Dynamic Role Allocation for Small Search Engine Clusters

Ndapandula Nakashole
Department of Computer Science
University of Cape Town
Private Bag X3, Rondebosch, 7701
nnakasho@cs.uct.ac.za

Hussein Suleman
Department of Computer Science
University of Cape Town
Private Bag X3, Rondebosch, 7701
hussein@cs.uct.ac.za

Calvin Pedzai
Department of Computer Science
University of Cape Town
Private Bag X3, Rondebosch
cpedzai@cs.uct.ac.za

ABSTRACT

Search engines facilitate efficient discovery of information in large information environments such as the Web. As the amount of information rapidly increases, search engines require greater computational resources. Similarly, as the user base increases search engines need to handle increasing numbers of user requests. Existing solutions to these scalability problems are often designed for large computer clusters. This paper presents a flexible solution that is deployable also on small clusters. The solution is based on the allocation and dynamic re-adjustment of indexing and querying roles to cluster nodes in order to optimize cluster utilisation. By allocating cluster machines to the job that requires the most computational power, indexing and querying may both realize performance gains, while neither overwhelms the limited resources available. A prototype system was built and tested on a small cluster using a dataset of over 100 000 Web pages from the uct.ac.za domain. Initial results confirm an improved system resource utilisation, which warrants further investigation.

Categories and Subject Descriptors

C.4 **[Performance of Systems]:** H.3.4 **[Information Storage and Retrieval]:** Systems and Software H.3.5 **[Information Storage and Retrieval]:** On-line Information Services

General Terms

Design, Performance

Keywords

Indexing, querying, small search engine cluster, dynamic allocation.

1. INTRODUCTION

Cluster computing is a popular underlying architecture for modern production search engines, such as those employed by Google and Yahoo!. While popular, clusters are not necessarily the best technology for such problems, as the data inversion involved in creating search engine indices is not easily parallelisable [5]. However, the price-performance index makes clusters an attractive choice, given the massive quantities of information and massive numbers of requests processed by such Web search engines.

Assuming that a cluster is the architecture of choice, computation must be distributed among the individual machines. Production Web search engines may divide both the processing and data among individual machines, with either a static assignment of processes to processors, an on-demand task allocation or some combination of these approaches. The static assignment approach works well in large clusters where some nodes can be dedicated to indexing new data while other nodes serve queries. In this case, changing the task performed by a single computational node does not have a major impact on the whole system.

In a smaller cluster, with possibly fewer users and possibly less data, this is not the case. The role of a single node (indexing or querying) may have a substantial impact on overall performance and resource utilisation. An obvious choice may be to have all nodes perform both indexing and querying tasks, but this may result in problems because of the small number of nodes. Firstly, the disk access operations of indexing and querying tasks typically do not follow similar patterns, thus caching can be sub-optimal if a node is interleaving indexing and querying operations. Secondly, in a smaller cluster, one operation can easily swamp the cluster, making it difficult for the alternative operation to execute to completion. For example, if a large amount of data needs to be indexed, all nodes could be heavily loaded, and an incoming query will take much longer to process. If some resources or nodes could be reserved for each operation, based on the current need for indexing and querying tasks, both of these problems may be suitably dealt with. This thus is the premise of this paper – that nodes in a small cluster search engine could be assigned a particular role, dynamically adjusted for changing loads, in order to best utilise available resources while obtaining the benefits outlined above.

The rest of this paper contains a brief discussion of core search engine concepts, followed by the design and evaluation of the dynamic role search engine, ending with a discussion of the implications and how these relate to other and future efforts.

2. SYSTEM DESIGN

2.1 Introduction to Search Engines

Most practical search engines are based on a common architecture with a set of key components, namely: the Crawler, the Local Store, the Indexer and the Query modules. This architecture is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAICSIT 2007, 2 - 3 October 2007, Fish River Sun, Sunshine Coast, South Africa

Copyright 2007 ACM 978-1-59593-775-9/07/0010…$5.00

used by systems such as Google [4] and FAST [16]. The relationships among various components are shown in Figure 1.

