PLIERS: A Parallel Information Retrieval System using MPI

A. MacFarlane[1,2], J.A.McCann[1], S.E.Robertson[1,2]
[1] School of Informatics, City University, London EC1V OHB
[2] Microsoft Research Ltd, Cambridge CB2 3NH

Abstract. The use of MPI in implementing algorithms for Parallel Information Retrieval Systems is outlined. We include descriptions on methods for Indexing, Search and Update of Inverted Indexes as well as a method for Information Filtering. In Indexing we describe both local build and distributed build methods. Our description of Document Search includes that for Term Weighting, Boolean, Proximity and Passage Retrieval Operations. Document Update issues are centred on how partitioning methods are supported. We describe the implementation of term selection algorithms for Information Filtering and finally work in progress is outlined.

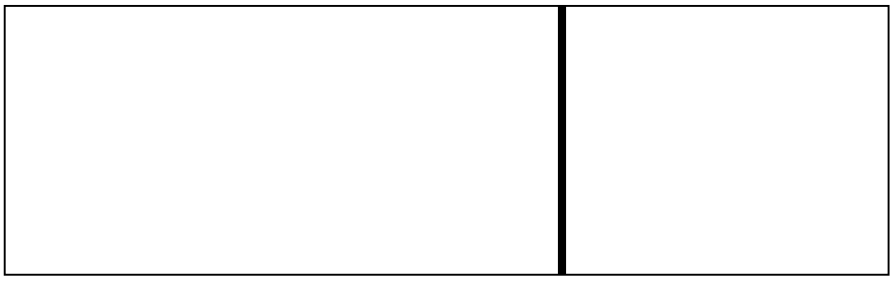
1. Introduction

We describe how MPI and the facilities that it provides are used to implement Parallel Information Retrieval Systems. Development work has been done at City University and further work is being continued at Microsoft Research in Cambridge on the PLIERS (ParaLLel Information rEtrieval System) on which our description is based. A particular interest has been the production of a Parallel IR system which is portable: much of the previous work in the area has produced methods and systems which cannot be ported between different architectures [1]. One of the main reasons MPI was chosen as the mechanism for Message Passing was that it provided this facility. Another was that collective operations such as broadcast, Scatter and Gather are very useful in a transaction processing context.

Much of the work done on PLIERS is either heavily influenced or based on work done on the Okapi system at City University [2]; a uni-processor based IR system. This includes methods for Indexing (discussed in section 3) methods for Document Search (section 4), Document Update (section 5) and Information Filtering (section 6). We also include a brief description of Inverted files and how they are organised in Parallel IR systems (section 2) and describe work currently in progress (section 7). A summary is given is section 8.

2. IR and Inverted File Organisations

IR or Information Retrieval is concerned with the delivery of relevant documents to a user. We restrict our discussion to textual data. Text Retrieval systems use Indexes in the form of Inverted files. Inverted files are typically split up into to main parts: a keyword/dictionary file and a Postings file (also known as the Inverted list): see figure 1 for a simplified example.



Dictionary File			Postings File		
Word	*Postings*	*Ptr*	*Id*	*Freq*	*Position List*
lamb	2	→	15	1	[1]
......			09	7	[1,2,3,4,5,6,7]
......					
mary	2	→	15	1	[1]
...			09	6	[2,3,4,7,8,9]

Figure 1 - An Example Inverted File

The dictionary file stores keywords found in the text collection together with number of documents in which the keyword occurs and a pointer to a list of document records in the Postings file. Each posting list may contain data on the positions of words for each document. Two main approaches have been proposed for distributing Inverted Indexes to disks: *DocId* and *TermId* partitioning [3]: figure 2 gives a simple example of them. The *DocId* approach partitions the Index by document assigning a document to a single disk, while *TermId* assigns a unique term to a disk. PLIERS supports both type of partitioning methods and we outline the significance of partitioning methods on parallel IR algorithms below. A Document Map which contains such information as the Document Id, its length and location on disk is also utilised.

Dictionary Files				Postings File		
File	*Word*	*Postings*	*Ptr*	*Id*	*Freq*	*Position List*
1	lamb	2	→	15	1	[1]
				09	7	[1,2,3,4,5,6,7]
2	mary	2	→	15	1	[1]
				09	6	[2,3,4,7,8,9]

Figure 2a - Inverted File Partitioning Methods -*TermId*

Dictionary Files				Postings File		
File	*Word*	*Postings*	*Ptr*	*Id*	*Freq*	*Position List*
1	lamb	1	→	15	1	[1]
	mary	1	→	15	1	[1]
2	lamb	1	→	09	7	[1,2,3,4,5,6,7]
	mary	1	→	09	6	[2,3,4,7,8,9]

Figure 2b - Inverted File Partitioning Methods - *DocId*

We use an architecture which is well known in the Parallel Systems field namely the Shared Nothing architecture. In essence this means each node in the parallel machine has its own CPU, main memory and local disk. Since the Index cannot be kept in memory for most applications (particularly web search engines) we use the Shared Nothing architecture to parallelise I/O as well as processing.

