression works in different levels. Lossless compression facilitates storage, archiving and network transfer of log data, saving storage space and network bandwidth. Whereas filtering facilitates analysis, and problem determination over the logs even before the logs are archived. One of the biggest disadvantages of filtering technique is the loss of as much as 99% of the data from the original logs. Records in the original records are being removed from the system logs, and can not be retrieved from the filtered logs. The filtering method is dependent on analysis to be performed on the logs, and records not important for a particular task will be important for another analysis. Hence it is not possible to perform filtering on logs and remove some records before archiving since lost information may prove to be important for other analysis.

On the other hand, the compression technique proposed in this paper is lossless. On passing the compressed data through the decompression algorithm reproduces the original data, with zero information loss. Data compression can help both storing and transferring such large amount of data. Considering the storage problem, efficient compression schemes can compress the data before storing, hence saving lots of storage space. Real-time streaming compression will also help to transfer such large amount of data through the network after compression, which can be decompressed at the receiving end, if required. If data is stored in a compressed format the received data can be stored as such, without decompressing. The compressed data can be processed and analyzed in compressed domain [10, 11], or can be decompressed and analyzed depending on the purpose.

This paper describes a lossless compression scheme for log data produced by Blue Gene/L prototype. The compression ratio and time of execution are compared with other popular compression utilities using generic compression algorithms. Our new compression algorithm provides 35.5% improvement in compression ratio and 76.6% improvement in compression time over bzip2 [13] and 21% in compression ratio and 72.5% in compression time over LZMA level 9 compression by 7zip [17].

2. Related work

Data compression is a matured area, and a number of generic and special purpose compression algorithm and utilities are available resulting good compression ratios and timings. General purpose compression utilities like bzip, bzip2[13], gzip [16] use generalized compression algorithms like Burrows-Wheeler Transform [4], Lmlpel Ziv algorithm [14] to name a few. These utilities can provide good compression schemes for large scale cluster event logs. However, the performance of log compression can be further improved, by leveraging specific attributes commonly observed within these large scale cluster logs. 7zip [17] compression utility, available on windows and UNIX platforms, implements many compression algorithms including one PPM(Prediction by Partial Matching)[15] which is one of the best performing algorithm on English text, and LZMA which generally gives good compression ratios than bzip2 [13].

Apart from these generic compression utilities, Bal´azs, Andr´as[2] discuss the log compression for web servers. Sahoo et al [1] discusses the filtering of failure logs of large scale clusters tested on Blue Gene/L data which can be used as a lossy compression technique. Pzip compression [18] proposes a better compression scheme for tabular data with fixed length records and fixed column widths. To the best of our knowledge, no work is done specifically to manage large amount of event logs in a lossless manner for large scale clusters while improving the compression ratio and timings.

3. Blue Gene/L architecture

Blue Gene/L compute node is made up of PPC 440 cores and two such cores are packaged in a compute chip. Each core has 32KB L1 cache, a 2 KB L2 (for pre-fetching), and share a 4MB EDRAM L3 cache. In addition, a shared fast SRAM array is used for communication between the two cores. The L1 caches including the internal buses in the core are parity protected., The L2 caches and EDRAM are ECC protected. Each chip has a total of four network interfaces such as ….(1) 3-D torus network, (2) a tree interface for global operations, (3) a gigabit Ethernet interface supporting I/O and host interface, while (4) a control network supports booting, monitoring and diagnostics through JTAG access.

A compute card contains two PPC 440 cores, and houses a 256MB or 512MB DRAM for each chip on the card. A node card contains 16 such compute cards, and a midplane holds 16 node cards (a total of 512 compute chips or 1K processors). I/O cards, each containing two chips and some amount of memory (usually larger than that of compute nodes), are also housed on each midplane. There are 4 such I/O cards (i.e., 8 I/O chips) for each midplane, i.e. one I/O chip for every 64 compute chips. All compute nodes are connected through a Gigabit ethernet interface to these I/O chips for their I/O needs. In addition, a midplane also contains 24 midplane switches to connect with other midplanes. When crossing a midplane boundary, the torus, global combining tree and global interrupt signals pass through these link chips. A midplane also has one service card that performs system management services such as monitoring errors and verifying the heart beat of the nodes. In most cases, a midplane is the granularity of job allocation, i.e., a job is assigned to an integral number of midplanes. Compute cards are normally shut down when they are not running any job. When a job is assigned, the card is reset and the network is configured before any execution begins.