A Crawler is a component that recursively downloads pages from the Web by following hyperlinked URLs to create a local copy of part of the Web.

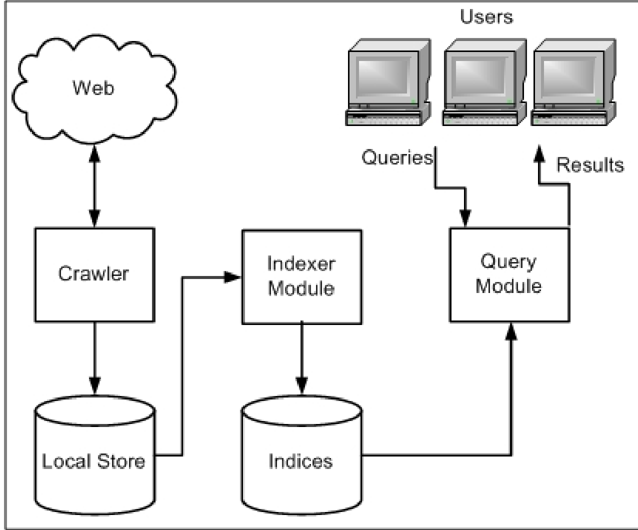

Figure 1. Common search engine architecture

The Local Store is a snapshot of the Web at a given crawling time for each document. These components are not necessarily present if the search engine is not based on Web documents. The Indexer module records which words appear in each document. For each encountered word, the indexing system maintains a set of URLs or identifiers that the word is relevant to, possibly along with other positional information regarding the individual occurrences of words. These indices must be kept in a format that allows their fast intersection and merging during querying time [9]. Thus the index is typically stored as inverted files. The inverted file for a term is a list of identifiers of documents where the term appears. The Query module accepts search queries from users and performs searches on the indices. The query module ranks the results before returning them to the user, such that the results near the top are most likely to be what the user is looking for.

The algorithms for most of these components are omitted as they are not critical to the discussion that follows, but details can be found in [10][13]. The search engine presented in this paper is made up of the components described above. In particular, the Indexing and Querying subsystems are parallelized using cluster computing, which is introduced in the next section.

2.2 System Overview

The prototype search engine used a cluster of computers to perform the core indexing and querying operations. A cluster in this sense is a collection of interconnected stand-alone computers working together as a single, integrated computing resource. Such a system can provide a cost-effective way to gain fast and reliable services that have historically been found only on more expensive proprietary shared memory systems [2].

The system was implemented in C++ in conjunction with the MPI library for parallel programming. MPI is a standard for distributed memory parallel computation using explicit message passing. The C++ programming language was chosen over Java because C++ has well-established parallel programming libraries. Furthermore, C++ execution speeds are preferable for high performance computing. Before the architecture of the search engine is presented, the dynamic role allocation algorithm is first discussed.

2.3 Dynamic Role Allocation

To illustrate the concept of dynamic allocation, an example that compares dynamic allocation to static allocation is shown in Figure 2. In the example, the parameter 'Files' indicates how many documents need to be indexed and the parameter 'Queries' indicates how many user queries are queued and need processing. The example shows that dynamic allocation changes the number of cluster nodes performing indexing or querying based on the workload. The allocation changes over time as the workloads on the querying and indexing machines change. In this example, the first time step has 2 machines allocated to indexing and 10 machines allocated to querying since there are no files that require indexing and 1000 queries that need responses. However, in the second and third time steps, the number of indexing nodes increases while the number of querying nodes decreases due to an increased number of files to be indexed and a decline in query numbers. The three time steps correspond to a reallocation count of two. The reallocation count is defined as the number of times reallocation of indexing/querying roles takes place during a fixed time period. The reallocation count does not apply to static allocation as node allocation does not change unless it is done manually.

