Performance Analysis of a Distributed Question/Answering System

Mihai Surdeanu, Dan I. Moldovan, and Sanda M. Harabagiu
Language Computer Corporation
and
Department of Computer Science
University of Texas at Dallas
mihai@languagecomputer.com
{moldovan, sanda}@utdallas.edu

Abstract

The problem of question/answering (Q/A) is to find answers to open-domain questions by searching large collections of documents. Unlike information retrieval systems, very common today in the form of Internet search engines, Q/A systems do not retrieve documents, but instead provide short, relevant answers located in small fragments of text. This enhanced functionality comes with a price: Q/A systems are significantly slower and require more hardware resources than information retrieval systems. This paper proposes a distributed Q/A architecture *that: enhances the system throughput through the exploitation of* inter-question parallelism *and* dynamic load balancing, *and reduces the individual question response time through the exploitation of* intra-question parallelism. *Inter and intra-question parallelism are both exploited using several scheduling points: one before the Q/A task is started, and two embedded in the Q/A task.*

An analytical performance model is introduced. The model analyzes both the inter-question parallelism overhead *generated by the migration of questions, and the* intra-question parallelism overhead *generated by the partitioning of the Q/A task. The analytical model indicates that both question migration and partitioning are required for a high-performance system: intra-question parallelism leads to significant speedup of individual questions, but it is practical up to about 90 processors, depending on the system parameters. The exploitation of inter-task parallelism*

Q.8	What is the name of the rare neurological disease with symptoms such as : involuntary movements (tics), swearing, and incoherent vocalizations (grunts, shouts, etc)?
Answer (short)	*... who said she has both* **Tourette's Syndrome** *and...*
Q.34	Where is the actress Marion Davies buried?
Answer (short)	*... from the fountain inside* **Hollywood Cemetery***...*
Q.73	Where is the Taj Mahal ?
Answer (long)	*... list of more than 360 cities throughout the world includes the Great Reef in Australia, the Taj Mahal* **in India**, *Chartre's Cathedral in France, and Serengeti National Park in Tanzania. The four sites Japan has listed include...*
Q.176	What is the nationality of Pope John Paul II ?
Answer (long)	*... stabilize the country with its help, the Catholic hierarchy stoutly held out for pluralism, in large part at the urging of* **Polish-born** *Pope John Paul II. When the Pope emphatically defended the Solidarity trade union during a 1987 tour of the...*

Table 1. Examples of answers returned by FALCON. The answers are in bold within 50 or 250 bytes of text.

provides a scalable way to improve the system throughput.

The distributed Q/A system has been implemented on a network of sixteen Pentium III computers. The experimental results indicate that, at high system load, *the dynamic load balancing strategy proposed in this paper outperforms two other traditional approaches. At* low system load, *the distributed Q/A system reduces question response times through task partitioning, with factors close to the ones indicated by the analytical model.*

Keywords: *distributed question answering, load balancing, migration, partitioning*

1 Introduction

1.1 Problem of Question/Answering

The task of a question/answering (Q/A) system is to find an answer to a question by searching a collection of documents and returning *only* the small fragment of text where the answer is located. Table 1 presents output examples of the Falcon Q/A system developed at SMU [16]. The length of the answer text is either 50 bytes (for short answers) or 250 bytes (for long answers). Both the questions and the answer format in Table 1 are from the Text REtrieval Conference (TREC) competition, the de facto standard in Q/A system evaluation [34]. Q/A systems can be extended to handle various problems, but currently their main applicability is to find answers to factual questions, whose answer fits in a small portion of text. The first two editions of the TREC competition

(1999 and 2000) focused on such questions, some of which are exemplified in Table 1. The answers shown in Table 1 all center around entities that can be determined from the question semantics. For example, the answer to the first question is centered around a DISEASE entity, the answers to the next two questions center around LOCATION entities, and the answer to the last question focuses on NATIONALITY entities.

The example previously presented certainly minimizes the complexity of a question answering system, but its simplicity highlights the main concepts used in a Q/A architecture: a Q/A task starts with *the processing of the natural language question*. The purpose of this subtask is two fold. The first goal is to extract semantic information used in the identification of the answer. An example of semantic information is the type of the answer entities previously introduced. The second goal is to select the keywords for document retrieval. *Document retrieval* based on the selected keywords is the second subtask of a Q/A system. Finally, the *answer extraction* subtask identifies the answer within the documents retrieved, by matching the semantic information retrieved from the question with the document contents. This novel concept of answer extraction from documents makes Q/A systems the ideal solution for many real-world applications, such as search engines or automated customer support.

1.2 Motivation

The work presented in this paper builds upon the previous work in question answering systems performed at Southern Methodist University [27, 26, 16]. At the only two editions of the TREC competition that had a Q/A track (TREC-8 in 1999 and TREC-9 in 2000), the Q/A systems developed at SMU were ranked first. The distributed Q/A system introduced in this paper modifies Falcon, the most recent Q/A system developed at SMU. At the TREC-9 competition Falcon provided correct short answers to 66.4% of the questions and correct long answers to 86.1% of the questions, more than any other of the 78 systems in the competition. Nevertheless, the focus of the TREC competitions (and implicitly of the Falcon architecture) was on qualitative results, not on the quantitative system performance. This paper addresses the quantitative performance issues of Q/A systems and is motivated by the following observations:

- Sequential question answering systems, and in particular Falcon, are not good candidates for *large-scale Q/A systems*. In the TREC-9 competition, for a 3 GB document collection, the average question is answered by Falcon in 94 seconds. The average task requires over 20 MB of dynamic memory for internal data. Over four simultaneous questions cause disk overload yielding significant slow down. These observations indicate that the throughput of a sequential question answering system is quite limited. The only solution to increase the throughput of a Q/A system is a distributed architecture where *inter-question parallelism* is exploited by allocating questions on all processors part of the system.

- Typical distributed approaches that exploit only the inter-task parallelism are not sufficient. *Interactive Q/A systems* require that the individual question answering time be significantly reduced. Hence more complex designs that offer both *inter-task* and *intra-task parallelism* are needed for this problem.

1.3 Approach

The novel issues addressed in this paper are the following: (i) A distributed architecture for question/answering, which is a fast growing area of research and commercial interest, is introduced. (ii) The architecture presented exploits both inter and intra task parallelism which improves both the overall system throughput and the individual question response time. (iii) The distributed Q/A system uses a dynamic load balancing model which outperforms traditional runtime scheduling techniques. (iv) The distributed Q/A architecture is analytically modeled and results are presented for large number of processors.

1.4 Related Work

The distributed question/answering problem addresses two distributed processing issues: *dynamic scheduling and load balancing* and *distributed information retrieval*.

Several distributed load balancing models were proposed in the literature: the gradient model [23, 25, 28], sender or receiver-initiated diffusion [35, 31, 4], the hierarchical balancing model [35], and others. Practical implementations of such models vary based on the resources considered for scheduling: CPU [10, 12, 18, 20, 22], memory [1, 33, 37, 36], or combinations of I/O, CPU and memory [3, 2, 11]. Of particular interest for us are load balancing systems used for distributed Internet services. Attempts were made to improve the performance of by now ubiquitous search engines and digital library interfaces [3, 2] using distributed processing techniques, in order to sustain very high loads.

We exploit the inter-task parallelism in a manner roughly similar with the SWEB project [3, 2], but we enhance their model with more scheduling points and intra-task parallelism. On the other hand, the exploitation of intra-task parallelism in the distributed Q/A system is roughly similar to the Piranha project [10]. The novelty of our approach is the seamless integration of both inter and intra task parallelism and its application to question answering.

Distributed information retrieval systems have recently received enhanced attention due to the Internet informational explosion. Hundred of gigabyte collections of documents are now common, making indexing and retrieval impractical on a single computer. The Text Retrieval Conference (TREC) [29] has now a separate section for collections exceeding 100 GB [19]. The experience gained from the TREC very large collection track fostered further research in the field of distributed information retrieval [6, 7]. An interesting result obtained in [7] is a query time evaluation heuristic based on the number of query terms and their frequencies in the given collection. Such information could be used by the load balancing mechanism, but unfortunately it does not apply to question/answering. As indicated in Section 2, the additional modules required by a Q/A system dominate the execution time.

1.5 Overview

Section 2 performs an analysis of the sequential Q/A system. Section 3 presents the architecture of the distributed Q/A system. Section 4 presents the most relevant issues related to the implementation of the inter and intra-question parallelism. Section 5 introduces an analytical model of the proposed distributed Q/A architecture. Section 6 presents empirical test results. Section 7 concludes the paper.

2 Analysis of the Sequential Q/A Architecture

We start with a parallelism analysis of the Falcon Q/A system. The goals of this analysis are: (i) to identify all the levels of parallelism available in a Q/A system, and (ii) to locate the performance bottlenecks in the sequential Q/A design.

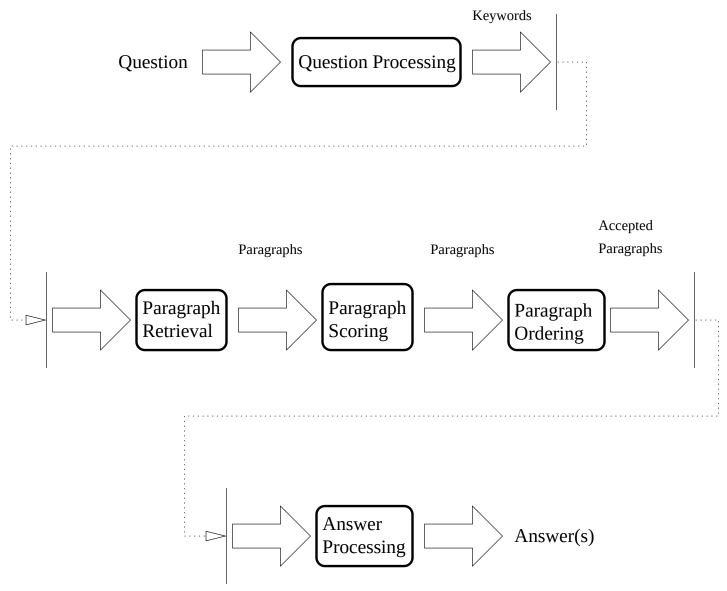


Figure 1. Sequential Q/A Architecture

2.1 Architecture Overview

Figure 1 shows the architecture of the Falcon Q/A system. The presentation is at the block diagram level to emphasize generality. While the focus of this paper is not on the sequential Q/A architecture, for completeness we describe shortly each module part of the Q/A task.

