A HIGH-SPEED LINEAR ALGEBRA LIBRARY WITH AUTOMATIC PARALLELISM

Michael L. Boucher
Dakota Scientific Software, Inc.
501 East Saint Joseph Street
Rapid City, SD 57701-3995

scisoft@well.sf.ca.us
(605) 394-1256 fax
(605) 394-9257 voice

ABSTRACT

Parallel or distributed processing is key to getting highest performance from the latest generation of high-performance workstations. However designing and implementing efficient parallel algorithms is difficult and error-prone. It is even more difficult to write code that is both portable to and efficient on many different computers. Finally, it is harder still to satisfy the above requirements and include the reliability and ease of use required of commercial software intended for use in a production environment. As a result, the application of parallel processing technology to commercial software has been extremely small even though there are numerous computationally demanding programs that would significantly benefit from application of parallel processing. This paper describes DSSLIB, which is a library of subroutines that perform many of the time-consuming computations in engineering and scientific software. DSSLIB combines the high efficiency and speed of parallel computation with a serial programming model that eliminates many undesirable side-effects of typical parallel code. The result is a simple way to incorporate the power of parallel processing into commercial software without compromising maintainability, reliability, portability, or ease of use. This gives significant advantages over less powerful non-parallel entries in the market.

INTRODUCTION

Designing and implementing a parallel algorithm in a way that is both portable and efficient on a wide range of hardware and software configurations is a task that is sufficiently difficult and time-consuming that it is rarely done. Even when developing codes that require only a moderate amount of optimization, it is common to use techniques that are specific to a particular machine and that are not easily portable to other hardware or software architectures. As a program becomes more closely tied to a specific environment it requires more extensive changes in order to adapt to changes in the environment. A predictable undesirable result of writing parallel programs in a way that binds them to a specific hardware or software environment is that the "dusty deck" codes of the future will be even more difficult to use and maintain than the dusty deck codes with which we have to deal today. A predictable result of the pace of technology change is that codes written today will require more frequent updates to adapt, customize, and optimize them for the latest computer system. Therefore, we can expect the maintenance phase to get even more expensive if we continue present approach of requiring that programmers customize programs for a specific environment in order to get acceptable speed. This paper first describes a method of implementing parallelism that avoids many of the problems that inhibit writing portable parallel code, then describes a library of parallel linear algebra subroutines that uses that approach.

Writing portable parallel codes is made difficult by a combination of factors listed below and expanded upon in the paragraphs that follow.

1. It is much more difficult to adequately test parallel code due to nondeterminism in the order in which operations are performed and a lack of good analysis tools.
2. Faults and events, such as division by zero or overflow, tend to be masked, reported incorrectly, or reported inconsistently from one run to the next.
3. Parallelization can slightly change the numerical properties of a given method.

4. Wide variations in the performance characteristics of different parallel and distributed architectures make it difficult to write a single code that is efficient on a range of machine types.
5. Dependency on the run-time environment makes it difficult to write a single code that is efficient on a single machine type under varying system loads.

Parallel algorithms, especially those that use medium- and coarse-grain parallelism, are almost intrinsically nondeterministic in their execution. For example, the order in which synchronizing semaphores are accessed can cause significant changes in the internal behavior of a program. It is possible that some, but not all, access patterns to semaphores or other global data will work properly. It is therefore possible that a bug will go undetected through even extensive structured testing. Few of the standard analysis tools such as static analyzers have adequate support for parallel programs and so those tools are not as helpful with parallel programs as they are with serial programs.

Virtually none of the available parallel systems report faults or events that take place on a parallel CPU. For example, a division by zero or an overflow on a parallel CPU will generally go undetected by the host processor. Virtually all of those systems that do report faults or events back to the host processor do so in a nondeterministic manner. This can significantly complicate the task of debugging.

Parallelism may change some of the numeric properties of a code. There are many well-known examples of this effect; one obvious example is that splitting a summation in different ways can generate different results.

Parallel and distributed architectures are available for all classes of machines ranging from PCs to supercomputers and each of these architectures has widely varying performance characteristics. The parallelism on a given computer system may be fine-, medium, or coarse-grain parallelism, or it may be any combination of those three models. Fine-grain parallelism can take the form of an instruction pipeline or independent functional units in a single CPU. There may also be multiple processor types in a single computer, for example independent CPU and I/O processors or a CPU and an FPU. Medium-grain parallelism is typically loop-level parallelism between several processors with a shared memory. Coarse-grain parallelism can occur between any two processors regardless of whether they share a common memory. All of this variation makes it difficult to design a code that will run well on many or all of the architectures.

Finally, variations in the run-time environment can make it very difficult to write code that is efficient even on a single machine type but under varying work loads. For example, distributing a computation across workstations in a cluster can be done efficiently when the workstations are available and the network is lightly loaded, but inefficient when the workstations are busy or if the network is heavily loaded. Finally, changes in problem size can significantly change the performance characteristics of a particular parallel algorithm.

BACKGROUND / EXISTING APPROACHES

We start by considering the systems for parallel and distributed processing that are widely available today. They appear to fall into one of three categories: remote procedure calls, subroutines libraries that provide parallelism primitives, and pre-parallelized subroutine libraries. The systems that we consider as the representatives of each of these categories are UNIX™ RPCs as implemented by Sun Microsystems [8], Parallel Virtual Machine, and LAPACK [1].

RPCs are a mechanism in which a UNIX programmer can run a procedure on a remote machine using the simple semantics of an ordinary procedure call. When invoked, a synchronous RPC transmits the arguments to a remote machine which then executes the procedure and returns the results. In this way, RPCs provide distributed processing. A layer called XDR tries to hide from the programmer machine-specific details about byte ordering, word length, and so forth by doing some of the data conversion necessary to make the data in the arguments understandable to the remote machine and to make the result from the remote machine understandable to the host. An asynchronous RPC is similar to a synchronous RPC except that the host does not wait for a result from a remote machine after initiating a remote procedure. After initiating a remote procedure on one machine, the host

may make one or more other asynchronous RPCs and in this way the host achieves parallel processing. RPCs are supported by `rpcgen`, which allows a programmer to create RPC templates relatively easily. The strengths of the RPC are its ease of invocation using standard procedure call semantics and its relatively easy portability among UNIX operating environments.

The standard form of RPC/XDR has many drawbacks. First, XDR has a clear bias towards C programs running on 32-bit machines with IEEE floating point arithmetic and it has poor support for data types that are not common on this configuration. For example, FORTRAN's complex data type (a data type not available in C), double precision floating point on a Cray (a 64-bit machine that does not use IEEE arithmetic), and BCD (often supported on IBM mainframes and 80x87 math coprocessors) are poorly handled by the standard XDR. While there is some portability among UNIX operating environments, there is essentially no hope of easily porting an RPC-based program to a non-UNIX environment. Synchronous RPCs do not allow parallel processing and the asynchronous RPCs that do allow parallel processing are almost hopelessly difficult to use. Synchronous RPCs are also less portable than asynchronous RPCs. RPCs do not duplicate the machine state of the host machine on the remote machine so that special processing options selected on the host will not operate correctly on the remote machine. For example, if a program sets the IEEE rounding mode on the host then computations on the host will round correctly and computations on remote machines will round incorrectly. Finally, RPCs have no fault tolerance. If a temporary network glitch occurs or if a remote machine crashes while an RPC-based program is running, then the program will hang or crash if the user is lucky, or the program will return the wrong answer with not even a hint of trouble if the user is unlucky.