4. Logging mechanism and log format

In case of Blue Gene/L, error events are logged through the Machine Monitoring and Control System (MMCS). There is one MMCS process per midplane, running on the service node. However, there is a polling agent that runs on each compute chip. Errors detected by a chip are recorded in its local SRAM via an interrupt. The polling agent at some later point pick up the error records from this SRAM and ship them to the service node using a JTAG-mailbox protocol. After procuring the events from the individual chips of that midplane, the service node records them through a DB2 database engine. These events include both hardware and software errors at individual chips/compute nodes, errors occurring within one of the networks for inter-processor communication, and even errors such as high temperature and fan speed problems are reported through an environmental monitoring process in the backplane. An event record has a number of fields such as Record ID, Event ID, Event Type, Facility and Location. Details of each field and possible options are as described below.

Record ID is the sequence number for an error entry, which is incremented upon each new entry being appended to the logs. These records are sequence of integers in ascending order.

Event time is the time stamp associated with the event. Time stamp format is YYYY-MoMo-DD HH:MM:ss.mmmiii (where Y representing Year, Mo, D, M, s, m, and i representing Month, Day, Minute, second, millisecond and microsecond respectively.)

Event type specifies the mechanism through which the event is recorded, with most of them being through RAS [10].

Event Severity can be one of the following five levels

1. INFO events are more informative in nature on overall system reliability, than problem or error reports, such as "a torus problem has been detected and corrected", "the card status has changed", "the kernel is generating the core", etc.
2. WARNING events are usually associated with node-card/link-card/service-card not being functional.
3. SEVERE events give more details on why these cards may not be functional (e.g. "link-card is not accessible", "problem while initializing link/node/service card", "error getting assembly information from the node card", etc.).
4. ERROR events report problems which are more persistent and further focusing on their causes ("Fan module serial number appears to be invalid", "cable x is present but the corresponding reverse cable is missing", "Bad cables going into the linkcard" etc) All of these above events are either informative in nature, or are related more to initial configuration errors, and are thus relatively transparent to the applications/runtime environment.
5. FATAL or FAILURE events (such as "uncorrectable torus error", "memory error", etc.) are more severe, and usually lead to application/software crashes.

Facility attribute denotes the component where the event has occurred, which can be one of the following: LINKCARD, APP, KERNEL, DISCOVERY, MMCS, or MONITOR. The LINKCARD events report problems with midplane switches, which are related to communication between midplanes. APP events are those flagged in the application domain of the compute chips. Many of these are due to the application being killed by certain signals from the console. In addition, APP events also include network problems captured in the application code. Events with KERNEL facility are those reported by the OS kernel domain of the compute chips, which are usually in the memory and network subsystems. These could include memory parity/ECC errors in the hardware, bus errors due to wrong addresses being generated by the software, torus errors due to links failing, etc. Events with DISCOVERY facility are usually related to resource discovery and initial configurations within the machine (e.g. "service card is not fully functional", "fan module is missing", etc), with most of these being at the INFO or WARNING severity levels. MMCS facility errors are again mostly at the INFO level, which report events in the operation of the MMCS. Finally, events with MONITOR facility are usually related to the power/temperature/wiring issues of linkcard/node-card/service-card. Nearly all MONITOR events are in the FATAL or FAILURE severity levels.

Location of an event (i.e., which chip/node-card/service-card/link-card experiences the error), can be specified in two ways. It can either be specified as (a) a combination of job ID, processor, node, and block, or (b) through a separate location field. We mainly use the latter approach (location attribute) to determine where an error takes place.

5. Compression utilities

This section gives a brief description of compression algorithms used by popular generic compression utilities. The intention is to provide an overview of features of the compression algorithms used by these utilities for better understanding of the paper. References are provided for readers interested in details.

gZip. Gzip is based on a variant of LZ77 [6] algorithm. The occurrences of a string are replaced by pointer to the previous occurrence of the string. If the previous occurrence is not found, the string is coded as such. The literals and references are coded using Huffman coding [5] after LZ77. The algorithm can compress streaming data.