The main role of the *Question Processing (QP)* module if to identify the answer type expected (i.e. `LOCATION`, `PERSON`, etc.) and to translate the user question into a set of keywords to be used in the next processing stages. The *Paragraph Retrieval (PR)* uses a Boolean Information Retrieval (IR) system to identify and extract the documents that contain the previously identified keywords and an additional post-processing phase to extract paragraphs from documents. Falcon currently uses a Boolean IR system, hence documents and paragraphs are not ranked after the PR phase. Even if documents were ranked by the IR system, the next two stages in the Q/A architecture are necessary, because the extracted paragraphs may have different relevance than their parent documents. The *Paragraph Scoring (PS)* module assign a rank to each paragraph provided by the PR module using three surface-text heuristics. The heuristics estimate the relevance of each paragraph based on the number of keywords present in the paragraph and the inter-keyword distance [27]. The *Paragraph Ordering (PO)* module sorts the paragraphs in descending rank order. In order to improve response time, only the paragraphs with a rank over a certain threshold are passed to the next stage. The task of the *Answer Processing (AP)* module is to extract and rank answers from the paragraphs fed by the PO module. Answer processing starts with the identification of *candidate answers* within paragraphs. Candidate answers are lexico-semantic entities with the same type as the question answer type. Around the candidate answers the system builds answer windows, which are text spans that include the candidate answer and one of each of the question keywords. Each window is assigned a score which is a combination of seven heuristics [27]. The heuristics used by the AP module use similar frequency and distance metrics as the PS heuristics. The difference is that the AP heuristics require the presence of a candidate answer. The windows are then sorted in descending order of their score. Further design and implementation details of the sequential Falcon Q/A system are available in [16, 26].

2.2 Timing Analysis

The complexity of the Q/A system is reflected in the question processing times. We measured the average question answering time using two versions of the Falcon Q/A system, developed for the TREC-8 and TREC-9 competitions. Falcon's average question answering time was 48 seconds on the TREC-8 collection and 94 seconds on the TREC-9 collection, at least an order of magnitude larger than a typical IR engine. The Table 2 analysis of the Q/A modules indicates that the bottlenecks of the Q/A system are the *paragraph retrieval* (PR) and the *answer processing* (AP) modules. The overhead of the bottleneck modules becomes more significant in the TREC-9 competition: the PR time increases with the collection size (an average of 21 seconds per question for the 2 GB TREC-8 collection versus an average of 24 seconds for the 3 GB TREC-9 collection). The larger number of paragraphs and the different module design increased the complexity of the AP module from an average of 23 seconds per question in TREC-8 to 65 seconds per question in TREC-9. In both tests, the advanced natural language processing techniques used in the AP module (i.e. named-entity recognition for the detection of candidate answers and syntactic parsing for the construction of answer semantic representations) are considerably more computational expensive than the light-weight techniques used in the paragraph scoring module.

The last column of Table 2 indicates that three Falcon modules (PR, PS, and AP) are iterative with various granularities: *document collection* for PR, and *text paragraph* for PS and AP. For our experiments we used a logical separation of the TREC-9 document collection into eight sub-collections, but arbitrarily small sub-collections can be built. The results presented in Table 2 lead

	TREC-8	TREC-9		
Module	% of Task Time	% of Task Time	Iterative Task?	Granularity
QP	1.1 %	1.2 %	No	
PR	44.4 %	26.5 %	Yes	Collection
PS	5.4 %	2.2 %	Yes	Paragraph
PO	0.1 %	0.1 %	No	
AP	48.7 %	69.7 %	Yes	Paragraph

Table 2. Analysis of Q/A modules

us to the following observations:

- A distributed Q/A system with specialized scheduling points before the computation-intensive modules (PR and AP) should achieve efficient load balancing. For the Q/A architecture, the requirements of the bottleneck modules are precisely identified: the PR module extensively uses the disk with minimal CPU load, while the reverse is true for the AP module. Based on this information one can design simple and effective load balancing heuristics specialized for each bottleneck module.
- Having independent scheduling points for each of the bottlenecks is feasible because the inter-module communication is minimal. The PR module takes as input the current set of keywords and produces a set of paragraphs. The set of documents produced by the internal information retrieval system (typically very large) is not visible outside of the module. A reduced set of paragraphs is input to the AP module which returns a set of answers considered correct.
- Another advantage of having separate scheduling points for the bottleneck modules is that each bottleneck module can be distributed between multiple processors. Task partitioning is possible because, as Table 2 shows, all bottlenecks are iterative tasks. Using partitioning, or intra-task parallelism, the response time for individual questions can be improved, which is not possible for a system that only takes advantage of the inter-question parallelism.

These observations indicate that a distributed Q/A system that combines inter-question with intra-question parallelism will achieve not only best overall throughput but also best individual question response times. The design presented in the next section follows this idea. The results shown in Section 6 prove that our intuition was correct.

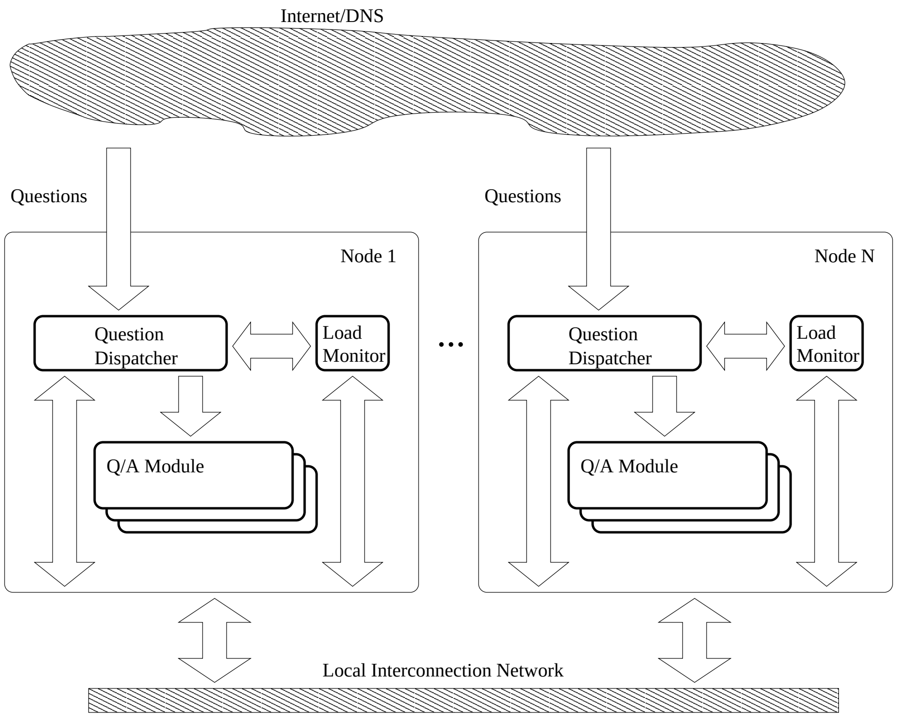


Figure 2. High Level Distributed Q/A Architecture

3 Distributed Question Answering Architecture

Based on the analysis presented in the previous section, the design of the distributed Q/A system can be presented in two stages: a high-level design that illustrates inter-question parallelism and a low-level design that emphasizes intra-question parallelism. The goals when designing the distributed Q/A system were: (i) *performance*: the system should improve overall throughput through dynamic load balancing and should minimize question response times through the exploitation of intra-question parallelism; (ii) *scalability*: the design should avoid hot points and single points of failure; and (iii) *flexibility*: processors must be able to dynamically join or leave the system pool.

3.1 Inter-Question Parallelism

A high-level view of the distributed Q/A system architecture is presented in Figure 2. The inter-question parallelism is exploited through distributed scheduling. The Q/A system is designed as an Internet service. Hence we assume that an initial distribution of questions among processors is already performed by the Domain Name Service (DNS). Using the one-to-many name to IP address mapping, requests are mapped to system processors in a round-robin manner [5]. In practice, load balancing using this strategy is far from perfect because DNS does not consider the system load information when doing the request redirection. Furthermore, even the round-robin assumption is not always true: due to DNS address caching, requests from the same net are directed to the same IP address for the lifetime of the cache. The question dispatcher performs additional rescheduling to compensate for the DNS limitations. If the DNS-allocated node is over-loaded, the dispatcher migrates the Q/A task to another node. The question dispatcher migration is performed before the task is started. In the next subsection, we describe two other dispatching points (before paragraph retrieval and before answer processing), which are capable of migrating a Q/A task during its execution. The dispatcher's strategy is to select the processor with the smallest *average load* for the Q/A task. To avoid useless migrations, a question is migrated only if the difference between the load of the source node and the load of the destination node is greater than the average workload of a single question. For every processor i, the Q/A task average load is defined as the weighted average of the required resource loads:

$$loadFunction_{QA}(i) = W_{QA}^{DISK} load_{DISK}(i) + W_{QA}^{CPU} load_{CPU}(i) \tag{1}$$

The resource weights indicate how significant the corresponding resource is for the Q/A task. In practice, the resource weights are equal with the percentage of the execution time the Q/A task spends accessing the corresponding resource. The measurement strategy for the resource weights and their values for our test platform are presented in Section 4.

The distributed load management is implemented by the set of load monitoring processes. Periodically each load monitor updates its local CPU and disk load and broadcasts the information on the local interconnection network. Thus every processor is aware not only of its own load but of the load of every other active processor in the system. This broadcast mechanism is also used to keep track of the current system configuration: if load information is not received from a processor in a predefined time, that processor is removed from the system pool. A processor automatically joins the pool when it starts broadcasting load information on the local network.