Parallel Virtual Machine (PVM) is the representative of the class of parallel and distributed processing tools that are characterized by giving the user direct access to parallel and distributed processing primitives such as send, receive, initiate task, synchronize, and so forth. Other systems that fall into this category are Linda, Express, and the tasking mechanism built into Ada. PVM was developed by Dr. Jack Dongarra and his team at Oak Ridge National Laboratory (ORNL). It is a library of subroutines that gives a programmer close control over the parallelism employed by an application. PVM is more portable than RPC because PVM is not tied to a specific operating system. Dongarra and his team are considerably more scientifically oriented than the designers of RPC and so PVM correctly handles data types from languages besides C and machines with configurations besides 32-bit CPUs using IEEE arithmetic. PVM is designed to allow parallel processing in addition to simply the distributed processing capability of synchronous RPCs. Parallel processing with PVM is much easier than with asynchronous RPCs.

PVM is generally superior to RPC, but it has some drawbacks. From the perspective of a computer scientist, the power of PVM comes largely from the degree of control that the programmer can exercise over the process of parallelization. From the perspective of an atmospheric scientist, the problem with using PVM is the degree of control that the programmer must exercise over the process of parallelization. Many of the messy details of interprocessor communication that were concealed with RPCs are now the programmers problem. Another drawback to using PVM is that it requires that PVM-based programs be parallel or distributed programs. PVM-based programs that are developed on a multiprocessor SPARCstation 10™ will run beautifully, in large part due to the extremely fast interprocessor communication that comes with shared memory. PVM-based programs that are run on a network of SPARCstation IPXs will run poorly, in large part due to the extremely slow interprocessor communication that comes with the Ethernet connection. Regardless of the extreme variations in efficiency between these two operating environments, PVM forces the program to behave in exactly the same way in both environments. Finally, PVM is slightly better than RPC at fault tolerance, but not much. If a fault occurs in a network or on a remote machine while a parallel computation is in progress, the application probably will fail.

LAPACK represents the approach of using parallel subroutine libraries. In contrast to PVM, whose subroutines allow the user to define the operations involved in building a parallel application, LAPACK is a library of subroutines that may be supplied to a user after being optimized and parallelized. LAPACK includes subroutines to perform many of the common operations in computational linear algebra including solving systems of linear equations, matrix factorizations, eigensystem solvers, SVD, and similar operations.

Much of LAPACK is built on block operations, meaning that it divides a data set into subblocks that can be processed independently. It then does operations on those blocks. These blocks are then mapped for processing to the resources of a given machine. If there are multiple processors present then the blocks may be mapped to processors. The blocks may also be selected to correspond to the size of a cache for efficient memory access. The standard version of LAPACK as it comes from Oak Ridge National Laboratory is not optimized or parallelized, but the block structure does make it simpler to parallelize than other subroutine libraries that perform similar functions. LAPACK uses a subroutine called ILAENV to help it determine how each subroutine call should be blocked and so it is possible for ILAENV to react to changes in the environment and adapt its parallelism strategy accordingly. A major drawback to LAPACK with respect to its utility as a parallel programming environment is the same as its major strength, which is that the programmer has no concern with or control over the parallel processing. As a result, the programmer has no way to extend the parallel processing to get some capability that is not built into LAPACK.

A PROPOSED SOLUTION: DSSLIB

We have developed a parallelization system named DSSLIB that will avoid many, though not all, of the pitfalls of the available parallel programming systems. In particular, because it is a library, DSSLIB has the drawback present in LAPACK that a user cannot extend it to perform computations that are not build in. DSSLIB is based on a combination of software programs transferred from a variety of US. Government agencies and projects. Some of the software has been in wide use since 1979 while others have been introduced as recently as 1991. Specifically, DSSLIB includes version 1.1 of LAPACK and the latest versions of LINPACK [3] and levels 1, 2, and 3 of the Basic Linear Algebra Subprograms (BLAS) [6, 5, 4]. We intend this system for use in production codes, including commercial software, users who are not sophisticated programmers of parallel or distributed processing machines, and for any user regardless of sophistication who needs a significant speedup in an application but does not have the resources to dedicate to a parallelization effort.

The choice of target users implies that the software must have at least the following characteristics:

1. most or all of the parallelization must be automatic
2. runs correctly and reasonably efficiently in a variety of hardware configurations
3. complete fault tolerance.
4. does not interfere with other software that may be in use, possibly including other parallelization systems
5. compatible with all of the standard tools such as debuggers, profilers, etc.
6. requires no changes to move among many different hardware and software configurations; retains all of the characteristics listed above even as it is being used in a variety of configurations

DSSLIB satisfies the criteria above by presenting to an application a serial programming model even when it is running in parallel. A serial programming model means that DSSLIB appears to an application to be a standard library running on a single CPU. Some of the implications of choosing a serial programming model are:

1. for a given set of data, results will always be exactly the same regardless of how a particular computation was parallelized on a specific run
1. standard tools such as debuggers and profilers continue to work in the same way that they always have
2. IEEE conditions are presented to an application in exactly the same way regardless of whether a computation is performed serially or in parallel
3. DSSLIB always presents signals to an application in the same way every time
4. parallel machines or processors use exactly the same environment as the host machine

Given our choice of target users, one of the most important requirements is that all of the parallelization be automatic. When an application calls one of the parallel subroutines then DSSLIB determines how many processors to use, how to partition the data, and divides the work among the available processors. The number of processors to use is computed based on the size of the computation, expected performance from the network, expected performance from the other processors, and other factors. For a given computation, the number of

processors assigned may vary as an application runs due to changes in the factors that influence the number of processors assigned. However, DSSLIB partitions all computations in a way that guarantees that for a given computation the answer returned from DSSLIB will always be bitwise identical, regardless of the number of processors.

As part of doing the automatic parallelism, DSSLIB records certain performance information about each computation and continuously tunes itself as a program runs. In addition to making the automatic parallelism more efficient, this has the interesting side-effect that large applications will tend to get faster as they run because DSSLIB will have time to learn and adapt to the environment. For example, an application may learn that the network is more lightly loaded than expected and that the cost of communicating with remote processors is less than anticipated. As a result, it may choose to use more processors for future computations than it would have chosen by default. To illustrate this property, we modified the LINPACK 1000x1000 benchmark to solve six linear systems and record six times instead of just one. In that test, DSSLIB solved the sixth linear system 18% faster than it solved the first system because it was able to apply to the sixth computation things that it had learned about the environment while doing the earlier computations.

The fact that the parallelism is automatic allows DSSLIB to be incorporated into production code, even commercial software. While it is possible for a researcher to specify in advance a detailed and possibly very narrow description of the types of problems to be solved, a commercial software package will be presented with a variety of problems or varying sizes and shapes. The researcher writing a specialized code to solve a narrow set of problems may know in advance a close estimate to the optimal number of processors, but a commercial package cannot have such assumptions built in. The researcher may have available the luxury of being able to schedule a block of time on an empty or nearly empty system. A production code may be run in such an environment, but it also needs to work well in a shared environment. The automatic and adaptive parallelism in DSSLIB allows the flexibility required of commercial or production software.