Bzip2. bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm[4], and Huffman coding[5]. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors[14], and approaches the performance of the PPM family of statistical compressors[15] for text. Bzip2 can not work on streaming data as compression involves a sorting step.

7Zip. 7zip supports a number of compression formats including LZMA, ZIP, CAB, RAR, ARJ, LZH, GZIP, BZIP2, Z, TAR, CPIO, RPM and DEB. Default compression scheme for 7zip is LZMA (Lampalle Ziv Markov Chain Algorithm), which provides better compression than bzip but slow.

PPM. PPM stands for Prediction by Partial Matching [18]. PPM is a adaptive statistical data compression based on context modeling and prediction. PPM models use a set of previous symbols in the uncompressed symbol stream to predict the next symbol. Recent PPM implementation are among the best performing lossless compression programs for english text.

6. Compression mechanism and system details

6.1 Problem Background

The compression work had started before Blue Gene/L system was operational. The idea was to address the Reliability, Accessibility and Serviceability issues of particular clusters architectures parallel to hardware and software development, to address these issues in commercial offerings with zero time to market after the hardware is available for commercial offerings. Earlier work was performed using the log date from a smaller prototype of Blue Gene/L in IBM Rochester, rather than on the full scale 64 rack Blue Gene installation in Lawrence Livermore National Labs [22]. The mechanism of logging and nature of log data in both systems are the same, though the log data rates are much higher in 64 rack LLNL systems than that in 2 racks IBM Rochester prototype. As described earlier the main goals of designing a custom compression algorithm are (1) To save the storage space and (2) compress the log data sent across the network to reduce the bandwidth requirements.

Blue Gene/L logs contain large fraction of redundant information. Similar problems are one of the common issues while addressing error events for large-scale clusters as reported in literature [1]. Most of these redundant log records form adjacent records in the output logs with majority information being repeated either due to different software or hardware retries until the time out period.

The compression is achieved by implementing a compression pipeline, with different compression algorithm used in each stage. The idea was to use mix and match of different compression algorithm in different stages to achieve better compression ratios, compression time and ability to handle streaming data. For example, using Burrows-Wheeler [4] as a stage in the pipeline gives better compression than LZ, but streaming data can not be handled by such a pipeline. Hence we can use LZ instead of Burrows-Wheeler in a pipeline compressing streaming data. Similarly, the combination of algorithms in the pipeline can be changed to meet other constraints imposed upon overall compression step.

The custom compression algorithm was designed to use only specific characteristics of Blue Gene/L logs rather than generic characteristics of English text. This is important, since if the custom compression algorithm is utilizing any generic characteristics of text the algorithms down the stream may be depending on the same characteristics for compression. Since the data is already compressed using this feature, the combined compression ratio will not improve, even though the custom compression algorithm will give better stand alone compression ratio, which is not the target.

Decompression is also faster than the standalone standard compression algorithms for most of the compression pipelines. The custom algorithm does not use a dictionary for compression. This evades the use of transferring a dictionary in case compressed data is transferred over a network.

6.2 Dataset

For final testing, log data collected for 103 days from 64-Node (65536 processors) LLNL Blue Gene/L installation, which currently tops the top500 list [3] with 137 teraflop of peak computing power, was used. For initial development and testing logs collected from a 8192-processor Blue Gene/L DD1 at IBM Rochester, which is currently ranked 16 in the Top 500 list of supercomputers [3]. The machine has been up since May, 2004, and has been primarily running parallel scientific applications. Results in this paper are based on the LLNL log data compression. The uncompressed log data was 195Megabytes in total and contained 1184010 log records. Please note that the amount of log data produced can be much more since log generated depends on machine utilization and logging behavior of applications running.

6.3 Compression method

We implemented a compression pipeline with different encoding methods as stages in the pipeline. This pipeline approach allows us to use combination of different encodings to achieve required characteristics such as higher compression ratios, ability to handle streaming compression etc. Irrespective of algorithms used later stages, we use the custom encoding scheme proposed in this paper as the first stage in the pipeline.