3.2 Intra-Question Parallelism

Figure 2 shows each question answering task being handled by a Q/A module. Figure 3 expands the internal architecture of the Q/A module. The design is based on the Q/A architecture presented

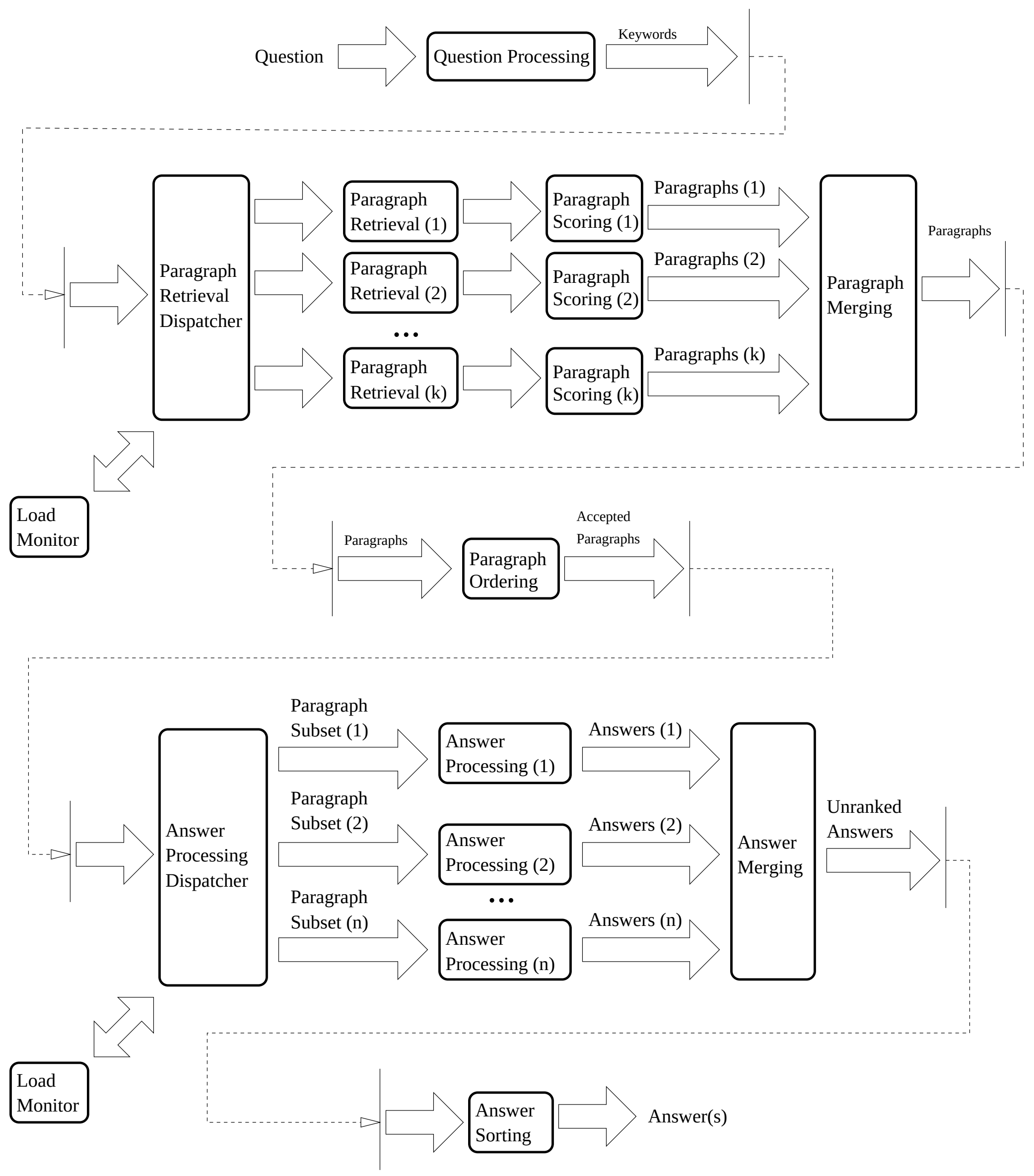


Figure 3. Low Level Distributed Q/A Architecture

in Figure 1, extended along the guidelines established in the first part of this section: bottlenecks are divided into lower granularity subtasks which are scheduled with heuristics specialized for each subtask. With this architecture we aim to provide better overall load balancing and improve the individual question answering time. Several modules were added to the design from Figure 1: scheduling points before the PR and AP modules, input splitting and output merging/sorting where necessary. The new modules are detailed next.

Specialized dispatchers were added before the PR and AP modules. Both dispatchers use the *meta-scheduling algorithm* presented in Figure 4. The meta-scheduling algorithm attempts to di-

```
metaScheduler(task, loadFunction, underloadCondition)
  1. select all processors $i$ with $underloadCondition(i)$ true
  2. if none selected then
     select processor $i$ with the smallest value for $loadFunction(i)$
  3. assign to each selected processor $i$ an unnormalized weight:
     $uw_i = \frac{\max_{i=1}^{n}\{loadFunction(i)\}}{loadFunction(i)}$, where $\max_{i=1}^{n}\{loadFunction(i)\}$ is the largest value
     observed for the load function in the selected processor set
  4. assign to each selected processor $i$ a normalized weight
     $w_i = \frac{uw_i}{\sum_{j=1}^{n} uw_j}$, where $n$ is the number of selected processors
  5. assign to each selected processor $i$ a fraction $w_i$ of the global task
```

Figure 4. Meta Scheduling Algorithm

vide a given task into smaller granularity sub-tasks and distribute them on the processors best fit for the task. The meta-scheduling algorithm has three parameters: the task to distribute, the load function, defined on each processor as a quantification of unavailable task-specific resources, and the under-load condition for the given module. The goal of the meta-scheduling algorithm is to automatically determine the degree of intra-parallelism available in the current system state. This is the justification for the first algorithm step, which attempts to select all under-loaded processors. If under-loaded processors are available in the system, intra-question parallelism can be exploited because there are distributed resources not used to their full potential. If there are no under-loaded processors, there is no reason to force intra-question parallelism. In such a case, the second step of the algorithm selects a single processor, which has the most available resources for the desired task. The third and fourth algorithm steps assign to each processor a weight according to the percentage of available resources on this processor. The task to be dispatched is then partitioned and allocated on all processors selected according to their normalized weights. Section 4.1 introduces three partitioning strategies tailored for the PR and AP modules.

The meta-scheduler is specialized for the PR and AP modules with different load functions and under-load conditions. For every processor i the load functions for the PR and AP sub-task are defined as:

$$loadFunction_{PR}(i) = W_{PR}^{DISK} load_{DISK}(i) + W_{PR}^{CPU} load_{CPU}(i) \qquad (2)$$

$$loadFunction_{AP}(i) = W_{AP}^{DISK} load_{DISK}(i) + W_{AP}^{CPU} load_{CPU}(i) \qquad (3)$$

Similarly with the resource weights for the whole Q/A task, the weights for the PR and AP load functions are empirically determined by examining the average percentage of the module execution time spent accessing the corresponding resource. The procedure used to measure the resource weights and their values for out test platform are presented in Section 4. The measured weight values indicate that the most significant resource for PR module is the disk (on our test platform 80% of the paragraph retrieval time is spent on I/O accesses), and the most significant resource for the AP module is the CPU (virtually 100% of the answer extraction time is spent in non-idle CPU cycles). In the same section we present the implementation of the under-load condition for both the PR and AP modules. The intuition behind this approach is that each dispatcher should select the processors with the highest availability for the resources used most extensively by the dispatched task (disk for PR and CPU for AP). The results presented in Section 6 validate our approach.

Due to the partitioning of the PR and AP tasks, three new modules are introduced: *paragraph merging*, which collects the paragraphs output by the paragraph scoring modules, *answer merging*, which collects the answers extracted by the answer processing modules, and *answer sorting*, which sorts the local best answers obtaining a global order. The centralized paragraph merging and ordering are justified by the filter at the end of the paragraph ordering module, which accepts only paragraphs ranked over a predefined threshold. Because one of our goals was to mimic the execution of the sequential system, the ranking and filtering of paragraphs had to be centralized in order to obtain the same paragraphs at the output of the PO module as the sequential system.

4 Implementation

This section presents implementation issues which were not covered by the conceptual model presented in Section 3, but are nevertheless important parts of the proposed distributed Q/A architecture. The following issues are addresses: (i) implementation of the paragraph retrieval and answer processing partitioning strategies; and (ii) measurement methodology for the empirical parameters used by the scheduling heuristics.

4.1 Implementation of Intra-Question Parallelism

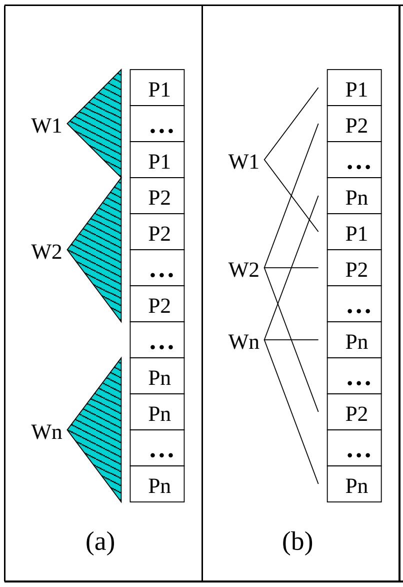
This subsection describes the implementation of the intra-question parallelism model introduced in Section 3. More exactly, we describe practical solutions for the implementation of Step 5 of the meta-scheduling algorithm presented in Figure 4. This part of the meta-scheduling algorithm partitions the given module (PR, PS, or AP) on the processors selected in the previous steps. This task raises two practical problems: first, the module input must be partitioned such that the granularities of the generated partitions are as close as possible to the processor weights reported by the load balancing module, and second, the allocation algorithm must implement a recovery strategy from node failures.

This section introduces three possible partitioning algorithms, classified in two classes: sender-controlled algorithms and receiver-controlled algorithms. All algorithms make the assumption that the input task is iterative. As Section 2 indicates, this assumption is true for the PR, PS, and AP modules. For simplicity, throughout the paper we use the term *item* to identify the smallest granularity of the input data (collection or paragraph), the term *sub-task* to identify the processing associated with one item, and *sub-task granularity* to indicate the sub-task resource requirements (CPU and/or IO).

4.1.1 Sender-Controlled Partitioning Algorithms

The first algorithm presented makes the assumption that the sub-task granularity does not vary widely among items. The input data is divided into successive partitions with sizes computed according to the selected processor weights. Each partition is then migrated to the corresponding processor. This partitioning strategy is illustrated in Figure 5 (a). We named this partitioning algorithm SEND, because it is a straight-forward implementation of the sender-initiated approach. Figure 5 (a) shows the partitioning of the input array for n processors, where partition i receives the next W_i consecutive items. The weights W_i are computed in the first 5 steps of the meta-scheduling algorithm presented in Figure 4.

The next partitioning algorithm does not make the assumption that sub-tasks have the same granularity. Instead, this algorithm makes the less restrictive assumption that, while the sub-task granularity varies, the input data is an array sorted in descending order of the sub-task granularities (this assumption holds only for the AP module, where the input paragraphs are sorted in descending order of their relevance to the question). Given this order, partitions are constructed through a round-robin interleaving of items, in an attempt to obtain similar average sub-task granularities across partitions. This strategy is illustrated in Figure 5 (b). Note that each partition i still receives



```
taskDistribution(task, set of partition weights)
1. build partitions using either
   the SEND or ISEND approach
2. for each selected processor $i$ do in parallel
     (a) migrate partition $i$ to processor $i$
     (b) wait task termination
     (c) if successful termination remove
         partition $i$ from the partition set
3. if unprocessed partitions
     (a) build a new task from the unprocessed partitions
     (b) jump to Step 1 of the metaScheduler algorithm
```
(c)

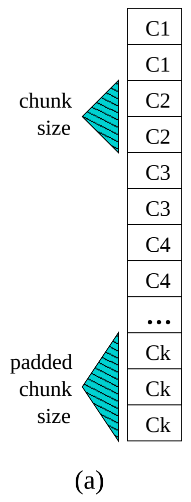
Figure 5. Sender-controlled algorithms: (a) direct partitioning (SEND), (b) interleaved partitioning (ISEND), and (c) common distribution strategy

W_i items, but its items are not necessarily consecutive in the item array. We named this algorithm ISEND, from “interleaved SEND”.