Of course, one of the characteristics of a serial programming model is that a program generally will not be adversely affected by problems in the network or on other computers. To support this aspect of a serial programming model, DSSLIB has been built completely fault tolerant. If one or more parallel machines crash due to problems in hardware, software, or network then DSSLIB will detect the problem and automatically restructure the computation so that it will complete correctly. Further, it will restructure the computation in such a way as to guarantee that the answer from the restructured computation will be bitwise identical to the answer that would have been computed under normal conditions. Of course, failures on the host machine can hurt the application, but this also is consistent with the serial programming model. As with the automatic parallelization above, the compensation for faults or errors is automatic and does not require any special code on the part of the user.

Detected errors or faults, such as IEEE conditions, are presented to an application in exactly the same way every time regardless of whether a computation is performed serially or in parallel. For example, if an application attempts to solve a singular linear system then the subroutine DGESL will divide by zero. (This is standard documented LINPACK behavior, not a DSSLIB problem.) Most other parallel systems will not return a divide by zero indication to a user's application. DSSLIB will return divide by zero or any other condition to a user's application exactly as if it had been run on a single CPU. Also, DSSLIB always presents multiple signals to an application in the same order every time. Consider a computation that will be performed in parallel in which one parallel machine will divide by zero and another will get an overflow. DSSLIB guarantees that those signals will always be presented to the user's application in the same order every time, just as they would be if the computation were performed on a single CPU. DSSLIB has no race conditions common in other parallel systems.

Parallel machines or processors use exactly the same environment as the host machine. For example, if an application changes the IEEE rounding mode to round towards zero instead of round to nearest then all parallel machines or processors will round towards zero. Other parallel processing systems do not ensure that parallel computations are performed in the environment requested by an application.

RESULTS

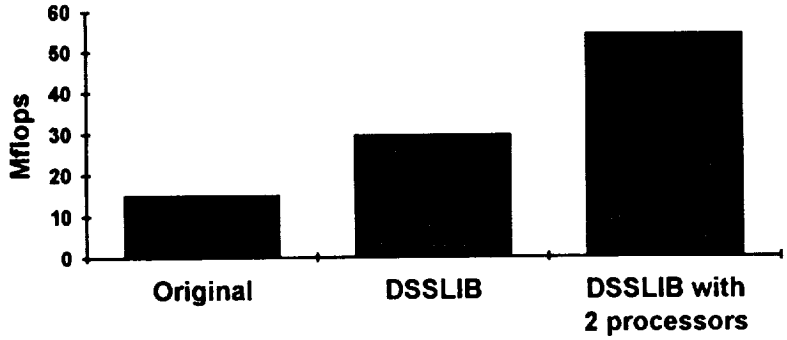
Of course, the acid test of any parallel or distributed system is speed. If it is not fast then none of its other characteristics are particularly interesting. It is even more helpful if the system is fast on real applications, rather than running well only on a selected set of benchmarks. DSSLIB is fast. Measurements on applications with run times ranging from 90 seconds to 12 hours shows that it delivers a pleasing level of performance for a reasonable variety of applications. Further, DSSLIB satisfies the requirement that it run well in a variety of hardware and software environments with no changes required of the user.

The hardware configurations in which DSSLIB was tested for this paper include both parallel and distributed processing where processors are defined to be parallel if they share a common memory. The shared memory machine used for this paper was a dual processor SPARCstation 10 with a 40 MHz SuperSPARC processor. Processors are defined to be distributed if they do not share a common memory but are linked via some network such as Ethernet or FDDI. Distributed processing machines used for this paper were single processor SPARCstation IPXs linked by a network. Networks used were the standard Ethernet and the SBUS FDDI product from Network Peripherals.

Both fine- and coarse-grain parallelism was tested. Fine-grain parallelism was done by making best use of the parallelism between the floating point and integer CPUs, and also between the independent add and multiply units in the floating point unit. On the SS10, additional fine-grain parallelism was measured by taking advantage of the multiple instruction per cycle capability, though this was limited by the fact that floating point instructions launch one at a time.

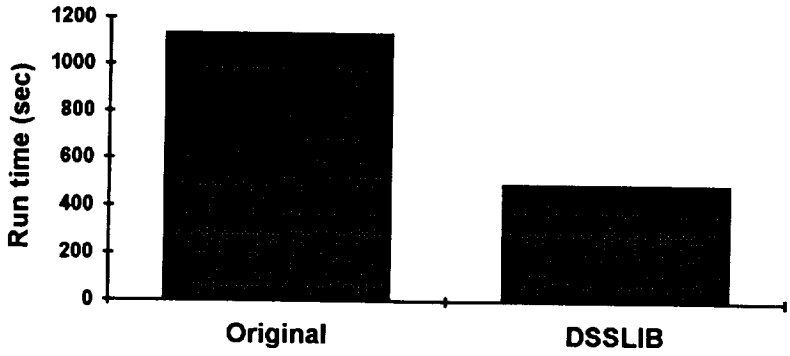
The software on which DSSLIB was tested included a matrix multiplication benchmark, a small image processing program written in IDL[1] that has a run time of 90 seconds, an artificial neural network written in FORTRAN 77 with run-times of 19 and 58 minutes for two data sets, and a discrete ordinate radiative transfer program [7] written in FORTRAN 77 with a run-time of 12 hours.

We ran the matrix multiplication benchmark on a single-CPU SPARCstation 10 model 40, then again on a dual processor machine. This benchmark simply generates 400x400 matrices and computes $\alpha AB + \beta C \rightarrow C$. The comparison below shows the speed of the standard form of DGEMM from netlib and the speed of the same subroutine from DSSLIB. The DSSLIB subroutine is timed on one and two processors where the single processor run takes advantage of only fine-grain parallelism and the two processor run takes advantage of all levels of parallelism.

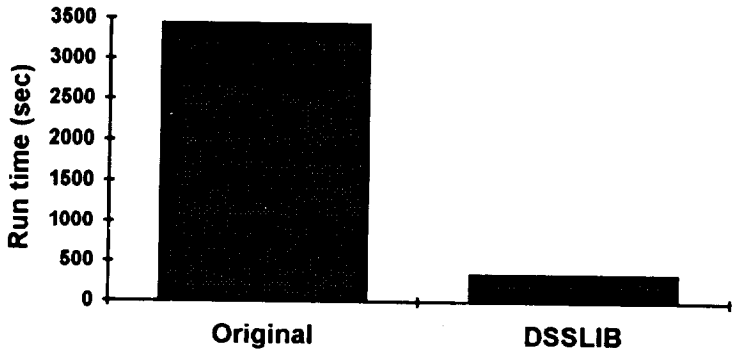


[1]IDL is an interpreted data modeling language from Research Systems, Inc. It appears to a user to be nearly identical in most ways to PV-WAVE from Visual Numerics, and the results reported for IDL are virtually identical to the results of similar experiments with PV-WAVE.

