Needles in a GigaStack

May 10, 2008

Team Bonampak:
Artena Hiebert
Mike Neilsen
Dave Sebesta

CS 5621 – Computer Architecture

Project Stage – Omega Report

Introduction:

The processing power of computers has advanced to a point which few people had ever thought possible. Collaborating the capabilities of modern day processors has led to a revolution in the field of supercomputing, and though this advance has allowed us to evaluate and solve exponentially complex problems, the amount of data produced by these methods has also increased at an exponential rate. Scientific simulations, human language processing, and corporate research and development list only a very small number of applications for which supercomputing power can utilized for both the generation and analysis of very large quantities of data. To handle such large quantities of data, it is essential that efficient use of these computing resources are discovered and utilized. In the following sections, we will look at some of these methods and discuss the trials of one team's struggle to topple the analysis of the massive news article collection know as Gigaword through the use of parallel processor supercomputing.

Problem Description:

This project aims to analyze the Gigaword corpus of text, a collection of over 10 gigabytes of news articles including articles from Associated Press, The New York Times, and more. A method is desired such that the number of words, number of articles, distinctness of n-word phrases, and a degree of calculated interest can be determined.

The importance of a term is based upon how unique it is in the corpus of text. Specifically, the degree of interest for a given term is based upon a function of the number of times the term occurs in the archive, the number of articles in the archive, and the total number of articles that contain one or more occurrences of the term.

This program will determine common words in any language in addition to finding unique words and terms. The information may be important to linguists in observing phrases used in writing in different geographic areas which can be varied depending on the files input. In this context, Bonampak could be useful for textual and statistical analysis. While this applies directly to searches and databases, the possibilities for specific applications are limitless.

This can be used extensively for natural language processing. One example of this is word sense disambiguation. Linguists may be interested in determining the uniqueness of terms to infer a value of overall importance for specific terms of the text, such as entropy calculations or data inference.

Another use could be if one is looking in the ACM portal for an article pertaining to a specific topic—such as 'word mining corpus gigaword processing parallel SMP blade'. Bonampak could return a listing of terms in the order of importance which may be show to be more useful than ACM's current portal search.

Methodology:

Using Worker-Manager paradigm, the Bonampak manager will take a directory of files and go through each file distributing articles to workers. The workers will run in parallel to find the number of word phrases in an article from n to m. The worker stores these into a hash table with the word phrase as the key mapped to an array holding the number of terms in phrase and the number of occurrences. Then the worker will return these counts to the manager by converting the hash into a struct holding the same data. In order to send this structure to the manager, we have to create a derived data type. In order to create a data type, first you must construct it, then allocate, then commit and use it. After doing this the manager will receive the struct and convert it into a hash table for quick reference. The manager's master hash table holds the key (term) mapped to an array holding the same data as before with the addition of a count of the number of times a word occurred in an article. This last variable will be used to calculate the uniqueness of a term.

Alpha Stage:

For our alpha release, we chose to partition the work such that each file would represent a primitive task. To distribute the work in a parallel fashion, we adopted the worker-manager design and gave the manager the task of assigning each worker process a file. Once the worker received the file, it would process it and return the results to the manager, which would, in turn, send another file to be processed. While this design was effective in the sense that it benefited from an increased number of processors running the application, it had a significant short-coming.

The design did not compliment situations in which there are few files. In the worst case possible—running the application with only one file—there is no benefit from concurrency, no matter how many processors are running the program. Because each worker task is a single file and there is only one file to process, there is only one task for the manager to distribute. Every process in excess of the manager and a single worker is immediately terminated, and the application is reduced to serial performance.

Beta Stage:

In the beta stage, we chose to partition the work into smaller pieces. We would consider each article in each of the input files as a primitive task instead of using one file per task. This essentially eliminates the previously mentioned problem, as a single file with multiple articles can be processed in parallel. Unfortunately, a bottleneck still exists with this approach. If there are fewer articles than worker processes, excess processes will be terminated, as they are unnecessary. We hope to resolve this with the final version.

So with beta, files are opened and parsed. In each file will be a number of articles which will be all of the information before a newline. When the program comes across a newline it will distribute the article to a worker and continue onto the next article. It will do this until it has reached the end of file and then continue processing files until it has finished working through all of the files in the directory.

When a worker receives an article it will process all of the word phrases in the article and find terms from lengths M to N where M is the lower bound and N is the upper bound on the number of words in a phrase. The worker will store these terms as keys in a hash table with the associated number of times the phrase appears in the article. Once this is done it will send th e results of its computation back to the manager. The manager will total all of the results and print the number of unique phrases from lengths M to N.