First step for designing the custom compression was to identify specific characteristics of Blue Gene/L logs which may aid a better compression. Most of the errors in BG/L event logs are reported by the thousands of different hardware components with same entry data, severity, and facility, but differing in record id, time stamp, and resource serial number (Please see the section describing log format). For example, failure to find a program image is reported by APP facility associated with all compute chips running the job. After careful observation and experimentation, we decided to leverage on following trends in the Blue Gene/L logs to design custom compression algorithm.

1. Most of the columns in the adjacent records tend to be same.
2. Record Ids are ascending integers in sequence.
3. The columns differing in most of the adjacent records are record ids, time stamps, serial number of the device, and location identifier.
4. For these fields(mentioned in 3 above) higher order bytes in the columns in the adjacent records tends to be the same, lower order bytes are differing from each other.
5. If a byte differs in a column for two adjacent records, the lower order bytes-bytes to the right to the differing byte-also tend to be different.

Let us examine the reasons for above trends in log data briefly. Trend 1 and 3 are due to the same error being reported by multiple devices. Trend 2 is the way logging is implemented. Reason for trend 4 is the temporal locality of adjacent records and spatial locality of the devices creating them. As noted in trend 3, columns differing in most of the adjacent records are record ids, time stamps, serial number of the device, and location identifier. Record ids are incrementing sequentially. For time stamps the higher order bytes recording date, hour, minute and second and even millisecond tend to remain the same, but microsecond part is generally differing from each other due to temporal locality, hence the lower order bytes will be differing (Note the time stamp format given in section above.). For location identifier, due to spatial locality, the rack, midplane and mostly node card of the processors producing records tends to be same, which comes as higher order part of the location identifier. For serial number of the devices, trend 4 is less evident, but first two leftmost bytes tend to be same for adjacent records.

Also we noticed that many of the generic compression utilities are taking more than linear compression time, hence reduction in input data size results more than linear improve in the compression time. For example: if a generic compression algorithm takes $O(n^2)$ time for compression reduction in size of the input by 50% in earlier stages in compression pipeline will reduce the compression times to 75% of the original time.

Based on these observations we decided to use a modified incremental encoding, which is a variant of delta encoding, for ordered text [23]. The basic compression strategy is to compare the given record with the previous record and encode only the difference. If the column in current record is same as the corresponding column in previous record proceed to the next column and nothing is encoded into the output. If there is a difference, continue after encoding current token as described below.

The encoding scheme for the differences is as follows. If there is difference at a byte for a token, the rest of the token is written as such in the encoded stream along with the offset from the last difference in the same record. For the first difference in each record, the offset from the beginning of the record is encoded instead of offset from the last difference. One difference is encoded as separator tab, offset, separator tab, and the difference. Hence encoding takes one or two bytes offset, two bytes for tabs, and the differing data bytes.

Percentage of increase in size is high for small runs of byte differences. For example, assuming a single byte offset, to encode a single byte difference we need to write 1 byte offset, two bytes for two tabs, and one byte difference to output compressed stream. This causes an encoding overhead of 300%. (The probability of decrease in size for next offset due to this difference is not considered here). But consider we are encoding a difference 10 bytes in a run, total size in output stream will be 13 bytes, or 30% of increase in size due to encoding. So it is desirable to keep the runs of differences to be encoded as lengthy as possible. Hence, based on Trend 5 above, if we encounter a difference of single byte in a column, the rest of the column is encoded as a single run of difference in without checking rest of the column and algorithm proceeds with encoding the next column. The tabs are used as separators for encoding since tabs are not used in log data.

As an example of encoding, let columns in two adjacent log records be,

First Record: FATAL 2004-12-14 22:52:46.714244
Second record: FATAL 2004-12-14 22:52:46.715247

Since FATAL is same for two lines nothing goes into output. For the time stamp everything till 2004-12-14 22:52:46.71 is same so nothing is encoded.