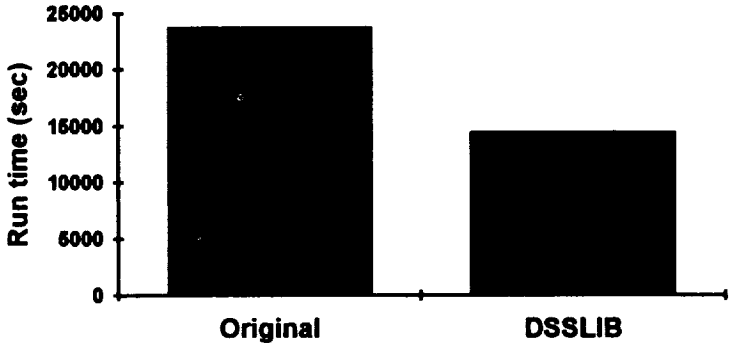
The neural network tested was part of a NASA project to classify cloud formations extracted from satellite data. The large Landsat data set was first reduced to a set of feature vectors extracted during a preprocessing phase and these feature vectors were used as input to the neural network. The neural network then classified the cloud formations in the image according to the information found in the feature vectors. The graph below shows the results obtained with a small data set. This data set is sufficiently small that it runs on a single CPU and so uses only fine-grain parallelism. The graph shows wall clock run time, so smaller values indicating shorter run times are better.



As one can clearly see from the graph above, the introduction of DSSLIB had a significant positive effect on the performance of the program, even for modest-sized data sets. Based on these results, Logar and Corwin decided to eliminate the data reduction in the preprocessing phase and send the raw satellite data directly to the neural network. In its present form, the neural network is limited by the size of the main memory on the Sun workstation rather than by the speed of the Sun CPU. The results below are for two Sun IPX workstations linked by an SBUS FDDI card from Network Peripherals.



DISORT is an application available used by NASA Goddard Space Flight Center to compute the thermal budget of a two-dimensional multi-layer region of the Earth's atmosphere. Each individual horizontal layer is required to be homogeneous, but different layers may have different characteristics. This program is dominated by eigenvalue computations done with modified subroutines from EISPACK. It also uses a significant amount of CPU time on LINPACK and various matrix algebra computations from the BLAS libraries. Because the EISPACK subroutines had been modified, there is no safe way to replace them with LAPACK subroutines, which is what one would usually do. However it is possible to insert a few BLAS calls into the eigenvalue subroutines and other places in the program. We did this in a way that provided some speed improvement, but did not compromise the accuracy or portability of the program. The results are shown in the following graph.



As one would expect from a program dominated by eigenvalue computation, parallelism provides significant speedup, but the improvement is not a factor of two for two processors. Nevertheless, DSSLIB cut three hours from a seven hour run using two processors. The customer is now able to make runs that would have been prohibitively expensive before these changes.

SUMMARY

DSSLIB is a library built on a parallel processing system that presents to an application a serial programming model. This serial programming model simplifies the development of a parallel application because it hides from the programmer the difficult details of parallelization. It also allows DSSLIB to be incorporated into production or commercial software because it is able to adapt to the variety of configurations and environments in which such software is used.

BIBLIOGRAPHY

Anderson, E., Z. Bai, C. Bischof, J.W. Demmel, J.J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK User's Guide, Society of Industrial and Applied Mathematics, Philadelphia, Pa., 1992.

Bernard G. et. al. Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment. IEEE Transactions on Software Eng., **15** (12), December 1989, pp 1567-1578.

Dongarra, J.J., C.B. Moler, J.R. Bunch, G.W. Stewart. LINPACK User's Guide, Society of Industrial and Applied Mathematics, Philadelphia, Pa., 1979.

Dongarra, J.J., J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, **16** (1990), pp 1-17.

Dongarra, J.J., J. DuCroz, S. Hammarling, and R.J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, **14** (1988), pp 1-17.

Lawson, C.L., Hanson, R.J., Kincaid, D., and Krogh, F.T. Basic Linear Algebra Subprograms for FORTRAN Usage, ACM Transactions on Mathematical Software, **5** (1979), pp 308-323.

Stamnes, K., Tsay, S., Wiscombe, W., Jayaweera, K. Numerically Stable Algorithm for Discrete-Ordinate-Method Radiative Transfer in Multiple Scattering and Emitting Layered Media. *Applied Optics*, **27** (1988), pp 2502-2509.

Sun Microsystems, Inc. Network Programming Guide. Revision A, March 1990, pp 33-167.

HIGH-SPEED DATA SEARCH

James Driscoll
Department of Computer Science
University of Central Florida
Orlando, Florida 32816, USA

ABSTRACT

The high-speed data search system developed for KSC incorporates existing and emerging information retrieval technology to help a user intelligently and rapidly locate information found in large textual databases. This technology includes: natural language input; statistical ranking of retrieved information; an artificial intelligence concept called semantics, where "surface level" knowledge found in text is used to improve the ranking of retrieved information; and relevance feedback, where user judgments about viewed information are used to automatically modify the search for further information. Semantics and relevance feedback are features of the system which are not available commercially. The system further demonstrates a focus on paragraphs of information to decide relevance; and it can be used (without modification) to intelligently search all kinds of document collections, such as collections of legal documents, medical documents, news stories, patents, and so forth. The purpose of this paper is to demonstrate the usefulness of statistical ranking, our semantic improvement, and relevance feedback.

INTRODUCTION

Locating information using large amounts of natural language documents (text) is an important problem. Examples at KSC are searching press releases and numerous other documents to quickly answer media questions, accessing bulky manuals and schematics compactly stored on a CD via a laptop computer, and retrieving digital images by means of their catalog descriptions.

The primary intent of our work has been to provide convenient access to information contained in the numerous and large public information documents maintained by Public Affairs at NASA Kennedy Space Center (KSC). The documents maintained by Public Affairs at NASA KSC consist of press releases, and other printed information created at KSC, and other NASA offices using various wordprocessors. There are also documents from outside contractors, such as Rockwell, which produces the "NASA National Space Transportation System Reference" more often called the "shuttle manual." During a launch at KSC, about a dozen NASA employees access these printed documents to answer media questions. The planned document storage for NASA KSC Public Affairs is around 300,000 pages (approximately 900 megabytes of disk storage).

Current commercial text retrieval systems focus on the use of keywords to search for information. These systems typically use a Boolean combination of keywords supplied by the user to retrieve documents. In general, the retrieved documents are not ranked in any order of importance, so every retrieved document must be examined by the user. This is a serious shortcoming when large collections of documents are searched.

The QA system is a high-speed data search system developed jointly by NASA KSC, the University of Central Florida, and Florida High Technology and Industry Council. It is a statistically based text retrieval system which ranks retrieved documents according to their statistical similarity to a user's request. Statistically based systems provide many advantages over traditional Boolean retrieval methods, especially for users of such systems, mainly because they allow natural language input. These systems have been a research success for over twenty years [9]. However, the transfer of this retrieval technique into large operational systems has been very slow because, until recently, there was no evidence that statistical ranking could be done in real-time on large document collections [4]. There are only three commercial systems in the United States which allow natural language input and perform statistical ranking of retrieved information [2].