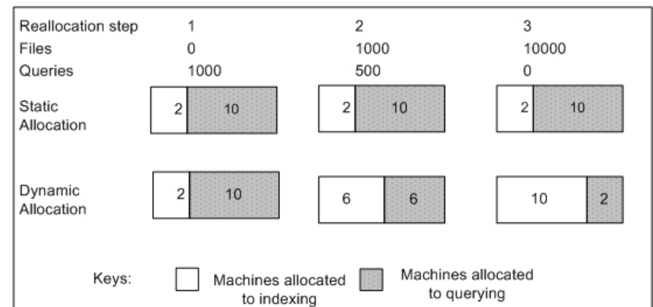

Figure 2. Difference between dynamic and static allocation

2.4 System Architecture

Figure 3 shows the overall architecture of the prototype search engine. The highlighted parts of the diagram collectively make up the Indexing subsystem – the non-highlighted parts show the Querying subsystem. The parts in dotted lines are the interfaces between the two subsystems. The interfaces through which the two subsystems are connected are in the form of inverted index files and a Load Balancer that is independently utilized by each subsystem. These interfaces are described below.

2.4.1 The Interface files

The index is made up of inverted files. The Querying subsystem relies heavily on the index produced by the Indexing subsystem as the former needs to access the index before it can respond to

queries. The id_urls.INFO file contains the ID-to-URL mappings of all the documents that have been indexed by the system. Identifiers (IDs) are used by the indexing system as an efficient way to uniquely identify each indexed document, but the query module needs to respond to user queries with actual URLs.

2.4.2 The Load Balancer

This component monitors the load averages on the nodes allocated to indexing and querying and redistributes roles as necessary. A node's load average is an indication of how much work it has been doing in terms of jobs in the run queue or waiting for disk I/O, averaged over a certain period of time. The UNIX virtual file */proc/loadavg* was used to obtain the load averages on individual nodes. The */proc/loadavg* file includes load average figures giving the number of jobs in the run queue or waiting for disk I/O, averaged over 1, 5 and 15 minutes respectively. The load balancer periodically polls nodes for this information and updates the list of nodes allocated to indexing and querying respectively. For simplicity, this list is stored as the number of machines allocated to indexing – all nodes with a higher node number are assumed to be allocated to querying.

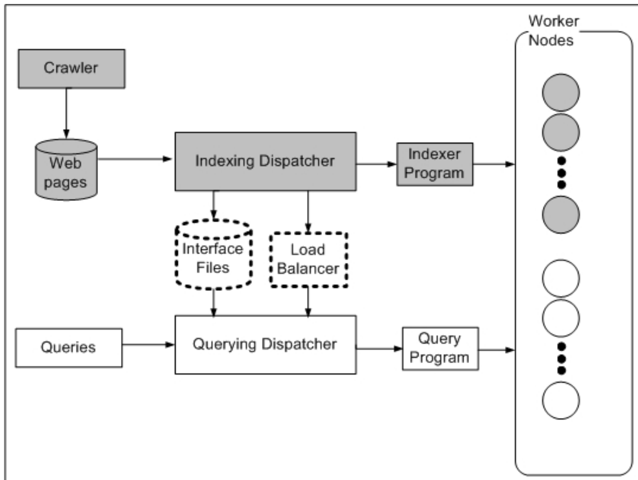

Figure 3. High level architecture of the dynamic role search engine

2.5 The Indexing Subsystem

In order to make the system easy to extend, the indexing subsystem was separated into six main components, namely: the Crawler, the Parser, the Stemmer, the Indexer, the index Updater and the Dispatcher. Parallel indexing was achieved by distributing these components on the cluster as shown on Figure 4. A master-slave approach was used to achieve parallel indexing. The idea behind this approach is that one process, the master, is responsible for coordinating the work of others, the workers. This mechanism is particularly useful when there is little or no communication among the slave processes and when the amount of work that each slave has to perform is difficult to predict [8]. Both of the above cases apply to the task of indexing.
.