On encountering first difference, i.e. 4 and 5, the rest of the column is encoded as *offTAB*5247TAB and proceed to next column, where *off* is offset from last difference and 'TAB' stands for tab character. For next difference offset will be difference from position of the character 5 in the second record above. After doing encoding of each record, the length of the encoded record is compared with that of the original record, and if the

Compression Scheme	Compression Ratio x 100	Compression time (In seconds)	decompression Time (In seconds)
Custom	37.47	5.79	7.25
bzip2	6.45	192.72	22.42
Gzip	11.17	11.07	2.33
7zip PPM	6.42	36.61	38.49
7zip LZMA	5.35	222.05	7.64
7zip LZMA level 9	4.48	1324.29	7.15
custom > bzip2	4.17	5.79 + 58.77=64.56	7.25+13.47 = 20.72
custom > gzip	7.10	5.79+8.12=13.91	7.25 + 2.33 = 9.58
custom > 7zip LZMA	4.03	5.79 +93.76=99.55	7.25+4.01 =11.26
custom > 7zip PPM	4.47	5.79 + 19.46 = 25.25	7.25+20.12 = 27.37
custom > 7zip LZMA level 9	3.61	5.79+ 358.13 = 363.92	7.25+3.92 =11.17
Gzip > 7zip LZMA level 9	10.19	36.10+11.07 = 47.17	5.19+7.25 = 12.44
Gzip > bzip2	10.53	17.05+11.07=28.12	7.26 +2.33 = 9.59

Table 1: The compression ratios, compression time, and decompression time for different compression schemes. The execution environment was Red Hat Linux 3.2.2-5 on an Intel Xenon 2.4 GHz with 4 GHz RAM. "custom" stands for algorithm proposed in this paper, 1 > 2 means output of 1 is given as input of 2 for compression. File based communication was used for testing combination of utilities; hence it incurs additional I/O operation time, though this is small compared to overall timings.

encoded length is greater than that of the compressed record the original record is kept as such. This condition is always true for first record in log data, since the previous record is null and encoding increases size. The first record gives the starting record id for the records which can be incremented for each record and restored. We used a new line character to separate record; the original record is kept as such records in compressed format. This allows the compression to omit record ids from compressed files.

For decompression, the reverse process is performed on encoded data. Till the first offset, bytes are copied from previous record in sequence. Then the bytes from the encoded line are copied to the output line. After this, copying from the reference record line continues for the next token till the next difference. As mentioned above, first line is always kept as such in encoded file. Record ids are incremented for each record and recreated in the output record. If we see an uncompressed line, which is marked by the long record id as the first token, it is copied as such to the output record. Having the n-1th record, the nth record can be decoded; hence the decompression can handle streaming data.

6. Results and discussion

The combination of preprocessing and different popular compression algorithm is tested on log data, and compression ratios, compression and decompression timings are tabulated in Table 1.

The compression ratio of standalone custom encoding is low, as shown in row 1 in the Table 1. However, the combination of custom compression algorithm with all other compression utility gives 28.3% better compression ratios and 43.4% of improvement in compression time on average than stand alone compression utilities. Combined compression time also shows a decrease except for combination with gzip. The best compression ratio is for the combination of custom and LZMA level 9 in 7zip, which shows 21% improvement in compression ratio and in 72.5% reduction in time over the standalone LZMA. The combination of gzip and custom algorithm shows 36% improvement in compression ratio though compression time shows an increase, but combination as such is faster than other schemes and can handle streaming data. The combination with bzip2 shows 35.5% improvement in compression ratio and 76.6% reduction in compression time. The last two rows show the combination of gzip used as a preprocessor for with bunzip2 and LZMA, which gives less compression than the standalone versions of bunzip2 and LZMA.

Decompression shows an increase in time for some combinations. But decompression time is less important, as decompression is performed offline and only on parts of data need to be analyzed.

7. Summary and future work

The compression algorithm is effective to achieve higher compression ratios and less compression timings, and fast enough to handle data rate of Blue Gene/L LLNL installation logging. Since the custom compression can operate on streaming data, in combinations with a compression utility which can handle streaming data such as gzip, it can be used for compression prior to network transmission of log data in order for reduced bandwidth requirement. Similar approach can be followed for other kind of data also.