The QA System incorporates two other features which are not available in any commercial text retrieval system, but have been shown to dramatically improve the statistical ranking of retrieved information. The first is an artificial intelligence concept called semantics, where "surface level" knowledge found in text is used to improve the ranking of retrieved information. The second is relevance feedback, where user judgments concerning viewed information are used to automatically modify the search for more information.

The QA System is very close to being a commercial product. It has been used to participate in a (first) Text Retrieval Conference (TREC-1) managed by the National Institute of Standards and Technology (NIST). Our participation in TREC-1 was funded by the Defense Advanced Research Projects Agency (DARPA). Participation in TREC-1 has enabled the QA System to be tested in an environment other than answering questions, and applied to databases other than aerospace text collections [3].

Conventional information retrieval using statistical ranking is demonstrated first in this paper. Demonstrations of improved statistical ranking due to the use of semantics within the QA System are then presented for comparison. This is followed by a demonstration of relevancy feedback within the QA System. In all demonstrations, the focus on paragraphs of information for retrieval will be evident. Finally, the issues of platforms and high-speed for the QA System are discussed in the Conclusion.

CONVENTIONAL INFORMATION RETRIEVAL

Finding relevant text and ranking the retrieved documents is not new and there are commercial systems which already perform this activity; we mention here an example of ranked, relevant text retrieval. For a demonstration to NASA KSC, the 1000 page shuttle manual was used by considering each paragraph of the manual as a document. This resulted in a collection of 5143 documents. A commercial hypertext IR system called SPIRIT [11] was used to automatically index the collection and provide natural language access. SPIRIT is a mainframe system. Running on an IBM 4381, SPIRIT required three and one-half hours of clock time to index the collection of 5143 documents.

Figure 1 is a screen generated by SPIRIT for asking the natural language query

What are the dimensions of the cargo area in the shuttle?

Figure 2 is a screen generated by SPIRIT revealing a ranked list of 245 relevant documents with CLASS 1 being the most relevant. Figure 3 is a screen generated by SPIRIT revealing the first document in CLASS 6, which contains the answer to the query. This paragraph was found by reading the single paragraph in CLASS 1 first, then the single paragraph in CLASS 2, and so on until the answer was read in the tenth paragraph.

NATURAL LANGUAGE QUERY ON THE SHUTTLE BASE

<1>: What are the dimensions of the cargo area in the shuttle?

EMPTY WORDS: What, are, the, of, the, in, the.

KEYWORDS: dimensions, cargo, area, shuttle.
***

Figure 1. Natural Language Query to the SPIRIT System.

CLASSES	NB DOCS	KEYWORDS
1	1	dimensions, cargo, shuttle.
2	1	cargo, area, shuttle.
3	1	dimensions, area.
4	2	dimensions, shuttle.
5	4	cargo, area.
6	30	cargo, shuttle.
7	12	area, shuttle.
8	7	dimensions.
9	40	cargo.
10	147	area.
BOTTOM OF LIST		

Figure 2. Document Classes Generated by the SPIRIT System.

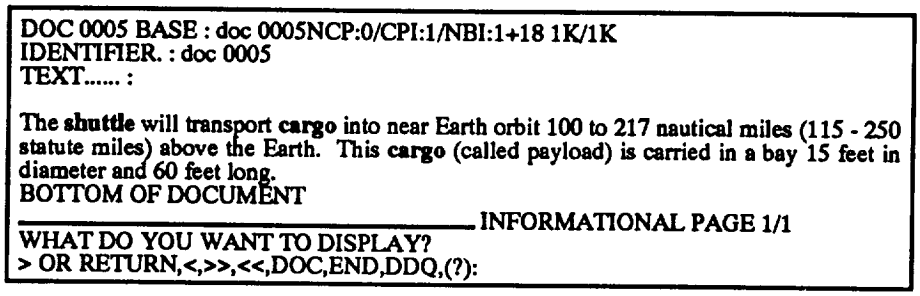


Figure 3. Document Display by the SPIRIT System.

Note that performance in this Question/Answer environment is measured by counting how many documents were examined to find the document containing the answer. This is not the usual way of measuring the performance of IR systems, but it is very appropriate for a Question/Answer environment.

The underlying principles and algorithms of automated IR systems like SPIRIT are well-known. Terms used as document identifiers are keywords modified by various techniques such as stop lists (removal of useless or empty words), stemming, synonyms, and query reformulation. Here, we present basic concepts associated with the calculation of weighting factors.

The calculation of the weighting factor (w) for a term in a document is a combination of term frequency (tf), document frequency (df), and inverse document frequency (idf). The basic term definitions are as follows:

tf_{ij} = number of occurrences of term T_j in document D_i

df_j = number of documents in a collection which contain T_j

$idf_j = \log\left(\frac{N}{df_j}\right)$, where N = total number of documents

$w_{ij} = tf_{ij} \cdot idf_j$.

When an IR system is used to query a collection of documents with t terms, the system computes a vector Q equal to $(w_{q1}, w_{q2}, \ldots, w_{qt})$ as the weights for each term in the query. The retrieval of a document with vector D_i equal to $(d_{i1}, d_{i2}, \ldots, d_{it})$ representing the weights of each term in the document is based on the value of a similarity measure between the query vector and the document vector. A common similarity function which normalizes the the similarity coefficient in case of different document sizes is the following:

$$sim(Q,D_i) = \frac{\Sigma_{j=1}^{t} w_{qj} \cdot d_{ij}}{\sqrt{\Sigma_{j=1}^{t} d_{ij}^2}} \qquad (1)$$

It is important to note that the calculation of a similarity coefficient for each document and the ranking of the documents relevant to a query is rather time consuming. This is due to the summations that occur in the above formula and the fact that every document that has a term in common with a given query must be considered. The main problem with text retrieval using statistical ranking has been the time required to produce the document ranking given a query. Consequently, query response time has been typically slow.

SEMANTIC APPROACH

Although the basic statistical ranking approach (as demonstrated by SPIRIT) has shown some success in regard to natural language queries, it ignores some valuable information. We now know that these systems can be further improved by imposing a semantic data model upon the "surface level" knowledge found in text.

Semantic Modeling

Semantic modeling was an object of considerable database research in the late 1970's and early 1980's [1]. Essentially, the semantic modeling approach identified concepts useful in talking informally about the real world. These concepts included the two notions of entities (objects in the real world) and relationships among entities (actions in the real world). Both entities and relationships have properties.

The properties of entities are often called attributes. There are basic or surface level attributes for entities in the real world. Examples of surface level entity attributes are Size, Color, and Position. These properties are prevalent in natural language. For example, consider the phrase "large, black book on the table," which indicates the Size, Color, and Position of a book.

In linguistic research, the basic properties of relationships are discussed and called thematic roles. Thematic roles are also referred to in the literature as participant roles, semantic roles, and case roles. Examples of thematic roles are Beneficiary and Time. Thematic roles are prevalent in natural language, they reveal how sentence phrases and clauses are semantically related to the verbs in a sentence. For example, consider the phrase "purchased for Mary on Wednesday" which indicates who benefited from a purchase (Beneficiary) and when a purchase occurred (Time).

Consider the following query:

How long does the payload crew go through training before a launch?