While the SEND and ISEND algorithms offer solutions for the partitioning problem, a complete scheduling algorithm must also handle failure recovery. Because both SEND and ISEND are sender-controlled algorithms, they use the same failure recovery strategy, presented in Figure 5 (c). In the actual implementation, the task distribution algorithm from Figure 5 (c) replaces Step 5 of the meta-scheduling algorithm. The task distribution algorithm presented in Figure 5 (c) allocates in parallel each partition to the corresponding processor and waits for the sub-task termination. The parallel monitoring of the working processors is implemented with one dedicated thread per processor. The failure of a working processor, or of the corresponding interconnection network, is detected through TCP error messages. If one or more of the working processors fail before the allocated partitions are processed, the distribution algorithm builds a new task input by concatenating all unprocessed partitions. The scheduling procedure is repeated until all partitions are processed.

4.1.2 Receiver-Controlled Partitioning Algorithms

The last partitioning algorithm presented makes no assumption about the sub-task granularities. Under these loose conditions the computed processor weights can not be used to approximate the



```
taskDistribution(task, set of partition weights)
1. build the chunk set
2. for each selected processor $i$ do in parallel
    (a) $broken[i] = false$
    (b) loop while (there are chunks available) $\wedge$ ($\neg\, broken[i]$)
        i. remove next chunk from the chunk set
        ii. migrate chunk to processor $i$
        iii. wait task termination
        iv. if unsuccessful termination
            A. move chunk back to the chunk set
            B. $broken[i] = true$
```

(b)

Figure 6. Receiver-controlled algorithm (RECV): (a) partitioning, and (b) distribution strategy

partition sizes. Instead, this algorithm divides the item array into *equal-size chunks*. Theoretically, the chunk granularity can be as small as one item. Figure 6 (a) illustrates the partitioning of the item array into k chunks of size 2, with chunk k extended to include the last item. The processor which generated the chunks initiates the task processing by alerting the selected processors that work is available. Nevertheless, the initiating processor does not make any assumptions about how much work will be allocated to each processor. Each working processor requests and processes one chunk at a time according to its local resource availability, until all chunks are processed. We consider this approach a *receiver-controlled algorithm* (hence the name RECV), because the exact amount of work migrated is decided by the receiving processors. The complete algorithm is presented in Figure 6 (b). The Figure 6 (b) algorithm concurrently assigns chunks to working processors until all chunks are processed. Recovery from a node failure is implemented by storing the unprocessed chunks back in the available chunk set, and removing the failed processor from the working processor set.

The RECV algorithm has two advantages: (i) the differences between sub-task granularities are minimized by using small-size chunks, and (ii) the algorithm provides a better mapping of the remote resource loads by letting the working processors compete with each other for work. The main disadvantage of this algorithm is that, for the AP module, the distribution overhead is inversely related with the chunk size, because a constant number N_A of answers must be extracted from each chunk, where N_A is the number of answers requested by the user. Hence, the smallest

```
time trace QP: 0.42 secs
PR: using 4 nodes: N1, N2, N3, N4
N1 started index trec.ft
N3 started index trec.latimes
N2 started index trec.fbis
N4 started index trec.ap.1
N4 finished index trec.ap.1 in 0.19 secs
N4 started index trec.ap.2
N1 finished index trec.ft in 0.40 secs
N1 started index trec.sjm
N4 finished index trec.ap.2 in 0.21 secs
N4 started index trec.wsj.1
N3 finished index trec.latimes in 0.51 secs
N3 started index trec.wsj.2
N1 finished index trec.sjm in 0.19 secs
N4 finished index trec.wsj.1 in 0.11 secs
N3 finished index trec.wsj.2 in 0.10 secs
N2 finished index trec.fbis in 1.52 secs
PR: all threads done
PR: received 912 paragraphs
PR: paragraph receiving time: 0.23 secs
time trace PR: 1.89 secs
time trace PO: 0.01 secs
AP: selected 4 nodes:
N1 (0.25), N2 (0.25), N3 (0.25), N4 (0.25)
AP: accepted 881 paragraphs
AP: using 4 nodes:
N1 (220), N2 (220), N3 (220), N4 (221)
N1 started 220 paragraphs
N4 started 221 paragraphs
N3 started 220 paragraphs
N2 started 220 paragraphs
N4 finished 221 paragraphs in 39.45 secs
N3 finished 220 paragraphs in 86.14 secs
N2 finished 220 paragraphs in 89.05 secs
N1 finished 220 paragraphs in 99.48 secs
AP: all threads done
AP: answer sorting time: 0.00 secs
time trace AP: 99.66 secs
time trace QA: 101.99 secs
```

(a)

```
time trace QP: 0.43 secs
PR: using 4 nodes: N1, N2, N3, N4
N1 started index trec.ft
N3 started index trec.latimes
N2 started index trec.fbis
N4 started index trec.ap.1
N4 finished index trec.ap.1 in 0.19 secs
N4 started index trec.ap.2
N1 finished index trec.ft in 0.40 secs
N1 started index trec.sjm
N4 finished index trec.ap.2 in 0.21 secs
N4 started index trec.wsj.1
N3 finished index trec.latimes in 0.51 secs
N3 started index trec.wsj.2
N1 finished index trec.sjm in 0.19 secs
N4 finished index trec.wsj.1 in 0.11 secs
N3 finished index trec.wsj.2 in 0.10 secs
N2 finished index trec.fbis in 1.52 secs
PR: all threads done
PR: received 912 paragraphs
PR: paragraph receiving time: 0.23 secs
time trace PR: 1.89 secs
time trace PO: 0.01 secs
AP: selected 4 nodes:
N1 (0.25), N2 (0.25), N3 (0.25), N4 (0.25)
AP: accepted 881 paragraphs
AP: using 4 nodes:
N1 (220), N2 (220), N3 (220), N4 (221)
N1 started 220 paragraphs
N3 started 220 paragraphs
N2 started 220 paragraphs
N4 started 221 paragraphs
N2 finished 220 paragraphs in 78.48 secs
N4 finished 221 paragraphs in 78.68 secs
N1 finished 220 paragraphs in 81.93 secs
N3 finished 220 paragraphs in 83.50 secs
AP: all threads done
AP: answer sorting time: 0.00 secs
time trace AP: 84.56 secs
time trace QA: 87.93 secs
```

(b)

```
time trace QP: 0.41 secs
PR: using 4 nodes: N1, N2, N3, N4
...
time trace PR: 1.94 secs
time trace PO: 0.01 secs
AP: chunk size 80
AP: accepted paragraphs 881
AP: using 11 chunks:
80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 81
AP: using 4 nodes: N1, N2, N3, N4
N1 started chunk 0
N2 started chunk 1
N3 started chunk 2
N4 started chunk 3
N2 finished chunk 1 in 31.89 secs
N2 started chunk 4
N4 finished chunk 3 in 32.59 secs
N4 started chunk 5
N1 finished chunk 0 in 38.77 secs
N1 started chunk 6
N3 finished chunk 2 in 38.67 secs
N3 started chunk 7
N4 finished chunk 5 in 25.99 secs
N4 started chunk 8
N2 finished chunk 4 in 31.69 secs
N2 started chunk 9
N1 finished chunk 6 in 29.41 secs
N1 started chunk 10
N1 finished chunk 10 in 2.98 secs
N3 finished chunk 7 in 35.42 secs
N2 finished chunk 9 in 14.24 secs
N4 finished chunk 8 in 23.92 secs
AP: all threads done
AP: answer sorting time: 0.00 secs
time trace AP: 82.52 secs
time trace QA: 85.91 secs
```

(c)

Figure 7. System traces with RECV for PR/PS and (a) SEND for AP, (b) ISEND for AP, or (c) RECV for AP

the chunk size, the greater the number of answers that have to be sorted after the AP module. In Section 6 we present an empirical experiment which finds the chunk size for which RECV delivers its best performance.

4.1.3 Partitioning Examples

For a better understanding of the partitioning algorithms, we present in this subsection examples of the system behavior under the three partitioning strategies. The examples presented are simplified traces of a homogeneous 4-processor system executing question 226 from the TREC-9 set.

The first trace, presented in Figure 7 (a), uses the RECV partitioning strategy for the PR and PS tasks and the SEND approach for AP. For the clarity of the example, the IP addresses used for node identification are replaced with four generic names: N1, N2, N3 and N4. The question answering task is started on node N1. Figure 7 (a) indicates that the four nodes selected for paragraph retrieval and scoring compete with each other for the 8 sub-collections part of the TREC-9 database. This dynamic strategy is extremely efficient for the PR task where the collection processing time varies significantly: from 0.19 seconds to 1.52 seconds. Figure 7 (a) shows that, while node N2 processes one collection, the other nodes process all other seven collections. Due to these high variances in collection processing times, the weight-based partitioning performed by the sender-controlled algorithms is virtually inapplicable. For this reason, in the actual system implementation the PR task is partitioned only with the RECV strategy. The second part of the Figure 7 (a) shows the timing obtained with the SEND partitioning strategy for the answer processing module. The trace indicates that, even though each node processes the same number of paragraphs, the processing time varies significantly: the node which received the first 220 paragraphs finished last, and the node which received the last 221 paragraphs finished first, more than 60 seconds ahead.

The ISEND algorithm achieves a much better partitioning for the AP module. Figure 7 (b) indicates that all ISEND-generated AP sub-tasks finish within a 6-second interval, and the overall processing time for the AP module is more than 15 seconds shorter than the processing time observed for the SEND algorithm.

The best behavior is observed for the RECV partitioning approach. Due to the fact that processors dynamically compete with each other for work, a more even load distribution is achieved. It is however surprising that, for the AP task, the RECV performance is very close to the ISEND performance, a much simpler algorithm. This is an indication that, even though it was not designed for this purpose, the paragraph ranking performed by the PO module provides also a good ranking of the paragraph processing complexity. In Section 6 we present experiments which analyze in more detail the performance of the three partitioning algorithms.