The Crawler and Dispatcher components are executed by the machine with the smallest internal identifier within the cluster, which henceforth assumes the role of the master node. The documents are stored on the local disk of the master node. The Indexer and Updater are executed by all machines allocated to indexing at a particular point in time. These machines are the worker nodes. All worker nodes create indices on their local disks which are merged by the Dispatcher to create the main index. The Indexer and Updater components parse and index the HTML documents that are made available by the crawler. The Indexer module creates an index from scratch whereas the Updater module updates an existing index based on newly available data since the last time indexing was performed. Extremely common words (stop-words such as "the" and "is") are excluded from indexing and all terms are case-folded to lower case. In addition, all terms are converted to canonical root forms using Porter's stemming algorithm [15]. The indexing subsystem employs an existing open-source crawler, GNU Wget, a non-interactive command line tool for retrieving files using HTTP, HTTPS and FTP [7].

2.6 The Querying Subsystem

The querying subsystem receives queries from users as a string of keywords that represent the information needs of a user. These queries are fed through the user interface to the querying dispatcher for processing. Once they reach the dispatcher, the dispatcher has to decide which machine in the cluster will handle the query. The allocation of machines to querying by the load balancer is consulted for this purpose.

When a cluster machine is chosen to respond to a query, the query is sent off to the machine and the necessary index files are copied over, if necessary. Each query is stemmed and stopped to improve on accuracy. Term occurrence weights for each document from the index files are used to compute the similarity of the document to the request. Once the computation and results are done, a ranked list of documents is sent back to the dispatcher to return to the user.

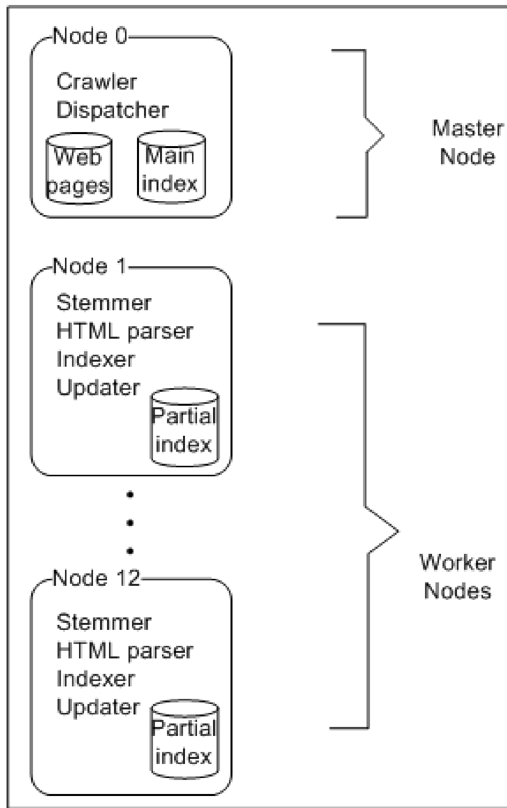

Figure 4. Distribution of the indexing subsystem components on the cluster

3. PRELIMINARY RESULTS

3.1 Experimental Design

Tests were conducted to assess the performance and cluster utilisation of the search engine system. A core aim of the evaluation was to verify that dynamic role allocation results in better cluster utilisation, as the main aim of this project is to improve use of resources in small clusters.

The experiments were conducted on a cluster of 13 Gentoo Linux PCs interconnected by a Gigabit Ethernet network. Of the 13 machines, 12 of the nodes could assume the roles of indexing or querying – the remaining machine was used as the master node. Each PC was equipped with a 3 GHz Pentium 4 processor, 512 MB of RAM and 80 GB disk storage. The MPI implementation on the cluster was LAM MPI version 7.0.6.