The basic statistical approach dismisses the following words in the query as empty: "how", "does", "the", "through", "before", and "a". Some of these words contain valuable semantic information. The following list indicates some of the thematic roles triggered by a few of the words in the above query:

long ⇒ Duration, Time
through ⇒ Location/Space, Motion With Reference To Direction, Time
before ⇒ Location/Space, Time

As another example, consider the query in Figure 1:

What are the dimensions of the cargo area in the shuttle?

The keyword "dimensions" indicates the attribute General Dimensions and the keyword "area" indicates both the thematic role Location/Space and the attribute General Dimensions. It would be reasonable to expect that the document that answers this query would have words in it that fall in the category of General Dimensions.

The primary goal of the QA System has been to detect thematic and attribute information contained in natural language queries and documents. When the information is present, the system uses it to help find the most relevant paragraph to a query. In order to use this additional information, the basic underlying concept of text relevance was modified. The major modifications include the addition of a lexicon with thematic and attribute information, and a modified computation of the similarity measure given in (1).

The Semantic Lexicon

The QA System uses a thesaurus as a source of semantic categories (thematic and attribute information). For example, Roget's Thesaurus contains a hierarchy of word classes to relate word senses [5]. For our research, we have selected several classes from this hierarchy to be used for semantic categories. We have defined thirty-six semantic categories as shown in Figure 4.

In order to explain the assignment of semantic categories to a given term using Roget's Thesaurus, consider the brief index quotation for the term "vapor":

vapor		
n.	fog	404.2
	fume	401
	illusion	519.1
	spirit	4.3
	steam	328.10
	thing imagined	535.3
v.	be bombastic	601.6
	bluster	911.3
	boast	910.6
	exhale	310.23
	talk nonsense	547.5

Thematic Role Categories
Accompaniment
Amount
Beneficiary
Cause
Condition
Comparison
Conveyance
Degree
Destination
Duration
Goal
Instrument
Location/Space
Manner
Means
Purpose
Range
Result
Source
Time

Attribute Categories
Color
External and Internal Dimensions
Form
Gender
General Dimensions
Linear Dimensions
Motion Conjoined with Force
Motion in General
Motion with Reference to Direction
Order
Physical Properties
Position
State
Temperature
Use
Variation

Figure 4. Thirty-Six Semantic Categories.

The eleven different meanings of the term "vapor" are given in terms of a numerical category. We have developed a mapping of the numerical categories in Roget's Thesaurus to the thematic role and attribute categories given in Figure 4. In this example, "fog" and "fume" correspond to the attribute State; "steam" maps to the attribute Temperature; and "exhale" is a trigger for the attribute Motion with Reference to Direction. The remaining seven meanings associated with "vapor" do not trigger any thematic roles or attributes. Since there are eleven meanings associated with "vapor," we indicate in the lexicon a probability of 1/11 each time a category is triggered. Hence, a probability of 2/11 is assigned to State, 1/11 to Temperature, and 1/11 to Motion with Reference to Direction. This technique of calculating probabilities is being used as a simple alternative to a corpus analysis. It should be pointed out that we are still experimenting with other ways of calculating probabilities.

Extended Computation of the Similarity Measure

The probabilistic details of a semantic lexicon and the computation of semantic weights can be found in [13]. A detailed explanation of the manner in which the QA System combines semantic weights and keyword weights can be found in [12].

Essentially we treat semantic categories like indexing terms, and the probabilities introduced by a semantic lexicon mean that the frequency of a category in a document becomes an expected frequency and the presence of a category in a document becomes a probability for the category being present. This means that the document frequency for a category becomes an expected document frequency, and this enables an inverse document frequency to be calculated for a category.

So the computation of a similarity coefficient as shown in (1) can be used, but now the summations in the formulas include semantic categories in the documents as well as terms in the documents. In other words,

$$sim(Q,D_i) = \frac{\sum_{j=1}^{t} w_{qj} \cdot d_{ij} + T \sum_{j=t+1}^{t+s} w_{qj} \cdot d_{ij}}{\sqrt{\sum_{j=1}^{t} d_{ij}^2 + B \sum_{j=t+1}^{t+s} d_{ij}^2}} \qquad (2)$$

where $s = 36$ is the number of semantic categories, and T and B are scaling factors for adjusting the blend.

SEMANTIC IMPROVEMENT

The QA System has demonstrated a noticeable semantic improvement using the similarity function in (2). Consider the same document collection and natural language query shown in the commercial system example of Figures 1, 2, and 3. Using the commercial system SPIRIT, ten paragraphs were read in order to find the answer to the following query:

What are the dimensions of the cargo area in the shuttle?

Considering the QA System, Figure 5 is a screen generated for asking this same natural language query. Figure 6 is a screen generated by the QA System graphically showing to the user the importance of the keywords found in the query. Figure 7 is a screen generated by the QA System graphically showing to the user the importance of semantic information found in the query. Notice the "importance" of the semantic category General Dimensions in the screen shown in Figure 7. This long bar means that the semantic category General Dimensions is present in the query and there are very few documents retrieved (using keywords) having this type of semantic content. Hence, the importance of the category.

Finally, Figure 8 is a screen generated by the QA System revealing the second paragraph found by proceeding through the ranked list of documents retrieved by the QA System for this query. The semantic information found in the query and displayed in Figure 7 is the reason the QA System ranked the answering paragraph second instead of tenth as did the SPIRIT system. Notice that the answering document in Figure 8 has several words in it which trigger the semantic category General Dimensions. We have lots of data like this and several technical papers which reveal a significant performance improvement due to semantic modeling in the NASA KSC Question/Answer environment.

For another example of semantic improvement, consider the shuttle manual and the query:

How fast does the orbiter travel on orbit?

This query is interesting for two reasons. One is that the words "orbiter" and "orbit" are rather frequent words in the shuttle manual so lots of paragraphs are retrieved. The other reason is that the word "fast" is used for reference to velocity or speed.

Figure 9 shows the number of paragraphs one must read to find a particular answering paragraph to this query for both a small and large collection of documents. In the small collection, the word "fast" does not occur at all and for the large collection, the word "fast" never occurs in an answering paragraph. Consequently, keyword only statistical ranking is never very good. But by using semantics, the word fast causes a similarity to paragraphs using the words velocity or speed. Consequently, semantics improves the statistical ranking of an answering paragraph. Different blends of keywords and semantics are shown using the similarity function in (2).

RELEVANCE FEEDBACK

It has been pointed out that conventional IR systems have a limited recall [6]; only a few relevant documents are retrieved in response to user queries if the search process is based solely on the initial query. This indicates a need to modify (or reformulate) the initial query in order to improve performance. It is customary to search the relevant documents iteratively as a sequence of partial search operations. The results of earlier searches can be used as feedback information to improve the results of later searches. One possible way to do this is to ask the user to make a relevance decision on a certain number of retrieved documents. Then this relevance information can be used to construct an improved query formulation and recalculate the similarities between documents and query in order to re-rank them. This process is known as relevance feedback [7,8,9,10] and it has been shown experimentally to improve the performance of the retrieval system.