4.2 Hardware-Dependent Empirical Parameters

This section reviews the hardware-dependent parameters used throughout this paper. These parameters are: *the weights used by the load functions* in Equations 1, 2, and 3, and the *under-load conditions* used by the meta-scheduling algorithm for the partitioning of the PR and AP modules (introduced in Figure 4). Because these parameters are dependent on the underlying hardware, it is important to have a simple measurement methodology in order to guarantee the portability of the system.

	CPU	DISK
QA	0.79	0.21
PR	0.20	0.80
AP	1.00	0.00

Table 3. Average resource weights measured for the TREC-9 question set

We first address the measurement of the load function resource weights. These weights indicate how significant the corresponding resource is for the desired Q/A module, for a particular question set[1]. In practice, the weight associated with the CPU resource[2] is computed as the percentage spent by the CPU in a non-idle state during the module execution. Because the only other resource highly utilized by the sequential Q/A application is the disk, the remaining CPU cycles are assumed to be spent performing I/O accesses. This procedure yielded the resource weights presented in Table 3 for the TREC-9 question set. Each row in Table 3 indicates the resource weights for the corresponding load function. For example, the first row indicates that the average Q/A task spends 79% of the execution time using the CPU, and 21% accessing the disk. Hence the load function used by the question dispatcher on our test platform is:

$$loadFunction_{QA}(i) = 0.21\, load_{DISK}(i) + 0.79\, load_{CPU}(i) \tag{4}$$

Similarly, the load functions for the PR and AP dispatchers become:

$$loadFunction_{PR}(i) = 0.8\, load_{DISK}(i) + 0.2\, load_{CPU}(i) \tag{5}$$

$$loadFunction_{AP}(i) = load_{CPU}(i) \tag{6}$$

The second issue addressed in this section is the implementation of PR and AP under-load conditions required by the meta-scheduling algorithm presented in Figure 4. The under-load conditions are empirical parameters which can be set either to minimize the question response time, or to maximize the throughput. For the experiments presented in this paper we use under-load conditions that favor throughput versus response time. Intuitively, if for each task the number of nodes considered as under-loaded is increased, the individual question response time is improved due to the enhanced partitioning, but the overall system throughput decreases due to the partitioning overhead. In practice, the throughput does not decrease immediately as the degree of partitioning

[1]Dynamic task workload detection strategies such as [11] are not addressed in this paper.
[2]By "CPU resource" we understand the combination of processing unit and dynamic memory.

is increased: for moderate partitioning, the overall throughput is improved due to efficient local scheduling. This is best illustrated by the following experiment: a set of questions is repeatedly executed on a single processor, and at each repetition the number of questions executing simultaneously is increased. This experiment indicated that, on our test platform, for 2 or 3 simultaneous questions the system throughput is better than the throughput obtained by running the questions sequentially, because the operating system reuses one question's I/O wait time as useful CPU time for another question. For more than 4 simultaneous questions the throughput is smaller than the sequential question execution throughput due to two factors: (i) *disk over-utilization* caused by multiple questions executing simultaneous paragraph retrievals, and (ii) *excessive page swapping* caused by the lack of dynamic memory. This experiment is consistent with the results reported in [7], where the best performance for a distributed IR system is obtained with 4 tasks per node. The above experiment indicates that the disk is under-utilized if the node executes a single PR or AP sub-task. In other words, the PR and AP under-load conditions for node i are true if the value of the corresponding load functions for this node are less than the load observed when a single sub-task executes on this node ($threshold_{PR}$ for the PR sub-task and $threshold_{AP}$ for the AP sub-task):

$$underloadCond_{PR}(i) = \begin{cases} true, & if\ loadFunction_{PR}(i) \leq threshold_{PR}, \\ false, & otherwise. \end{cases} \tag{7}$$

$$underloadCond_{AP}(i) = \begin{cases} true, & if\ loadFunction_{AP}(i) \leq threshold_{AP}, \\ false, & otherwise. \end{cases} \tag{8}$$

5 Analytical Performance Analysis

This section analyses the analytical performance of the distributed Q/A architecture introduced in Sections 3 and 4. Two analytical models are introduced: the first model analyzes inter-question parallelism, and the second model analyzes intra-question parallelism. Through this analysis we find: (i) the maximum system speedup obtainable using inter-question parallelism, (ii) the maximum question speedup obtainable using intra-question parallelism, and (iii) the number of nodes from which the overhead introduced by the intra-question parallelism overcomes its benefits.

The following notations are used in this chapter:

- t_N^M, the average time to execute M questions on N nodes;

- N_K, the average number of keywords extracted from a question;
- N_P, the average number of paragraphs obtained by the paragraph retrieval module;
- N_{PA}, the average number of paragraphs accepted after the paragraph ordering stage;
- K_{size}, the average keyword length;
- P_{size}, the average paragraph size;
- N_A, the number of answers per question requested by the user;
- A_{size}, the answer size;
- t_{load}, the average time to measure the local system load;
- L_{size}, the size of the load monitoring broadcast packet;
- Q_{size}, the average question size;
- b_{net}, the network bandwidth;
- b_{disk}, the disk bandwidth; and
- b_{memo}, the local memory bandwidth.

5.1 Inter-Question Parallelism Analysis

In this subsection we compute the maximum speedup obtainable using inter-question parallelism. The assumption made here is that all three dispatching points are used, but partitioning is not enabled due to the high system load. The partitioning overhead is analyzed in the next subsection. In other words, we analyze the system behavior when it executes a large number of questions, and each question may change its hosting processor at any of the three dispatching points: before the Q/A task is started, before the PR module, and before the AP module. Any dispatcher may migrate a question if the associated scheduling heuristic (see Section 3) detects that the runtime of the monitored sub-task is improved if executed to a different processor. The following additional notations are used in this subsection:

- p_{qa}, the probability that a Q/A task is migrated before it is started;
- p_{pr}, the probability that a Q/A task is migrated by the PR dispatcher;
- p_{ap}, the probability that a Q/A task is migrated by the AP dispatcher;
- p_{net}, the probability that a Q/A task accesses the network at any time; and
- R, the average number of questions per processor.

The speedup is computed with the formula:

$$S_N^1 = \frac{t_1^M}{t_N^M} \tag{9}$$

where t_1^M is the time to execute M questions on 1 node, and t_N^M is the time to execute M questions on N nodes. t_1^M and t_N^M are computed as follows:

$$t_1^M = \sum_{i=1}^{M} t_1^1(i) \tag{10}$$

$$t_N^M = \frac{\sum_{i=1}^{M}\{t_1^1(i) + t_{overhead}(i)\}}{N} \tag{11}$$

where $t_1^1(i)$ denotes the time to execute the ith question on a single-processor system, and $t_{overhead}(i)$ is the distribution overhead for the ith question. Thus, the speedup equation can be expanded as:

$$S_N^1 = \frac{N}{1 + \frac{\sum_{i=1}^{M} t_{overhead}(i)}{\sum_{i=1}^{M} t_1^1(i)}} \cong \frac{N}{1 + \frac{t_{overhead}}{t_1^1}} \tag{12}$$

where $t_{overhead}$ denotes the average overhead measured for the M questions, and t_1^1 is the average question execution time on a single-processor system. $t_{overhead}$ is expanded as:

$$t_{overhead} = t_{monitoring} + t_{scheduling} + t_{migration} \tag{13}$$

where: $t_{monitoring}$ is the average overhead of the load monitoring process for the duration of the question; $t_{scheduling}$ is the average overhead of the three dispatchers; and $t_{migration}$ represents the average migration overhead when all three dispatching points are enabled.

Every second, the load monitoring process performs the following tasks: (i) it inspects the kernel to collect information about the local load; (ii) it broadcasts the local load on the interconnection network; and (iii) it stores in memory the local load and load information received from the other nodes. The local load measurement overhead is t_{load}. Theoretically, all nodes broadcast their local load in the same time, hence the network bandwidth is b_{net}/N. Thus, the broadcast time is: $L_{size}\frac{N}{b_{net}}$. The memory storage time is $L_{size}\frac{N}{b_{memo}}$. Because the load monitor repeats these operations every second, the monitoring overhead for the average question becomes:

$$t_{monitoring} = t_1^1(t_{load} + L_{size}\frac{N}{b_{net}} + L_{size}\frac{N}{b_{memo}}) \cong t_1^1 L_{size}\frac{N}{b_{net}} \tag{14}$$

The question dispatcher scans the processor load information set and selects the processor with the smallest value for $loadFunction_{QA}$, where $loadFunction_{QA}$ is defined in Equation 1. Hence the scheduling overhead depends linearly on the number of nodes. The PR and AP dispatchers have a similar overhead:

$$t_{scheduling} = 3L_{size}\frac{N}{b_{memo}} \tag{15}$$

The migration overhead is expanded as follows:

$$t_{migration} = t_{qa\_migration} + t_{pr\_migration} + t_{ap\_migration} \tag{16}$$

where the three terms indicate the overhead of the three dispatching points: $t_{qa\_migration}$ denotes the overhead of the migration initiated by the question dispatcher, $t_{pr\_migration}$ indicates the overhead of the PR-dispatcher-initiated migration, and $t_{ap\_migration}$ indicates the overhead of the AP-dispatcher-initiated migration. $t_{qa\_migration}$ is expanded as the cost of moving the question to a remote node and the cost of retrieving the answers from the remote node. Because the network is accessed by each question with probability p_{net}, the available network bandwidth is $b_{net}/(Mp_{net})$. Considering that a Q/A task is migrated by the question dispatcher with probability p_{qa}, $t_{qa\_migration}$ becomes:

$$t_{qa\_migration} = p_{qa}(Q_{size} + N_A A_{size})\frac{Mp_{net}}{b_{net}} \tag{17}$$

The overhead of the PR migration consists in sending the keywords to the destination node, receiving the paragraphs extracted by the remote paragraph retrieval engine, and reading the paragraphs from disk. The available disk bandwidth is b_{disk}/R, and, similarly with the previous equation, the available network bandwidth is $b_{net}/(Mp_{net})$. $t_{pr\_migration}$ is then expanded as:

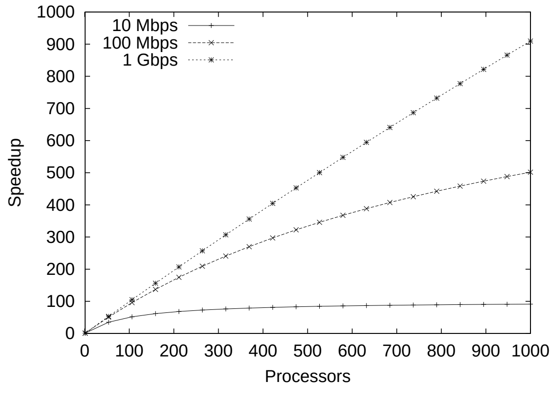
$$t_{pr\_migration} = p_{pr}[N_K K_{size}\frac{Mp_{net}}{b_{net}} + N_P P_{size}(\frac{Mp_{net}}{b_{net}} + \frac{R}{b_{disk}})] \cong p_{pr} N_P P_{size}\frac{Mp_{net}}{b_{net}} \tag{18}$$