3.2 Results

The system was tested to establish how dynamic role allocation affects the utilization of the cluster. Utilization is a measure of how well the load is distributed within the cluster, and may be defined as follows:

$$U = \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{1}{1 + |l_i - \bar{l}|}$$

where n is the total number of worker nodes in the cluster, l_i is the load on node i, and $\bar{l}$ is the average workload on all the nodes n. Thus, if all workloads are equal, U will be equal to 1, but U will have lower values as workloads deviate further from the average l. The workload l_i refers to the per-node workload obtained from the */proc/loadavg* file. The load average figure refers to the number of jobs in the run queue or waiting for disk I/O, averaged over a fixed interval of time.

Figure 5 shows the cluster utilization for indexing operations with increasing datasets. The utilisation is close to 1 independent of dataset size. During indexing of different datasets, a random number of queries were fed to the cluster. The number of queries was varied between 0 and 2824. Each query is handled by a single node in parallel with other nodes which process other queries.

Tests were then carried out to determine how this reasonably balanced utilization affects performance of the indexing and querying subsystems. The indexing subsystem was tested for the effect of the two (static and dynamic) role allocation schemes on the indexing time. Figure 6 shows the results for this test. In this test, dynamic allocation was performed multiple times with different reallocation counts. There are 6 nodes that performed indexing in the static allocation case seen in Figure 6. The number 6 was chosen to assume indexing and querying have equal priority, thus splitting the 12 worker nodes equally between the two roles. The query load was held constant for this test scenario.

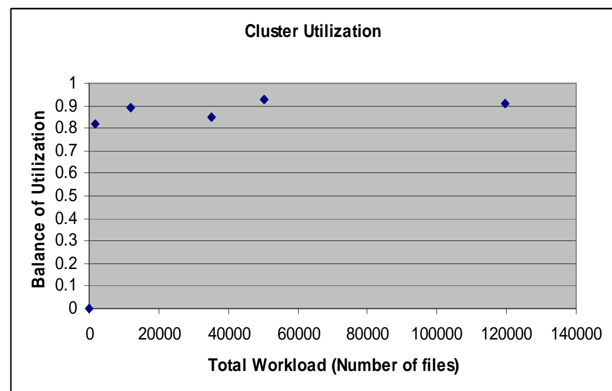

Figure 5. Cluster utilization for different sizes of document collection

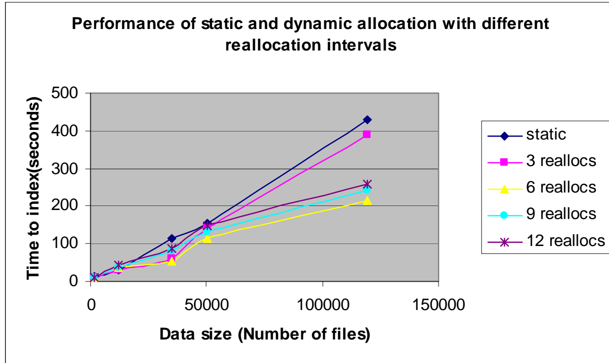

Figure 6. Indexing performance for static and dynamic allocation

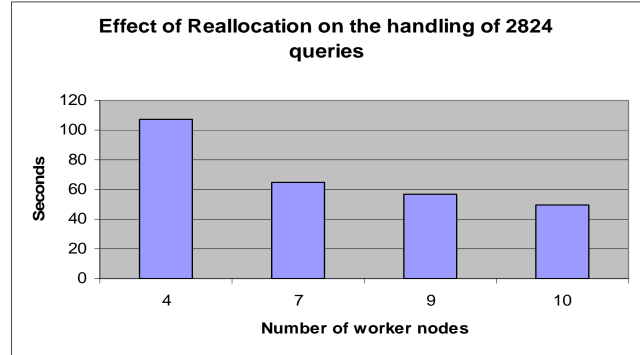

Figure 7. Effect of dynamic allocation on query throughput

From Figure 6 it can be seen that for small data sizes, the time taken to index data for dynamic and static allocations is almost the same. Different reallocation counts result in some performance variance, in particular the smallest number of reallocations (3) resulted in performance similar to the static case. 6 reallocations provided the best performance in this particular test. Comparing the static allocation case with the best case of dynamic allocation (i.e., 6 reallocations), it can be seen that for small data sizes, the time taken to index data for dynamic and static role allocation is almost the same. However, as the size of the data increases, the static allocation performance is significantly worse than that of dynamic allocation. Therefore, with an optimal number and distribution of reallocations, dynamic role allocation can realize shorter indexing times than static allocation, as expected.