One notable feature is that the preprocessing step is a simple one, and achieves only small standalone compression, but very effective in combination with generic compression utilities. The idea of preprocessing data with a custom scheme before passing to a generic compression utility is an effective method for better compression incurring least implementation overhead. Also it shows that cascading generic compression utilities which can give good standalone compressions is not an effective method, as it sometimes increases compressed sizes.

As a future work, compression ratios and timings for the method on generic tabular data can be tested, as this method or a variant may be effective for this purpose. Also applicability of method for data compression for logs of other kind, like loosely coupled clusters and architectures other than Blue Gene/L need to be tested. There is scope for optimizing the decompression algorithm implementation, since not much care was taken to optimize decompression, since running time is much less compared to the compression times and not very crucial for our work in hand.

References

[1]Y. Liang, Y. Y. Zhang et al. Filtering Failure Logs for a Blue Gene/L Prototype. In *Proceedings of IEEE International Conference on Dependable Systems and Networks* , 2005.

[2] Bal´azs R´ACZ, A. Luk ´acs. High density compression of log files. In *Proceedings of Data Compression Conference (DCC'04)*, IEEE Page 557, 2004.

[3] Top 500 Supercomputers in world list, for June, 2005. http://www.top500.org.

[4] M. Nelson. Data Compression with the Burrows-Wheeler Transform. In *Dr. Dobbs Journal September* 1996.

[5] D. A Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the IERE, vol. 40, Pages 1098--1101*, 1952.

[6] J. Ziv, A. Lampel. A Universal Algorithm for Sequential Data Compression. In *IEEE Transactions on Information Theory,* May 1997.

[7] J. C. Mogul, Fred Douglis. A. Feldman, B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *ACM SIGCOMM Computer Communication Review*, Volume 27, Issue 4, Pages 181-194, October1997.

[8] The Blue Gene/L Team, IBM and Lawrence Livermore National Laboratory. An Overview of the Blue Gene/L Supercomputer. In *IEEE Supercomputing 2002 Technical Papers,* 2002.

[9] G Almasi, L Bachega et al. System Management in the Blue Gene/L Supercomputer. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[10] B. C. Smith, L. A. Rowe, Algorithms for Manipulating Compressed Images. In *Proceedings of IEEE Computer Graphics and Applications*, vol.13, No 5, Pages 34-42. September 1993.

[11] S. Acharya, B. Smith. Compressed Domain Transcoding of MPEG. In *Proceeding of IEEE Multimedia*, 1998

[12] S.J. O'Connell, N. Winterbottom. Performing Joins without Decompression in a Compressed Database System. In *SIGMOD Record,* Vol. 32, No. 1, March 2003.

[13] Bzip2 and libbzip2 project official home page, http://www.bzip.org/ .

[14] J. Ziv, A. Lamapel. A Universal Algorithm for Sequential Data Compression. In *IEEE Transactions on Information Theory*, May 1977

[15] M. Drini´c, D. Kirovski et al. PPMexe: PPM for Compressing Software. In *Proceedings of the Data Compression Conference,* IEEE, 2002.

[16] gzip official home page, algorithm description, http://www.gzip.org/algorithm.txt.

[17] 7 zip project official home page, http://www.7-zip.org.

[18] J. G. Clary, I. H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on communication*, April 1984

[19] IBM Research, Blue Gene Project page, http://www.research.ibm.com/bluegene/

[20] A.J.Oliner, R.K.Sahoo, J.E.Moreira, M.Gupta, A Sivasubramanium. Fault-Aware Job Scheduling of Blue Gene/L System. *18th International Parallel and Distributed Processing Symposium proceedings*, 2004.

[21] T. C. Kramer, A. Shoshani, D. A. Agarwal, B. R. Draney, G. Jin, G. F. Butler, and J. A. Hules. Deep Scientific Computing Requires Deep Data. In *IBM Systems Journal*, Volume 42, Number 2, Pages 209-233, 2004.

[22] Blue Gene Home, Lawrence Livermore National Labs, http://www.llnl.gov/asci/platforms/bluegenel/ bluegene_home.html

[23] Delta encoding in HTTP, RFC3229, http://www.ie f.org/rfrfc3229.txt