The overhead of the AP migration consists in sending the accepted paragraphs to the remote AP module, receiving the answers, and reading the answers from disk:

$$t_{ap\_migration} = p_{ap}[N_{PA} P_{size}\frac{Mp_{net}}{b_{net}} + N_A A_{size}(\frac{Mp_{net}}{b_{net}} + \frac{R}{b_{disk}})] \cong p_{ap} N_{PA} P_{size}\frac{Mp_{net}}{b_{net}} \tag{19}$$

From Equations 16, 17, 18, and 19, the migration overhead becomes:

$$t_{migration} \cong (p_{qa}(Q_{size} + N_A A_{size}) + (p_{pr} N_P + p_{ap} N_{PA}) P_{size})\frac{Mp_{net}}{b_{net}} \tag{20}$$



(a)

L_{size}	100 bytes
Q_{size}	100 bytes
N_A	5 answers
A_{size}	300 bytes
N_P	372 paragraphs
N_{PA}	271 paragraphs
P_{size}	1133 bytes
t_1^1	94 seconds
R	8
p_{qa}	0.42
p_{pr}	0.45
p_{ap}	0.43
p_{net}	0.45

(b)

Figure 8. (a) Analytical system speedup for various network bandwidths; (b) Plot parameters based on the TREC-9 question set

From Equations 14, 15 and 20 $t_{overhead}$ becomes:

$$t_{overhead} \cong Nt_1^1 \frac{L_{size}}{b_{net}} + (p_{qa}(Q_{size} + N_A A_{size}) + (p_{pr}N_P + p_{ap}N_{PA})P_{size}) \frac{Mp_{net}}{b_{net}} \tag{21}$$

From Equations 12 and 21 the speedup becomes:

$$S_N^1 = N/(1 + N\frac{L_{size}}{b_{net}} + Mp_{net}\frac{p_{qa}(Q_{size} + N_A A_{size}) + (p_{pr}N_P + p_{ap}N_{PA})P_{size}}{t_1^1 b_{net}}) \tag{22}$$

$$S_N^1 = 1/(\frac{1}{N} + \frac{L_{size}}{b_{net}} + Rp_{net}\frac{p_{qa}(Q_{size} + N_A A_{size}) + (p_{pr}N_P + p_{ap}N_{PA})P_{size}}{t_1^1 b_{net}}) \tag{23}$$

Using data from the TREC-9 competition, Figure 8 (a) plots the analytical system speedup for various network bandwidths. Figure 8 (b) shows the parameters used to plot the system speedup. Figure 8 (a) shows that, for large scale systems, high-performance interconnection networks are required to obtain good performance. For a fast (according to today's standards) 1 Gbps network, the system efficiency E, defined as $E = S_N^1/N$ is approximately 0.9 for 1000 processors. This indicates that the model proposed in this thesis is a viable candidate for efficient, large-scale Q/A systems. For smaller scale systems, the system is efficient even for slower interconnection networks: the system obtains an efficiency $E = 0.9$ for 100 processors and a 100 Mbps interconnection network.

5.2 Intra-Question Parallelism Analysis

While the previous subsection analyzes the system behavior as multiple questions are executed, this subsection focuses on the individual question. In this subsection we compute the individual question speedup obtainable using intra-question parallelism and the number of nodes from which the overhead introduced by the intra-question parallelism overcomes its benefits. The question execution time on the sequential system is defined by the equation:

$$t_1^1 = t_{QP} + t_{PR} + t_{PS} + t_{PO} + t_{AP} \tag{24}$$

where each module from Figure 1 contributes with its own term in the summation. The question execution time on the N-node system is defined as:

$$t_N^1 = t_{QP} + \frac{t_{PR}}{N} + \frac{t_{PS}}{N} + t_{PO} + \frac{t_{AP}}{N} + t_{overhead-intra} \tag{25}$$

The paragraph ordering time t_{PO} can not be improved due to the inherent sequential nature of the corresponding module. In the current Falcon architecture question processing is also considered a sequential module, hence t_{QP} is not improved. The rest of the question answering modules, consisting of over 90% of the overall execution time (see Table 2), can be parallelized with a factor N. The parallelization process introduces an overhead denoted with $t_{overhead-intra}$ in Equation 25. The overhead time is expanded as:

$$t_{overhead-intra} = t_{monitoring} + t_{scheduling} + t_{partition} \cong t_{partition} = t_{pr\_partition} + t_{ap\_partition} \tag{26}$$

where $t_{monitoring}$ represents the time spent performing load monitoring; $t_{scheduling}$ denotes the overhead of the PR and the AP dispatchers. Both $t_{monitoring}$ and $t_{scheduling}$ are ignored in Equation 26 because they are comparatively much smaller than the other components of the overhead time (see Equations 14 and 15). $t_{partition}$ represents the task partitioning overhead, generated by the additional inter-processor communication and the additional modules which control the partitioning mechanism. The partitioning overhead is further expanded as the cumulative overhead of the PR and AP module partitioning.

The PR module partitioning time is generated by the time required to send the input keywords towards the remote paragraph retrieval engines, the time to receive the output paragraphs from the paragraph retrieval modules, and the overhead of the additional paragraph merging module, which reads the received paragraphs from disk:

$$t_{pr\_partition} = N\frac{N_K K_{size}}{b_{net}} + \frac{N_P P_{size}}{b_{net}} + \frac{N_P P_{size}}{b_{disk}} \cong \frac{N_P P_{size}}{b_{net}} + \frac{N_P P_{size}}{b_{disk}} \tag{27}$$

In Equation 27 we ignored the keyword propagation overhead, because for our domain of interest $N \ll N_P$ and $N_K K_{size} \ll P_{size}$.

The AP module partitioning time is generated by the time to divide the accepted paragraph set into sub-sets to be assigned to remote AP modules (task performed by the paragraph assignment module), the time to send the accepted paragraph sub-sets to the remote AP sub-tasks, the time to receive the answers, the time to merge the received answers into a single set (task performed by the answer merging module), and the time to sort the answer set (task performed by the answer sorting module). Note that each AP module has to return N_A answers. The final best N_A answers output to the user are decided in the answer sorting stage.

$$t_{ap\_partition} = \frac{N_{PA} P_{size}}{b_{disk}} + \frac{N_{PA} P_{size}}{b_{net}} + \frac{N N_A A_{size}}{b_{net}} + \frac{N N_A A_{size}}{b_{disk}} + N N_A \log(N N_A) \tag{28}$$

$$t_{ap\_partition} \cong \frac{N_{PA} P_{size}}{b_{disk}} + \frac{N_{PA} P_{size}}{b_{net}} \tag{29}$$

The time to receive, merge and sort the answers are ignored in Equation 29 due to their relatively small values when compared to the other components of the AP partitioning time. Replacing the expanded overhead times in Equation 25, t_N^1 becomes:

$$t_N^1 = t_{QP} + \frac{t_{PR}}{N} + \frac{t_{PS}}{N} + t_{PO} + \frac{t_{AP}}{N} + \frac{(N_P + N_{PA})P_{size}}{b_{net}} + \frac{(N_P + N_{PA})P_{size}}{b_{disk}} \tag{30}$$

or in a simplified notation:

$$t_N^1 = \frac{t_{par}}{N} + t_{seq} \tag{31}$$

where t_{par} and t_{seq} are defined as:

$$t_{par} = t_{PR} + t_{PS} + t_{AP} \tag{32}$$

$$t_{seq} = t_{QP} + t_{PO} + \frac{(N_P + N_{PA})P_{size}}{b_{net}} + \frac{(N_P + N_{PA})P_{size}}{b_{disk}} \tag{33}$$

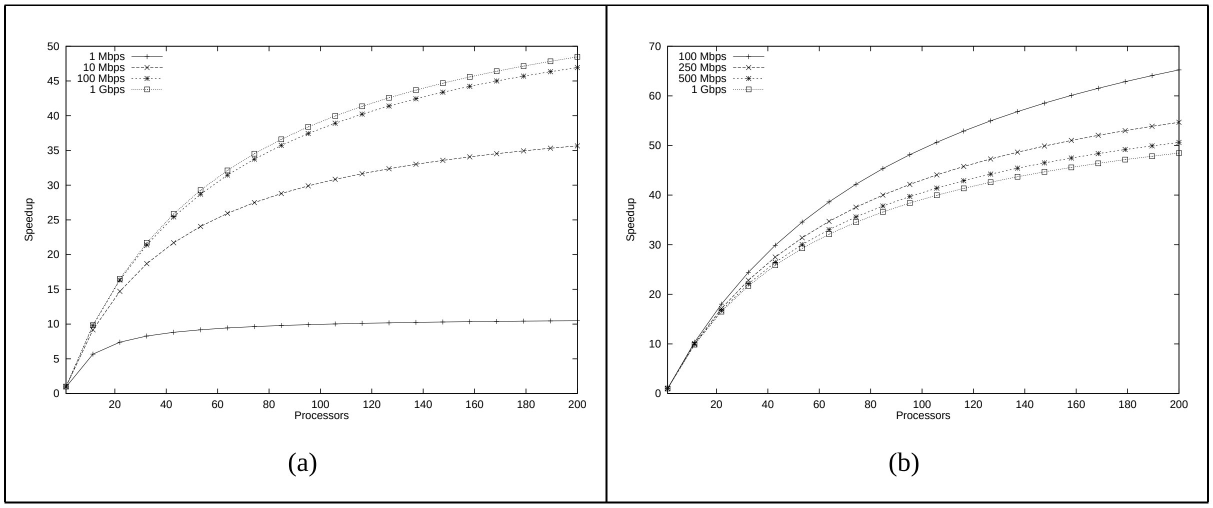


Figure 9. Analytical question speedup for: (a) disk bandwidth of 1 Gbps and various network bandwidths; and (b) network bandwidth of 1 Gbps and various disk bandwidths

As Equation 31 indicates, t_N^1 has two major parts: a first part consisting of parallelizable module times (t_{par}), which decreases linearly with N, and a second part made of sequential modules and overhead times (t_{seq}), which is constant for a given question. For practical considerations, it is worth increasing the number of processors as long as t_{par}/N is the significant part of t_N^1. On the other hand, it is not justified to increase the number of processors after t_{seq} becomes the dominant part of t_N^1. This is expressed as:

$$N \leq \frac{t_{par}}{t_{seq}} \tag{34}$$

The individual question speedup is computed using the equation:

$$S_N^1 = \frac{t_1^1}{t_N^1} \tag{35}$$

$$S_N^1 = \frac{t_{QP} + t_{PR} + t_{PS} + t_{PO} + t_{AP}}{t_{seq} + t_{par}/N} \tag{36}$$