3. Indexing

Indexing is the generation of the Inverted file given the particular text collection. The indexing consists of parsing the text to identify words, recording their occurrence in the dictionary and updating posting/position data in the Inverted list. Two methods for the use of Parallel Indexing on text collections have been implemented: Local build (section 3.1) and Distributed build (section 3.2). We describe how these build methods relate to the partitioning method below.

3.1 Local Build Indexing

With Local Build the text collection is split up into N sub-collections and distributed to local disks to each node. The process of Indexing is quite simple: N Indexers are applied to each sub-collection in parallel that can proceed independently and very little communication is needed. In fact MPI is hardly used at all in the method apart from the initial start up messages and a MPI_Barrier command to notify a timing process that all Indexers have finished their work. Only *DocId* partitioning is available for this type of build as all data and text for any given document is kept locally.

3.2 Distributed Build Indexing

Distributed build indexing keeps the text collection on one disk using a Text Farmer process to distribute documents to a number of Worker Indexer processes which do much the same work as the Indexers in Local build. The method can distribute single documents or text files, but in practice we distribute text files because it reduces the level of communication needed. The communication interaction between the Farmer and Worker uses the MPI_Ssend, MPI_Irecv and MPI_Waitsome functions. The distributed indexing starts with the Farmer giving each worker an initial text file for analysis together with a given set of document identifiers: MPI_Ssend is used. The farmer then waits for the worker to request either another file to index or a new set of document id's. We use MPI_Waitsome to wait for request for work and if more than one request is received they are recorded in a worker queue at the Farmer, so that no data is lost. The farmer keeps a buffer of asyncronous receives issuing a MPI_Irecv for a worker each time it has completed servicing a request. From the point of view of the farmer the file interaction is a simple MPI_Irecv / MPI_Ssend interaction whereas the document id interaction requires the exchange of document map data: the interaction is therefore MPI_Irecv/MPI_Recv/MPI_Ssend. When the text collection has been distributed, the farmer sends a termination notice using MPI_Ssend to each worker. This has different implications for different request types. If the request type is for more id's then a termination is not sent: the worker has further work to do.

Termination is only sent to the worker when a File is requested. The last set of map data can then be received at the farmer. The termination interaction is identical to the file server interaction. It should be noted that some problems were found when using MPI_Send in the termination interaction: the worker moves into a phase which requires a great deal of CPU and I/O resources which prevented the send map data message being sent on one MPI implementation sequentialising the termination process: MPI_Ssend solved the problem.

If the *DocId* partitioning method is used then the process stops, but a further process is needed for *TermId* called the Global Merge. The Global Merge is split up into three parts: collection of partition statistics for allocation to nodes, distribution of data to nodes given one of a number of criteria on those statistics and a local merge to create the file Index for that node. Only the first two parts use MPI functions. The Statistics collection part uses MPI_Gather to obtain one of the following quantities: number of words in a partition, collection frequency in a partition and term frequency in a partition. A heuristic is then applied to the chosen quantity, which allocates partitions to nodes. The data is then distributed to nodes by gathering files in partitioning order to the nodes which has been allocated that particular partition: MPI_Gather is used.

4. Document Search

Document search in terms of Inverted files is the submission of user search requests in the form of a Query and applying it to the index using specified operations. These operations can be explicit, for example AND in boolean logic, or implicit as found in many web search engines which use term weighting operations. Search in PLIERS is based on the Search Set method, which in essence works by applying merge operations to sets of postings taken from the Inverted File. Parallel search has two types of processes: a Top or interface process that communicates with the client and collates data, and a number of leaf processes that manage a given Inverted file fragment. The interaction between the Top and Leaf processes is as follows: the top process receives a query and broadcasts it (using MPI_Bcast) to the leaf processes; the leaf processes retrieve and merge sets, sending only as much data to the top as is needed using a gathering mechanism (MPI_Gather is used). The last point has particular significance for the partitioning method used and we discuss this for Boolean/Proximity operations (section 4.1), Term Weighting Operations (section 4.2), and Passage Retrieval (section 4.3).

4.1 Boolean and Proximity Operations

Boolean operations on search sets use the normal Union (OR), Intersection (AND) and Difference (AND NOT) methods found in set theory. Proximity operations are an extension on Boolean operators and provide a further restriction on search e.g. searching for two words which are adjacent to each other (message ADJ passing). Proximity operations use position data that may have the format field, paragraph, sentence and word positions.

DocId. Boolean or Proximity operations of any type can be applied to each set element because all data on a document is kept local. The Leaf result sets can then be gathered by the Top process which does a final OR merge to produce the result ready for presentation at the client.

TermId. For some simple operations such as OR the set merge and process interaction is identical to *DocId* partitioning query service. However in many cases the final result cannot be computed until all data has been gathered e.g. all Proximity operations. This means that all search sets much be transmitted to the Top process for it to do all the set merges: parallelism is therefore restricted on this type of partitioning. It should be noted that unlike *DocId*, in *TermId* some Leaf nodes may have no work to do if a given query has no keywords in that partition: this has implications for load balance which affects all search set operations.