The basic assumption behind relevance feedback is that, for a given query, documents relevant to it should resemble each other in a sense that they have reasonably similar keyword vectors. This implies that if a retrieved document is identified as relevant, then the initial query can be modified to increase its similarity to such a relevant document. As a result of this reformulation, it is expected that more of the relevant documents and fewer of the nonrelevant documents will be extracted.

The automatic construction of an improved query is actually straightforward, but it does increase the complexity of the user interface and the use of the retrieval system, and it can slow down query response time. Essentially, the terms and semantic categories for documents viewed as relevant to a query can be used to modify the weights of terms and semantic categories in the original query. A modification can also be made using documents viewed as not relevant to a query. Experimental results show a very promising improvement for relevance feedback within the QA System.

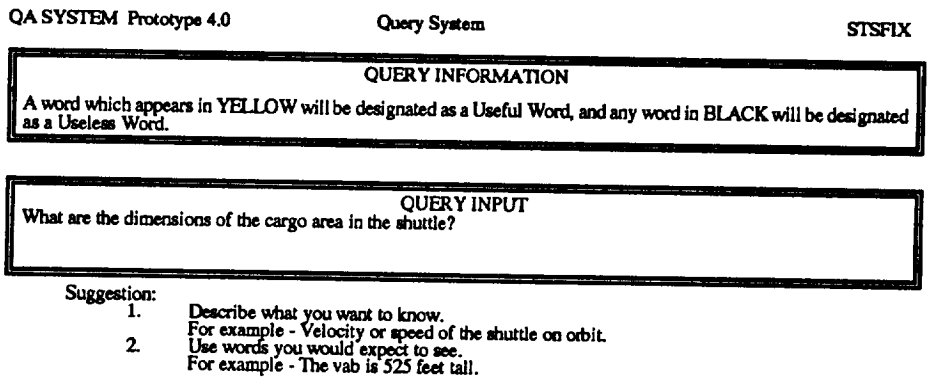


Figure 5. Natural Language Query to the QA System.

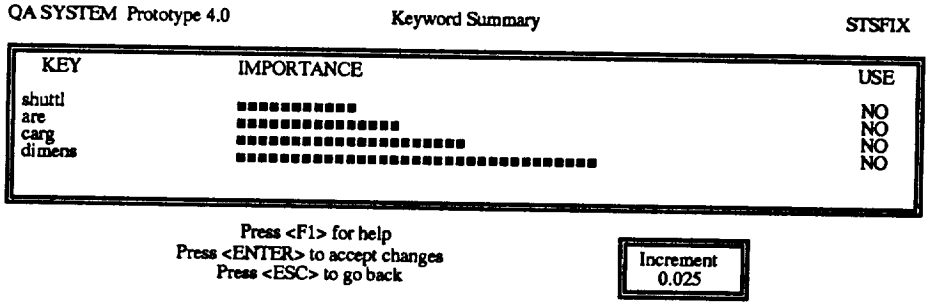


Figure 6. Keyword Summary by the QA System.

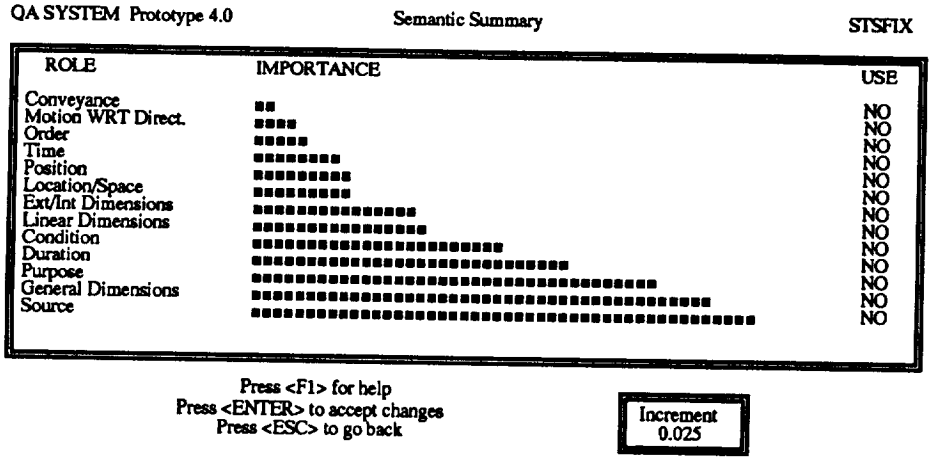


Figure 7. Semantic Summary by the QA System.

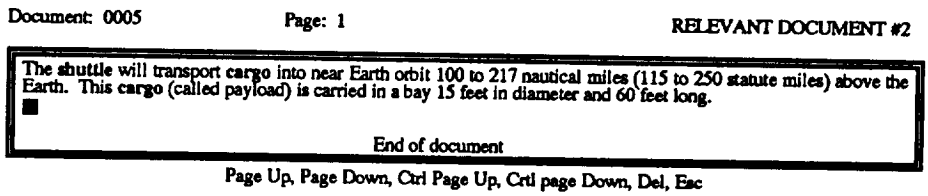


Figure 8. Document Display by the QA System.

	Keywords Only	$T-B=1.10206$ Blend of Keywords and Semantics	$T-B=8.0$ Blend of Keywords and Semantics
First 26 pages of the shuttle manual (160 documents)	19	4	2
The entire shuttle manual (5143 documents)	145	126	14

Figure 9. Number of paragraphs read to find a particular answering paragraph for: How fast does the orbiter travel on orbit?

Figure 10 provides an example using the first 26 pages of the shuttle manual and the query:

How fast does the orbiter travel on orbit?

Recall from Figure 9 that 19 paragraphs were read to find an answering paragraph. The document identifiers for these 19 paragraphs are shown in the left column of Figure 11 along with the notes that Document #13 and Document #16 were considered relevant to the original query, and Document #14 answered the query. All the other viewed documents were not relevant to the query.

If relevance feedback is selected within the QA System and the system is told to display two documents and then reformulate the query, then the documents shown in the right column are viewed. Each document viewed must be tagged as relevant or not-relevant. Document #14 shows up earlier in the statistical ranking primarily because Document #13 was tagged as relevant to the original query.

It is interesting to note that if one tags Document #14 (which answers the query) as relevant, then Document #87 is retrieved and it almost exactly answers the query. Document #87 would never be retrieved using just keywords without feedback because it has no keywords in common with the original query. Documents 13, 14, 16, 69 and 87 are shown in Figure 11. The keywords that these documents have in common with the original query are underlined. Clearly, Document 69 is not relevant to the original query.

CONCLUSION: PLATFORMS AND THE ISSUE OF HIGH SPEED

Originally, the QA System was restricted to an IBM compatible PC platform running under the DOS operating system and without the use of any other licensed commercial software such as a DOS extender. The QA System is implemented in Borland C and one version uses B+ tree structures for the inverted files. We felt the speed of the system and its storage overhead was not efficient so a hashing scheme was added to eliminate the use of B+ trees and provide codes for keywords. We expected this second version to have improved indexing time, storage, and retrieval speed.