Using data from the TREC-9 competition, Figure 9 plots the analytical individual question speedup for various disk and network bandwidths. As expected, the speedup increases with the network bandwidth (see Figure 9 (a)) because the overhead of inter-node communication becomes less significant as the network performance improves. Figure 9 (b) shows that the speedup decreases as the disk bandwidth increases. The explanation is that t_{PR} decreases as disk bandwidth increases, hence the distribution overhead becomes comparatively more significant.

disk\net	1 Mbps	10 Mbps	100 Mbps	1 Gbps

100 Mbps	N = 17
	S = 8.65
		N = 64
		S = 32.84
			N = 89
			S = 45.75
				N = 93
				S = 47.73

250 Mbps	N = 13
	S = 6.61
		N = 49
		S = 25.30
			N = 68
			S = 35.33
				N = 71
				S = 36.87

500 Mbps	N = 12
	S = 6.01
		N = 43
		S = 22.49
			N = 61
			S = 31.81
				N = 64
				S = 33.28

1 Gbps	N = 11
	S = 5.59
		N = 41
		S = 21.35
			N = 57
			S = 29.90
				N = 60
				S = 31.34

Table 4. Practical upper limits on the number of processors and the corresponding speedups

While the speedup functions depicted in Figure 9 asymptotically approach ideal upper limits, these ideal speedup values will never be reached by a practical system. Table 4 shows the practical upper limit on the number of processors computed by Equation 34 and the corresponding speedup values for various disk and network bandwidths. Depending on these two parameters the practical upper limit on the number of processors varies from 11 to 93, and the corresponding speedup values range between 5.59 to 47.73.

The conclusion of this analysis is that the exploitation of intra-question parallelism achieves significant reduction of the individual question latencies, but it is limited to a small number of processors due to the increasing inter-module communication overhead (see Table 4). The exploitation of intra-question parallelism is necessary for an interactive system where individual question execution times need to be reduced as much as possible, but in order to achieve an overall system throughput intra-question parallelism is not sufficient. The solution for an interactive, large scale Q/A system is a distributed model where both inter-question and intra-question parallelism are used.

6 Experimental Results

This section performs an empirical performance analysis of the distributed Q/A model introduced in Section 3. The tests presented in this section focus on the following directions: (i) comparison with other load balancing models, and (ii) analysis of individual question speedup obtained using intra-question parallelism. These experiments show the two advantages of the proposed architecture: the first tests show that at *high system load*, the load balancing strategy implemented by the three scheduling points outperforms other load balancing models. The second test shows that at *light system load*, the exploitation of intra-question parallelism provides significant speedup for individual questions. Note that even though this section presents separate tests to illustrate these issues, the Q/A system dynamically detects the current load and selects the appropriate degree of inter and intra task parallelism at runtime.

The experimental hardware platform used in our experiments consists of a network of twelve 500 MHz Pentium III computers running Linux kernel 2.2.12. Each computer has 256 MB of RAM and 50 GB disk space. A separate computer was dedicated to the parsing server, which requires 180 MB of RAM. The interconnection network is a star-configuration 100 Mbps Ethernet. Each node has a copy of the TREC-9 collection. The TREC-9 collection was divided into 8 sub-collections, separately indexed using a Boolean information retrieval system built on top of Zprise [30].

6.1 Comparison with Other Load Balancing Models

The experiments presented in this subsection compare the system performance obtained using the scheduling techniques described in Section 3 with other approaches. The first technique considered as benchmark uses a round-robin processor allocation, which emulates the DNS name-to-address mapping. Throughout the tests presented in this section, this load balancing strategy is identified as *DNS*. The second approach adds a dispatcher before the question answering task. The goal of the question dispatcher is to improve on the rough balancing performed by the DNS method. Because the round-robin strategy used by DNS does not take into consideration the actual system load, some requests can be allocated to over-loaded processors. In such situations, the question dispatcher migrates the question to the least loaded processor at the moment. A distributed design for the question dispatcher is presented in Section 3. To our knowledge, this is the only model currently implemented in distributed information retrieval systems [3, 7]. We name this model *INTER*. The third method adds two more dispatchers, before the paragraph retrieval and the answer processing modules. This scheduling approach is the distributed Q/A model presented in detail in Section 3. We name this model *DQA*.

In order to achieve a reliable evaluation of the three load balancing models, the Q/A system has to be in a high load state. Due to the memory limitations of our test platform (256 MB per processor), we consider a node to be fully-loaded when it simultaneously runs 4 questions (a question requires from 25 to 40 MB of memory). A node running more than 4 simultaneous questions is considered overloaded. The system is brought to a high load state by starting twice the number of questions that will generate an overload state ($8N$, where N is the number of processors), at

	DNS	INTER	DQA
4 processors	2.64	3.45	4.18
8 processors	5.04	5.52	7.77
12 processors	7.89	9.71	12.09

Table 5. System throughput (questions/minute)

	DNS	INTER	DQA
4 processors	143.88	122.51	111.85
8 processors	135.30	118.82	113.53
12 processors	132.45	115.29	106.03

Table 6. Average question response times (seconds)

intervals of time ranging between 0 and 2 seconds. The questions were selected randomly from the TREC-8 and TREC-9 question set. We were however careful to use the same questions and the same startup sequence for all tests. For all tests we assumed a perfect round-robin initial question distribution.

The system throughput is presented in Table 5. The measured throughput indicates that the DNS scheduling relying only on the round-robin distribution of questions performs the worst out of the three approaches investigated. The INTER strategy increases the throughput with an average of 21%. This performance improvement is due to the question dispatcher which migrates questions from over-loaded to under-loaded processors. However, the approach implemented by the INTER strategy is not perfect: questions from over-loaded processors are migrated to processors considered best for the average question, but which are not necessarily the best for the currently migrated question. The limitation of the INTER approach is illustrated by the good performance measured for the DQA strategy, which increases the INTER throughput with an additional 28.8%.

The system throughput is a good indication of system resource utilization. However, for an interactive system an equally important metric is the user satisfaction, which, from a quantitative perspective, is expressed by the individual question latency. Table 6 shows the average question latencies for the three load balancing strategies investigated. The same performance ranking is observed: INTER outperforms DNS, but the best performance is obtained by the DQA strategy.

The results presented in Tables 5 and 6 indicate that DQA is the best load balancing strategy out of the three algorithms investigated. The DQA good performance is caused by the succession of three dispatchers, where each dispatcher improves upon the decision taken by the previous one. This behavior is illustrated in Table 7, which shows the number of times each dispatcher disagreed

	INTER	DQA
32 questions (4 processors)	QA: 8	QA: 17 PR: 10 AP: 10
64 questions (8 processors)	QA: 15	QA: 26 PR: 34 AP: 33
96 questions (12 processors)	QA: 23	QA: 37 PR: 43 AP: 41

Table 7. Number of migrated questions at the three scheduling points

with the previous allocation. The information presented in Table 7 should be interpreted as follows: *QA* indicates the number of times the question dispatcher disagreed with the initial round-robin question distribution. *PR* and *AP* indicate the number of times the paragraph retrieval and answer processing dispatchers disagreed with the decision taken by the question dispatcher. The results presented in the third column of Table 7 were also used to compute the migration probabilities at the three dispatching points used by the analytical model introduced in Section 5. The information presented in Table 7 is important because it shows that the PR and AP dispatchers are quite active. For example, for the 4 processor test, the PR dispatcher disagreed with the question dispatcher for 10 questions (out of 32). The same number of disagreements is observed for the AP dispatcher. This indicates that, in many cases, the processor allocated by the question dispatcher is not the best for the individual modules.

6.2 Analysis of Intra-Question Parallelism

The first part of this section evaluated the performance of the distributed Q/A system at high load, by comparing the effectiveness of the implemented load balancing heuristics with other load balancing strategies. This subsection focuses on the reverse situation: the system performance is evaluated at low load, when multiple resources are available such that intra-question parallelism can be exploited. The first experiment presented measures the individual question response times and speedups obtained using intra-question parallelism. For this test we selected from the TREC8 and TREC9 Q/A track 307 questions that are complex enough to justify distribution on all nodes (we selected the questions which have at least 20 paragraphs allocated to each AP module). Questions were executed one at a time, in order to accurately measure t_N^1 and $S_N^1 = t_1^1/t_N^1$. For both the PR and AP modules we used the receiver-controlled partitioning algorithm described in detail

	QP	PR	PS	PO	AP	Question response time (including overhead)
1 processor	0.81	38.01	2.06	0.02	117.55	158.47
4 processors	0.81	9.78	0.54	0.02	31.51	43.13
8 processors	0.81	7.34	0.41	0.02	17.86	27.07
12 processors	0.81	7.34	0.41	0.02	11.90	21.17

Table 8. Observed module times and average question response times (measured in seconds)

	Keyword Sending (to PR)	Paragraph Receiving (from PS)	Paragraph Sending (to AP)	Answer Receiving (from AP)	Answer Sorting (from AP)	Total
4 processors	0.04	0.19	0.15	0.05	0.01	0.44
8 processors	0.08	0.24	0.19	0.09	0.01	0.61
12 processors	0.08	0.24	0.22	0.12	0.01	0.67

Table 9. Measured distribution overhead per question (in seconds)

Section 4 (we justify our decision in the next subsection by showing that the receiver initiated partitioning algorithm achieves the best performance out of the three algorithms introduced in Section 4). The measured module times and average question response times are presented in Table 8. Table 8 shows that, besides the QP and PO modules which are not partitioned, the module latency times are reduced significantly as the number of processors increases. For example, for the 4-processor configuration the overall measured speedup is 3.67. The results observed for the 8 and 12-processor configurations, while still showing significant improvement of the module execution times, are not as impressive as the results measured for the 4-processor test. The explanation is two-fold: the first cause for the 8 and 12-processor results is the *uneven partition granularity*. For example, even though the TREC-9 document collection is divided in 8 sub-collections of roughly equal sizes, the PR sub-task granularities vary drastically based on the frequencies of the keywords in the given sub-collection. This uneven distribution of computational effort among sub-tasks reduces the speedup of the PR module on the 8-processor configuration from an ideal value of 8 to 5.17. To a less extent, the same observation is valid for the answer processing module. The second observation applies to the timing of the PR module on the 12-processor configuration. Because the TREC9 collection is divided in only 8 sub-collections, the execution time of the PR module on the 12-node configuration is similar to the execution time on the 8-node configuration.