In omega, we changed how we were sending data. This means that instead of three sends we are only using two. The other changes made were cleaning up the code itself and implementing a way to compute and print the interesting word phrases. We faced many challenges getting the code back up to beta after fixing it, but after we did we were pleased with the progress and our results.

Team Bonampak Trials:

While working to implement beta, we ran into numerous complications. We were successful in creating methods that would go through a directory, then go through the files in that directory. The manager would send these articles to the workers. After some complications we were able to get the manager to send the articles and the workers to receive them. The worker would then go through the article and collect the number of times word phrases of length M to N would occur. We stored these on a hash table and that made it easy to see if we had multiple occurrences of the word phrase. After calculating the results, the worker would send them back to the manager. This is where we ran into considerable challenges.

First we tried to implement sending the hash table by creating a MPI derived type. This proved to be a lot of work and we ran out of time before the beta release was due. With the extension we were able to scale back and create 2-D arrays that we could successfully send from the worker to the manager. Then we worked on reading the data back into a hash table so that the manager could merge the incoming results from multiple workers quickly. Here is where we were accidentally double-freeing memory. Once this was resolved we were able to merge the results and find the number of N word terms. Because we are using a hash table, ideally lookup of the most interesting words should be easy for the final stage.

During beta we found one other set back. After we succeeded with sending the articles and the results, we noticed that for certain articles (and certain amounts of articles) we would sometimes get output that indicated the manager was continually sending an article to a worker, and the worker was continually sending a work request. Interestingly enough, this seemed to be resolved by giving MPI a break and then running the program. The final beta battle became getting the results to travel back to the initial setup and print out. As it was we had results and the number of word terms when looking at the output.

In the final stage we recovered drastically from the frenzy of submitting beta (in addition to the extended deadline). We cleaned everything up and after resolving some very mystifying errors we had the final version almost complete.

Beta Benchmarks:

We were able to get our program to count distinct N word terms on the included directory as long as the program doesn't time out. When counting 1 to 1 we can return an accurate count of the total number of words, however with any range after that the number of words may vary.

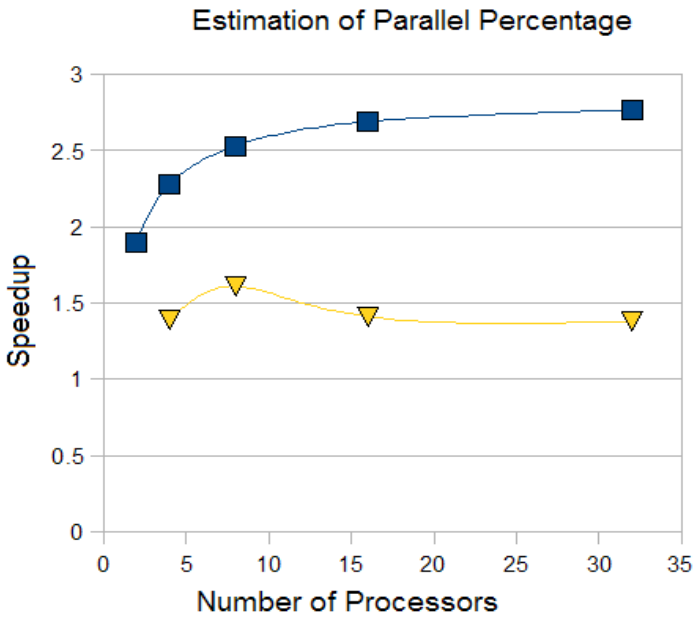
For the final project our plan is to rework the code. We plan to look at the individual functions with special attention to the results passing. We will also fix the word count. Hopefully we will be able to find the most important terms according to the specified equation