The querying subsystem was tested for the effect of dynamic role allocation on query throughput – the total time it takes to respond to a number of queries. Queries of varying lengths were generated randomly by a separate program and written to a file. The querying module then obtained a specified number of queries from this file. Each node executed its own query in parallel with the other nodes. Figure 7 shows the results with 2894 queries where the number of nodes handling querying was dynamically assigned to 4, 7, 9 and finally 10 nodes based on the workload. This decreasing service time confirms that dynamic role allocation can bring into service additional nodes as needed to improve the performance of query processing. It important to note that in this test scenario query response times are affected by the cost of disk access since queries are obtained from disk. In a situation where queries come from the network, which is often the case in practice, response times are likely to be faster since network access is often faster than disk access.

In summary, these experiments have provided some initial evidence that dynamic role allocation can result in scalable system performance and balanced resource utilization, while maintaining the core advantages of such a system as outlined earlier.

4. RELATED WORK

Clusters of low cost workstations are exploited by many large-scale Web search engines such as Google, Inktomi and FAST [6]. The architectures of these search engines require high performance, high scalability, high availability and fault tolerance. It is a challenging task to develop a cluster that meets these requirements. The difficulty is that most developments were done in competitive companies that do not publish technical details, thus very few papers discuss Web search engine architecture.

Orlando et al. [12] describe the design of their cluster-based search engine called My Own Search Engine (MOSE). Their aim is to increase query throughput by implementing an efficient parallelization strategy. MOSE uses a combination data and task parallel algorithm. The task parallel part is responsible for load balancing. It does so by scheduling the queries among a set of identical workers, each implementing a sequential Web search engine. The data parallel part partitions the database, allowing each query to be processed in parallel by several data parallel tasks, each accessing a distinct partition of the database.

Lifantsev and Chiueh [9] describe Yuntis, a working search engine prototype. One of the goals of Yuntis is to utilize clusters of workstations to improve scalability. A Yuntis node runs one database worker process that is responsible for data management of all data assigned to that node. When needed, each node can also perform crawler tasks. Yuntis differs from our system in that the query nodes remain dedicated to responding to user queries. There is no dynamic allocation of nodes to the roles of querying and indexing. If the system is experiencing massive incoming data that needs to be indexed and there are no incoming queries, query nodes will be idle while the indexing nodes will be overloaded. In this case, the cluster will be under-utilized.

The Google search engine architecture [3][4][6] combines more than 15,000 commodity-class PCs with fault-tolerant software. Each of the PCs has 256MB to 1GB of RAM, two 22GB or 40GB disks and runs the Linux operating system. The nodes (PCs) are connected with 100Mbit Ethernet to a gigabit Ethernet backbone [3]. The architecture permits different queries to run on different processors. The index is partitioned into individual segments, thus queries are routed to the appropriate server based on which segment is likely to hold the answer. Our system is different in that it takes into account constrained-resource environments of

small or/and multi-use clusters as opposed to large task–specific clusters inherent in the Google architecture.

The Inktomi search engine architecture serves many Web portals such as Yahoo, HotBot, Microsoft and others. It is a cluster-based architecture utilising RAID arrays with special focus on high availability, scalability and cost-effectiveness. The index is distributed and queries are dynamically partitioned across multiple clusters. Each segment of the database handles a certain set of sub-queries. Queries arrive at the manager where they are directed to selected workers. Each worker sends the queries to all workers that are tightly coupled with it through Myrinet [6].