Experiments revealed that indexing time of the QA System did not improve much. We were not surprised because the QA System is restricted under the PC DOS platform. This platform has a serious memory addressing restriction which results in memory page swapping and this seriously affects the speed of processing, especially during creation of the hashing table and index structures. The improvement in storage, however, was very impressive. It is very much matched to our objective which is to make our storage ratio of indexes to text, around 0.5. This is comparable to the ratio of very efficient, retrieval systems using statistical ranking.

Addressing the high speed issue, we now have the Borland C compiler for OS/2 so we expect to have a very high speed QA System running under OS/2 very soon. We are also in the process of converting the QA System to run in the UNIX environment. Figure 12 reveals achieved and projected run-time performances of the QA System on different operating system platforms. The DOS, B+ tree version of the system is shown in the upper left corner. Below (diagonally) are shown the OS/2, UNIX B+ tree and hashing versions of the QA System for different amounts of RAM. Indexing and typical query response times are shown for both a small (2.4 megabyte) and a large (1.2 gigabyte) document collection. Data for this chart was obtained in part from experiments performed for TREC-1 [3].

160 Documents

Answer can be found in Document 14, 87

Keywording			Relevance Feedback (view 2)		
1	69		1	69	
2	13	Relevant	2	13	Yes
3	82		3	82	
4	15		4	107	
5	123		5	85	
6	106		6	124	
7	85		7	16	Yes
8	124		8	14	Yes, Answer
9	21		9	87	Yes, Answer
10	23				
11	24				
12	83				
13	31				
14	26				
15	16	Relevant			
16	84				
17	11				
18	12				
19	14	Answer			

:

never get 87 (no query words in 87)

Figure 10. Relevance Feedback Improvement for the Query: How fast does the orbiter travel on orbit?

Document 13

The two orbital maneuvering system engines are used to place the orbiter on orbit, for major velocity maneuvers on orbit and to slow the orbiter for re-entry, called the deorbit maneuver. Normally, two orbital maneuvering system engine thrusting sequences are used to place the orbiter on orbit, and only one thrusting sequence is used for deorbit.

Document 14

The orbiter's velocity on orbit is approximately 25,405 feet per second. The deorbit maneuver decreases this velocity approximately 300 feet per second for re-entry.

Document 16

For deorbit, the orbiter is rotated tailfirst in the direction of the velocity by the primary reaction control system engines. Then the orbital maneuvering system engines are used to decrease the orbiter's velocity.

Document 69

- Atlantis (OV-104), after a two-masted ketch operated for the Woods Hole Oceanographic Institute from 1930-1966, which traveled more than half a million miles in ocean research.

Document 87

Entry interface is considered to occur at 400,000 feet altitude approximately 4,400 nautical miles (5,063 statute miles) from the landing site and at approximately 25,000 feet per second velocity.

Figure 11. Documents 13, 14, 16, 69, and 87. Keywords in common with the original query are underlined.

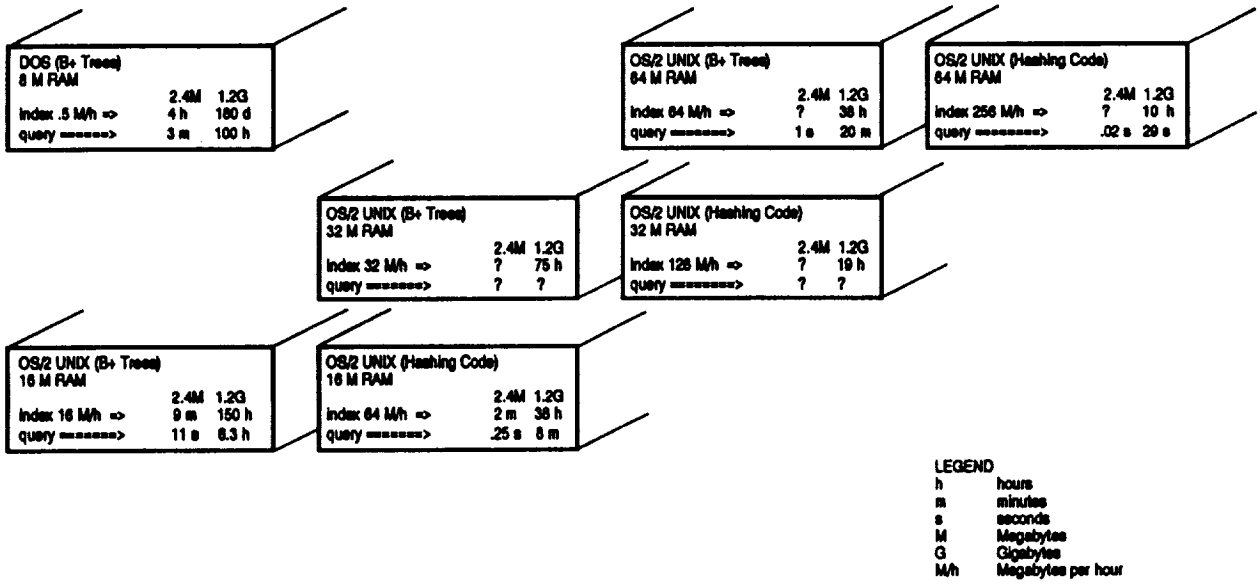


Figure 12. Run-Time Performance of the QA System.

References

[1] C. Date, *An Introduction to Database Systems*, Vol. I, Addison Wesley, 1990.

[2] Delphi Consulting Group, 1991, *Text Retrieval Systems: A Market and Technology Assessment*, 266 Beacon Street, Boston, MA, 1991.

[3] J. Driscoll, J. Lautenschlager and M. Zhao, "The QA System," *Proc. of the First Text Retrieval Conference (TREC-1), NIST Special Publication 500-207* (D. K. Harman, editor), March, 1993.

[4] D. Harman and G. Candela, "Retrieving Records from a Gigabyte of Text on a Minicomputer Using Statistical Ranking," *JASIS*, Vol. 41, pp. 581-589. 1990.

[5] *Roget's International Thesaurus*, Harper & Row, New York, Fourth Edition, 1977.

[6] G. Salton, *Automatic Information Organization and Retrieval*, McGraw-Hill, 1968.

[7] G. Salton, *The Smart Retrieval System—Experiments in Automatic Document Processing*, 1971.

[8] G. Salton, E. A. Fox, and E. Voorhees, "Advanced Feedback Methods in Information Retrieval," *JASIS*, Vol. 36, pp. 200-210, 1985.

[9] G. Salton, *Automatic Text Processing*, Addison-Wesley, Reading, MA, 1989.

[10] G. Salton and C. Buckley, "Improving Retrieval Performance by Relevance Feedback," *JASIS*, Vol. 41, pp. 288-297, 1990.

[11] *SPIRIT Version 2.1 User's Manual*, SYSTEX Company, Ferme Du Moulon, 91190 Gif Sur Yvette, France (French Edition), May 1986.

[12] D. Voss and J. Driscoll, "Text Retrieval Using a Comprehensive Semantic Lexicon," *Proceedings of ISMM First International Conference on Information and Knowledge Management (CIKM-92)*, Baltimore, MD, November 1992.

[13] E. Wendlandt and J. Driscoll, "Incorporating a Semantic Analysis into a Document Retrieval Strategy," *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, Chicago, IL, pp. 270-279, October 1991.