The question response times presented in the last column of Table 8 include the overhead of the partitioning mechanism. This overhead is detailed in Table 9. As expected, the highest overhead is observed for the manipulation of paragraphs. Nevertheless, the complete overhead is small, ac-

	Analytical	Measured
4 processors	3.84	3.67
8 processors	7.34	5.85
12 processors	10.60	7.48

Table 10. Analytical versus measured question speedup

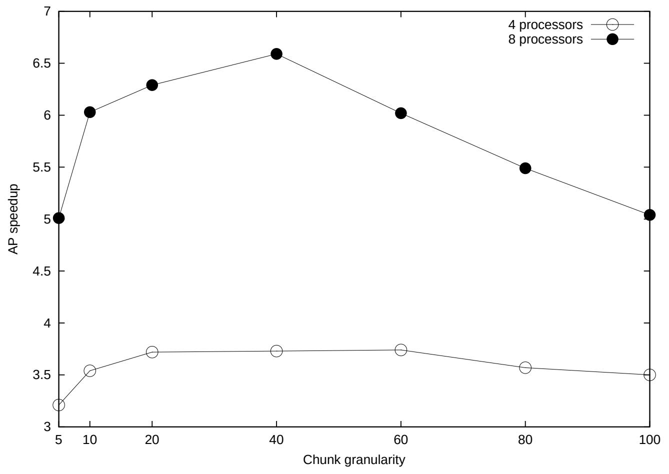


Figure 10. Answer processing speedup for the RECV partitioning algorithm and various paragraph chunk sizes

counting for less than 3% of the question response time in all configurations. Table 10 summarizes our analysis of intra-question parallelism: we measured significant speedup of individual questions, but the speedup measured is lower than the theoretically estimated speedup mainly due to uneven partition granularities.

6.3 Comparison of Partitioning Strategies

This subsection investigates the performance of the three partitioning algorithms proposed in Section 4: *SEND*, *ISEND*, and *RECV*. The performance of these algorithms is measured for the partitioning of the answer processing module, because the low task granularity allows more experimental flexibility. The first experiment measures the optimal chunk granularity for the RECV algorithm. This partitioning strategy divides the module input in smaller equal-size chunks, which are retrieved and processed one by one by the processors selected for partitioning. The chunk size can be preset (i.e. the sub-collection for paragraph retrieval) or set before partitioning (i.e. number of paragraphs for the answer processing modules). Intuitively, small chunks have the advantage that the *uneven partition granularity* problem is minimized. The previous section showed that this

	SEND	ISEND	RECV
4 processors	2.71	3.61	3.73
8 processors	4.78	6.25	6.58
12 processors	7.17	9.22	9.87

Table 11. Answer processing speedup for different partitioning strategies

is a significant problem, especially for the PR module where the chunk sizes can not be modified (see Table 8). On the other hand, small chunks have the disadvantage of *increased distribution overhead* (more connections are required and more answers have to be sorted afterwards). This experiment provides an empirical answer to the above dilemma. Figure 10 shows the speedup for answer processing module when the RECV partitioning strategy is used with chunk sizes varying from 5 to 100 paragraphs. We measured the system behavior for 4 and 8-processor configurations. Figure 10 supports our intuition: the best performance is observed for chunks of approximately 40 paragraphs. The performance for bigger chunks decreases due to the uneven granularity problem, while the performance for smaller chunks suffers from the increased overhead.

The next experiment compares the performance of all three algorithms. For the RECV strategy we used chunks of 40 paragraphs. The speedups measured for the AP module are presented in Table 11. These results show that the SEND algorithm has by far the worst performance out of the three. The best performance is observed for the RECV algorithm, closely followed by ISEND. The good performance of the RECV algorithm is no surprise: it is an accepted fact that receiver-controlled algorithms achieve better performance than sender-controlled algorithms [32]. A surprise was the good performance measured for the ISEND partitioning strategy. ISEND's excellent performance indicates that the PO module indeed sorts the retrieved paragraphs in decreasing order of their granularity, even though the metrics used address qualitative issues.

Even if the above experiments were performed in the context of answer processing partitioning, the above observations apply also to paragraph retrieval partitioning with few modifications. A separate experiment indicated that the RECV strategy outperforms the SEND algorithm for PR partitioning. The difference between the two strategies was even more significant than the results presented in Table 11. The explanation is that the uneven granularity problem is more evident for paragraph retrieval than answer processing due to the large, topic-oriented collections. The ISEND algorithm does not apply to PR partitioning, because document collections are not ranked like the paragraph set considered in AP partitioning.

7 Conclusions

This paper introduces a distributed architecture for question answering systems. The architecture proposes a dynamic load balancing tailored for the specific requirements of the Q/A application: a large number of simultaneous questions must be simultaneously serviced, and the individual question latencies must be reduced as much as possible. The distributed Q/A model proposed exploits both *inter and intra-question parallelism* using three scheduling points: one before the Q/A task is started, and two before the Q/A task performance bottlenecks: paragraph retrieval and answer extraction. All schedulers use allocation heuristics specialized for the resource requirements of the managed module. The overall effect is that the Q/A task is *migrated* at any of the three scheduling points to the node best fit for the remaining of the Q/A task in order to maximize system throughput, and, if multiple nodes are under-loaded, the performance bottleneck modules are *partitioned* in order to reduce the individual question response time.

An analytical performance model is introduced. The model analyzes the system behavior under two conditions: the *inter-question parallelism overhead* is analyzed when all three dispatchers are deployed, but partitioning is not used because the distributed system is used at full capacity, and the *intra-question parallelism overhead* is analyzed when the system is under-loaded and the Q/A task can be partitioned. The analysis of the partitioning overhead indicates that intra-question parallelism leads to significant speedup of individual questions, but it is practical up to about 90 processors, depending on the system parameters. On the other hand, while the exploitation of inter-task parallelism does not guarantee the same individual question speedups, it provides a scalable way to improve throughput. The analytical model indicates that, if fast interconnection networks are available, the system efficiency is good (approximately 0.9) even for 1000 processors.

The experimental results indicate that, at high system load, the dynamic load balancing strategy proposed in this paper outperforms two other traditional approaches. The analysis of the two partitioning strategies proposed (sender and receiver-controlled) indicates that for the paragraph retrieval module the receiver-controlled partitioning approach is the only practical procedure. For the answer extraction task, the sender-controlled approach achieves comparable performance with the receiver-controlled method.

References

[1] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999.

[2] D. Andresen and T. Yang. SWEB++: Partitioning and scheduling for adaptive client-server computing on WWW. In *Proceedings of 1998 SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[3] D. Andresen, T. Yang, O. Ibarra, and O. Egecioglu. Adaptive partitioning and scheduling for enhancing WWW application performance. *Journal of Parallel and Distributed Computing*, 49(1), February 1998.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[5] T. Brisco. RFC 1794: DNS support for load balancing, April 1995.

[6] E. Brown and H. Chong. The GURU system in TREC-6. In *6th Text Retrieval Conference (TREC)*, November 1997.

[7] B. Cahoon, K. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, 18(1), January 2000.

[8] J. Callan, M. Connell, and A. Du. Automatic discovery of language models for text databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 479–490, 1999.

[9] J. Callan, Z. Lu, and W. Croft. Searching distributed collections with inference networks. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, 1995.

[10] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. *IEEE Computer*, 28(1):40–49, January 1995.

[11] S. Chen, L. Xiao, and X. Zhang. Dynamic load sharing with unknown memory demands of jobs in clusters. In *Proceedings of the 21st Annual International Conference on Distributed Computing Systems (ICDCS'2001)*, pages 109–118, 2001.

[12] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 46(2), 1997.

[13] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of Supercomputing '99*, November 1999.

[14] L. Gravano and H. Garcia-Molina. Generalizing GLOSS to vector-space databases and broker hierarchies. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 78–89, 1995.

[15] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of GLOSS for the text database discovery problem. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 126–137, 1994.

[16] S. Harabagiu, D. Moldovan, M. Pasca, R. Mihalcea, M. Surdeanu, R. Bunescu, R. Girju, V. Rus, and P. Morarescu. FALCON: Boosting knowledge for answer engines. In *Proceedings of the Text Retrieval Conference (TREC-9)*, November 2000.

[17] S. Harabagiu, M. Pasca, and S. Maiorano. Experiments with open-domain textual question answering. In *Proceedings of COLING-2000*, August 2000.

[18] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for load balancing. *ACM Transactions on Computer Systems*, 3(3), 1997.

[19] J. Hawking, N. Craswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In *Proceedings of the Text Retrieval Conference (TREC-7)*, November 1998.

[20] C. Hui and S. Chanson. Improved strategies for dynamic load sharing. *IEEE Concurrency*, 7(3), 1999.

[21] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6), June 2000.

[22] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7), 1991.

[23] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, January 1987.

[24] P. K. K. Loh, W. J. Hsu, C. Wentong, and N. Sriskanthan. How network topology affects dynamic load balancing. *IEEE parallel and distributed technology: systems and applications*, 4(3):25–35, Fall 1996.

[25] R. Luling, B. Monien, and F. Ramme. Load balancing in large networks: A comparative study. *3rd IEEE Symposium on Parallel and Distributed Processing*, pages 686–689, December 1991.

[26] D. Moldovan, S. Harabagiu, M. Pasca, R. Mihalcea, R. Goodrum, R. Girju, and V. Rus. The structure and performance of an open-domain question answering system. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, pages 563–570, October 2000.

[27] D. Moldovan, S. Harabagiu, M. Pasca, R. Mihalcea, R. Goodrum, G. R., and V. Rus. LASSO: A tool for surfi ng the answer net. In *Proceedings of the Text Retrieval Conference (TREC-8)*, November 1999.

[28] F. Muniz and E. J. Zaluska. Parallel load balancing: An extension to the gradient model. *Parallel Computing*, 21:287–301, 1995.

[29] NIST. The Text REtrieval Conference. http://trec.nist.gov/.

[30] NIST. The ZPRISE 2.0 home page. http://www-nlpir.nist.gov/works/papers/ zp2/main.html.

[31] V. A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Fifth Distributed Memory Computing Conference*, pages 995–990, April 1990.

[32] A. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[33] G. Voelker. Managing server load in global memory systems. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997.

[34] E. Voorhees and D. Harman. Overview of the ninth Text REtrieval Conference. In *Proceedings of the Text Retrieval Conference (TREC-9)*, November 2000.

[35] M. Willebeck-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[36] L. Xiao, X. Zhang, and Y. Qu. Effective load sharing on heterogenous networks of workstations. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, May 2000.

[37] X. Zhang, Y. Qu, and L. Xiao. Improving distributed workload performance by sharing both CPU and memory resources. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, April 2000.