AltaVista, Lycos and Excite make use of large Symmetric Multi-Processor (SMP) supercomputers. The use of large SMP machines allows fast access to a large memory space. The database is stored and processed on one machine. Processors handle queries independently on the shared database. The disadvantage of such systems is mostly the high cost, that makes them infeasible for smaller organisations.

5. CONCLUSIONS AND FUTURE WORK

Search engines are usually designed for very specific scenarios – Web search engines in particular deal with large numbers of requests and large quantities of data. The architectures of these systems do not always scale down and it is not usually possible to run a flexible search engine in an environment where resources are limited and maximum utilisation is a key concern, such as at institutions in developing countries.

This paper has presented a possible resource utilisation maximisation approach that retains scalability, and is aimed at smaller operations where changes in the actual resources can have a substantial impact on system performance. The initial experimental results indicate that resources are being utilised effectively and that there is some degree of scalability in both the indexing and querying operations, while in all experiments some resources are always dedicated to handling incoming tasks. More experiments are needed to further verify the initial results and to prove that this approach works well with differing workloads and scales as nodes are added to or removed from the system.

In general, systems for handling large quantities of data must work at all scales of systems, not just for large numbers of nodes, and not restricted to only search or information retrieval operations. This ultimately supports a de-centralisation of search operations and other services and will empower users in all countries to provide interesting services with limited, but well-utilised, computing resources. At the very least, everyone can and should have their own little Google-like system based at their organisation, so searching in an internal organisation does not have to be effected through an external service provider as is currently the norm.

6. REFERENCES

[1] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke and S. Raghavan. Searching the Web. *ACM Transactions on Internet Technology*, 1(1): 2-43, 2001.

[2] M. Baker and R. Buyya. Cluster computing at a glance. In *Rajkumar Buyya, editor, High Performance Cluster Computing, volume 1, Architectures and Systems*, Chapter 1. pp. 3-47. Prentice Hall, 1999.

[3] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro,* IEEE, 23(2):22-28, 2003.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107-117, 1998.

[5] F. Cacheda, V. Plachouras and I. Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *Information Processing and Management*, 41(5): 1141-1161, 2005.

[6] B. Choi and R. Dhawan. Distributed Object Space Cluster Architecture for Search Engines. High Availability and Performance Workshop. 2003.

[7] Free Software Foundation, Inc. GNU Wget. Available at: http://www.gnu.org/software/wget/. 2006.

[8] W. Gropp, E. Lusk and A. Skjellum. Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, 1994.

[9] M. Lifantsev and T. Chiueh. Implementation of a Modern Web Search Engine Cluster. In *Proceedings of USENIX Annual Technical Conference*, pp. 1-14, 2003.

[10] N. Nakashole. *A Dynamic Query/Index Role Search Engine*. Honours Project Report, Department of Computer Science, University of Cape Town.

[11] A. Ntoulas and J. Cho. What's New on the Web? The Evolution of the Web from a Search Engine Perspective. In *Proceedings of the 13th International Conference on World Wide Web*, pp.1-12, 2004.

[12] S. Orlando, R. Perego and F. Silvestri. Design of Parallel and Distributed Web Search Engine. In *Proceedings of the 2001 Parallel Computing Conference*, 97-204, 2001.

[13] C. Pedzai.. *A Dynamic Query/Index Role Search Engine*. Honours Project Report, Department of Computer Science, University of Cape Town. 2006.

[14] G.F. Pfister. In search for clusters: The ongoing battle in lowly parallel computing. Prentice Hall 1998.

[15] M. Porter. The Porter Stemming Algorithm: Available at: http://www.tartarus.org/martin/PorterStemmer/ . 2006.

[16] K.M. Risvik and R. Michelsen. Search Engines and Web Dynamics. *Computer Networks,* 9(3): 289-302, 2002.

[17] C.S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham and F. Sommers. Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers. In *A. Y. Zomaya, editor, Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies*, chapter 16, pp 521-551. 2006.