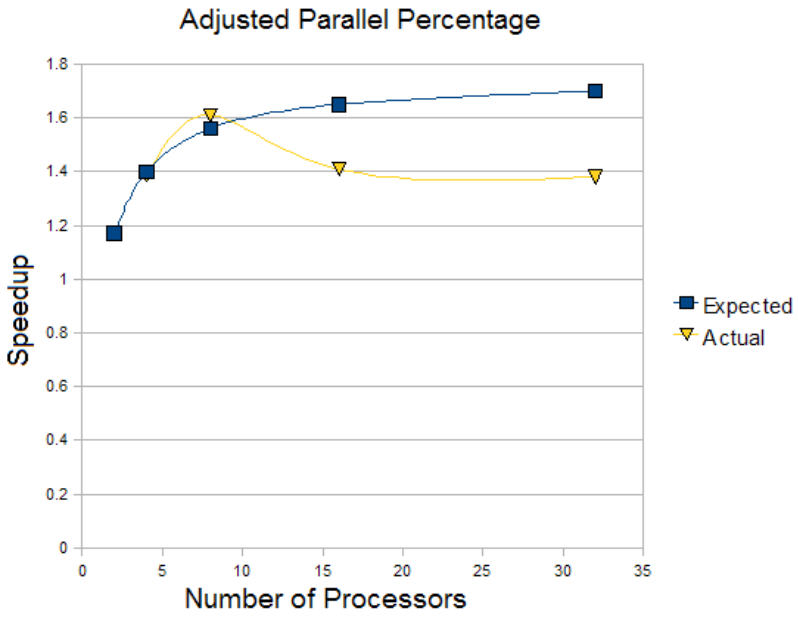
Our current shortcomings are largely due to inefficient message passing—both in design and performance. For the next stage, emphasis will be placed on designing a single package representing the results of a workers' computations which will be transferred by workers to the manager. This package will contain a table of each necessary term in the article along with its corresponding number of occurrences and the number of words in the term. Along with each package, a total word count for the article be attached. This will make the manager's task of merging the workers' results much more manageable. Performance will benefit from this condensed message passing approach by cutting down on the total number of communications between processes. As currently stands, the typical communication between a worker and the manager takes two sends and two receives. By reducing this to one send and one receive, the average communication rate will decrease by 50%.

In Beta running the program with range 1-1 we found:

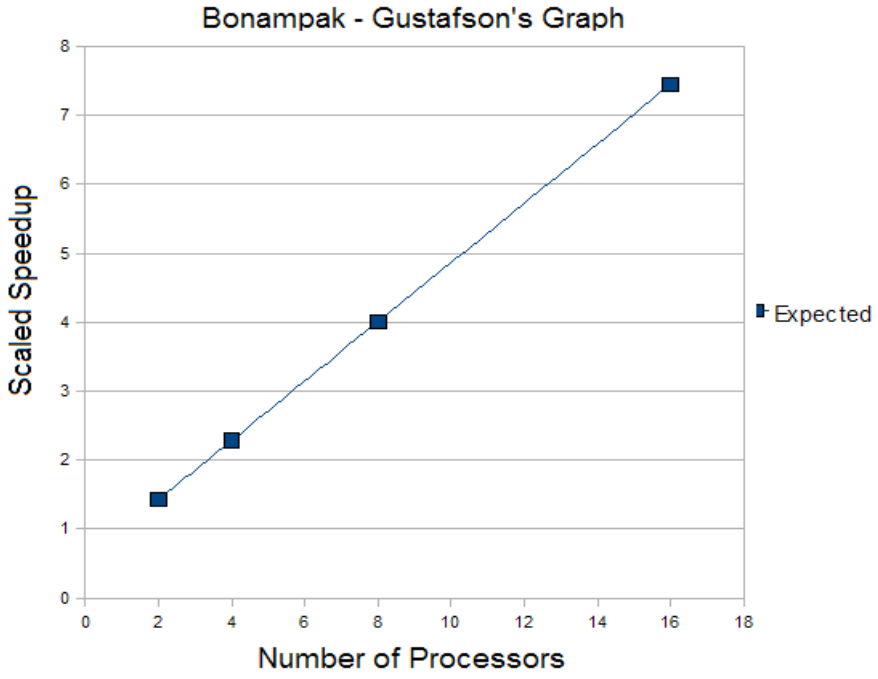
2	.006512 s
4	.004606 s
8	.004595 s
16	.004875 s
32	.005255 s

Using Amdahl's law we guessed Bonampak was only running in parallel 65% of the time. Our results seemed to disagree, after some calculations we concluded that Bonampak is actually running in parallel only 43% of the time. These differences are shown in the graphs below.





Along the lines of Gustafson as the facilities increase (processors) the problem will get bigger. Using his linear equation for scaled speedup our graph looked as follows:



Omega Benchmarks:

For the final stage of our project, we decided to review all of our beta code to further error proof the last stage. This took a lot of time and we implemented some changes that made the sending and receiving much more concise. We are now creating an MPI structure to send our data (from the hash table) and then receiving this structure and converting it back into a hash table. After a lot of error

checking we had beta working again, but only with small files. Bonampak is able to get any range of n-m word terms on files up to about 1 MB. One major limitation of Bonampak is that an article length must not be greater than the program's set buffer size (variable outside runtime, about 2500 bytes). The following is what Bonampak found on a file of 935,682 bytes:

There are 1069 articles and 160137 words
Benchmarking: 12.028066 seconds on 2 processors.
The number of distinct 2 word terms is: 5312
The number of distinct 3 word terms is: 6413
The number of distinct 4 word terms is: 6669
The number of distinct 5 word terms is: 6728

Once we got it working on multiple processors and we were pleased with the following results:

```
     There are 1069 articles and 160137 words
Benchmarking:   1.844824 seconds on 2 processors.
   The number of distinct 1 word terms is: 1998

-------------------------------------------------

     There are 1069 articles and 160137 words
Benchmarking:   0.925066 seconds on 4 processors.
   The number of distinct 1 word terms is: 1998

-------------------------------------------------

     There are 1069 articles and 160137 words
Benchmarking:   0.801151 seconds on 8 processors.
   The number of distinct 1 word terms is: 1998

-------------------------------------------------

     There are 1069 articles and 160137 words
Benchmarking:   0.811024 seconds on 16 processors.
   The number of distinct 1 word terms is: 1998

-------------------------------------------------

     There are 1069 articles and 160137 words
Benchmarking:   0.840119 seconds on 32 processors.
   The number of distinct 1 word terms is: 1998

-------------------------------------------------
```

Next we worked on implementing the computing the 'interestingness' of a word. To do this we decided to use yet another hash. However, Bonampak wasn't able to mature fully. Computing the final 'importance' ratings required somewhat more effort than was expected. We had a "master results" hash table, containing each term, the number of words in the term, and the number of occurrances in the corpus as a whole. Because of asymmetries in the required input of the output and importance computations and our design, we had to design a new hash table type specifically for the master results table. This new table contained the previously mentioned values (and still used a term as the key) and came with the new addition of an article tally to count the number of times the term occurred in at least once in an article, and a value for the importance (initially defaulting to zero). After the master results table accumulated all of the results, Bonampak's next task was to insert a valid importance value into the table for each term and then sort the table based on the level of importance. We were able to treat this hash table as a linked list, due to the design of the library (uthash), which made "sorting a hash table" a reality. With these sorted values, we ran a printing algorithm to print the top N terms (as specified by the initial user input).

This is when many bugs in the revised master table design and logic surfaced. We were able to print the terms and their associated importance values--but the importance values were quite askew. Through some tracing and debugging, we discovered that the logic for building the master table was inserting one "garbage" value in a critical entry for the importance computation. Fixing this did help with eliminating the '-inf' and 'nan' values we were getting for our importance values (the poor numerator didn't stand a chance), but more troubles awaited us. While we had values that appeared to be correct, another flaw in the master hash table management emerged. Some entries were being counted multiple times during the hash table processing, and fixing this issue became a somewhat difficult burden, given the few minutes remaining until deadline.

Future Works:

Throughout development of the Bonampak implementation, the worker-manager paradigm has catered to our needs. Designating a single manager that distributes articles (or files) to an assembly of workers allows the work to be divided in a balanced fashion. A concern that came up while working with this method was the question of how much time an arbitrary worker spent waiting for the manager to be able to send it a new article. If one manager is trying to service, for example, 127 workers at the same time, most of those workers will spend a considerable amount of time blocked since the manager can send only one article to one worker at a time. So, what if we were to promote another manager to help the distribution process? With two managers able to simultaneously hand out articles to eager workers, the amount of time spent waiting in the worst case scenario could be cut in half. Generalizing the idea, if we were to assign a number of managers in proportion to the total worker count, the time spent waiting by a given worker could be drastically reduced. This leads to the final question of worker-manager organization. If there are, say, 8 managers handing out articles and receiving results, how will the managers themselves be managed? Managers must be coordinated in their efforts to open files (so that the same file is not being read more than once), and tally the results that the workers compute. The answer is to designate one process as a CEO, a process that will distribute files or large chunks of articles to the managers, who in turn distribute individual articles to individual workers. After distribution, the results of the workers are relayed to the managers, who in turn relay their results back to the CEO, who keeps track of the final congregation of results.

From this the CEO-Manager-Worker paradigm was born. Unfortunately we ran out of time with omega and were not able to implement this idea for structured parallelism.

Cost Evaluation:

This project was extremely interesting and challenging. For the user's sake, we thought we would include a general idea of what it might cost to create the Bonampak experience.

183.7 man hours (x$42.2/hour)	--------	$7752.14 (no tax)
pizza and laffy taffy	--------	$45
coffee	--------	$50
travel	--------	$83
therapy	--------	$400
passing class	--------	priceless.
TOTAL:		$8315.14

Related Works:

This article by Vipin Kumar, member of the Computer Science department at the university of Minnesota, and Mohammed Zaki, member of the Computer Science department at Rensselaer Polytechnic Institute, describes the need for data mining and techniques by which it can be done (Kumar, V. and Zaki, M. 2000.). It looks specifically at using parallel processing as a means to acquire statistical observations from large sources of data. In their report, Kumar and Zaki look at several aspects of parallel processing.

In one analysis, they compare the methods of task and data parallelism. Using the technique of data parallelism, the work of finding interesting data among a large data set is divided up among a portion of the total processors available, with each processor executing the same instructions. This breaks the data set into many smaller sets and thus reduces the amount of serial processing that takes place. Using task parallelism, the data set is not divided up as it was in the data parallelism technique, but rather, each processor has its own set of instructions which it is executing on the entire data set. This also allows for a reduction in serial processing. In our problem of finding interesting words and terms within a large corpus of text, either of these techniques or a combination of the two could be beneficial. Data parallelism would allow us to take the corpus of text and partition it into a number of smaller segments, allowing each processor to find interesting terms within its own section. Task parallelism would allow us to designate each processor its own specialized task of finding one-word terms, two-word terms, etc. Since we are dealing with a large number of processors (>32) a combination of these techniques may work. For example, groups of processors may be assigned a segment of the corpus, in which each processor in the group will be designated the task of finding one, two, three, etc. terms for that given segment. Also, Kumar and Zaki look at varying types algorithms which can be used to optimize the workload between processes. The techniques described include static and dynamic load balancing.

A statically balanced load is predetermined before a processor begins executing on it. This allows a process to work in a more independent space from other processors due to a reduced amount of communication. In our problem, a processor would initially be given a certain percentage of the entire corpus to work with, with each processor's work totaling up to the whole corpus. A dynamically balanced load would be used in a situation where a corpus of text is segmented into varying file sizes. These files would be assigned to processes in a first come, first served fashion, where a process requests a new segment when it has finished with the one had been working on. In our problem, it cannot be assumed that the corpus will be presented as many varying sized files, so a method based on static load balancing may be favored in this case.

The results of the research conducted by Kumar and Zaki provide insight as to what we might select as a method of execution in our own implementation. Although the goal of our project is to determine specific occurrences of particular word sequences, as opposed to finding probability, the methods are very similar and can be applied in a similar manner.

One similar example is called 'Query Based Event Extraction along a Timeline' by Hai Leong Chieu and Yoong Keok Lee. In this project they are trying to abbreviate a large collection of documents that were news articles and put sentences that report “important” events on a time line. In this example they are assigning dates by using rules to resolve dates based on the documents date which is time tagged. Sentences that are associated to a date duration are extracted based on the date of the sentence. It seems that most of the processing went toward computing the importance within sentence segmentation.

The work done on this study was interesting and its purpose was to allow users to have quick overviews of events relating to their query. Their system did not require noun phrase chunking or naming entity recognitions. The hope is that the system would serve as an aid to users doing research on events that are covered in multiple news articles.

Another paper explores various ways to exploit the principle of locality to efficiently manage or process large sets of data. The focus is primarily on external memory, and Vitter is mainly concerned with efficiency in terms of I/O operations and disk usage.

This article also explores specific data structures that can be used to further exploit locality; in particular, external hashes and trees are explored in depth. Section 9, titled "External Hashing for Online Dictionary Search," explains that "the advantage of hashing is that the expected number of probes per operation is a constant, regardless of the number N of items." Section 10, titled "Multiway Tree Data Structures," states that "an advantage of search trees over hashing methods is that the data items in a tree are sorted..."

Much of the later sections of this paper detail specific examples of using principles of locality and data structures to efficiently operate on external memory. Section 13 is of special interest to our project, as it describes several operations regarding string processing.

Works Cited:

1. Chieu, Hai Leong, and Yoong Keok Lee. "Query based event extraction along a timeline," *Annual ACM Conference on Research and Development in Information Retrieval archive* (2004): 425-432. http://doi.acm.org/10.1145/1008992.1009065 (accessed April 10, 2008).

2. Vipin, Kumar, and Mohammed Zaki. "High performance data mining (tutorial PM-3)," *Conference on Knowledge Discovery in Data: Tutorial notes of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* (2000): 309-425. http://doi.acm.org/10.1145/349093.349109 (accessed April 10, 2008).

3. Vitter, Jeffrey S. "External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)* 33. 2 (2001), 209-271, http://doi.acm.org/10.1145/384192.384193 (accessed April 9, 2008).