4.2 Term Weighting Operations

Users apply these operations by specifying a natural language query e.g. "parallel text retrieval". Term Weighting operations assign a weight to a keyword/document pair and do a set merge in order to calculate a total score for each document. Term Weighting operations have following phases: retrieve sets for the keyword, weight all sets given collection statistics, merge the sets accumulating scores for documents and sort for final results. With both types of partitioning method, set retrieve can be done in parallel, but different strategies are needed for the other phases. It should be noted that map data statistics must be exchanged on Local build Indexes: using MPI_Reduce followed by MPI_Bcast does this. This communication is not needed for distributed build as the map data is replicated.

DocId. Certain collection statistics such as collection frequency for a keyword do not reside on one node. Therefore the top process must gather this data and the information recorded in the query to be sent back to the Leaf process. Currently MPI_Gather is used, but alternatively MPI_Reduce could be used or MPI_Allreduce that would restrict message exchange between leafs. When weighting is done local set merges can be done in parallel and sorts initiated on local set results. The top N results are then picked off by each Leaf and sent to the Top node: MPI_Gather is used. This method restricts the amount of data to be transmitted to the Top node. The final result is generated by picking of the top set off all leaf results based on descending weight order. This result can then be delivered to the client.

TermId. Term weighting can proceed in parallel without any communication and set merges can be applied in parallel to generate the result set for a Leaf. However sorts must be done at the top node, as the weight for any given document has not been generated. The Top process (MPI_Gather is used) therefore gathers the intermediate result sets and a merge is applied to generate the final result set. The sort can then be applied and the top set of results picked off ready for presentation to the client. More communication is needed in this method, and no parallelism is available for one of the most important aspects of Weighting operations, namely the sort.

4.3 Passage Retrieval

Passage Retrieval search is the identification of a part of a document which may be relevant to a user e.g. in a multiple subject document. A computationally intensive algorithm for Passage Retrieval has been implemented [4] that is of order $O(n^3)$ unoptimised (this is reduced to $O(n^2)$ using various techniques). The algorithm works by iterating through a contiguous list of text atoms (paragraphs have been used) and applying a Term Weighting function to each passage, recording the best weighted passage. We outline two methods on *DocId* partitioning for this type of search: a Term Weighting operation is applied and Passage retrieval is only done on the top set of results.

Method 1. With this method, the Passage Retrieval algorithm is applied to the top set of results locally at each node. Currently this is the top 1000 documents on each node. Therefore many more documents are examined than in the next strategy. A second sort is applied at each node to produce a final result set and the same operation described in *DocId* weighting search is used to generate the final result set.

Method 2. This method applies the Passage Retrieval algorithm only on the top 1000 documents found in Weighting search over the whole collection. We use a document accumulator mechanism for this. The top set identified during weighted search is broadcast (using MPI_Bcast) and each Leaf node generates Passage Retrieval scores for its documents. The results are then gathered by the Top node (using MPI_Gather) and a final sort is applied to generate the final result.

5. Document Update

Document update is the addition of new documents to an existing Index. We reuse code from the Indexing module but use it in a Index maintenance context. The update of Inverted Indexes is very resource intensive because of the need to keep individual Inverted Lists in contiguous storage to enable efficient search: I/O is a significant cost in IR systems. We have hypothesized that parallelism may reduce Update costs and have implemented a method on both types of partitioning. The search topology is used for update: this allows for both search and update transaction service. Updates and search requests are delivered to the system. New documents are recorded in a buffer and the Leaf nodes reach distributed agreement on a re-organisation of the whole Indexing using MPI_Allreduce if one node has reached its buffer limit: when this condition is met all further transactions are locked out until the Index has been reorganised. We are currently considering various aspects of this algorithm.

6. Information Filtering

Information Filtering is the process of supplying documents to a user based on a long term information need: in this application the Query is persistent (though it may be modified). Documents chosen by the user as being relevant are examined and a set of terms is chosen using a statistical technique. In [4] it was stated that an alternative to some ranking methods described, would be to "evaluate every possible combination of terms on a training set and used some performance evaluation measure to determine which combination is best". Various heuristics have been implemented for Term Selection in order to examine some of this search space. Given that these algorithms can take many hours or days we have applied parallel techniques to them in order to speed up processing. Since a Term can be evaluated independently we can split up the Term Set amongst processes and each node can apply the chosen Term Selection algorithm in parallel to its given sub-set of terms in the training set. Terms are scattered to Slave nodes by the Master node (MPI_Scatter is used). The Index is replicated so any node can apply the algorithms on any term. The Term Selection algorithms are applied iteratively until one of a number of stopping criteria is reached. After each iteration MPI_Gather is used to obtain the best term for that iteration, and nodes are notified of the choice using MPI_Bcast. Performance data is gathered using MPI_Gather when the Term Selection algorithm is complete.

7. Work in Progress

We discuss various aspects of our work with MPI including ease of portability, programming with MPI with its advantages/disadvantages and performance.

7.1 Ease of Portability

Various implementations of MPI have been used by PLIERS including CHIMP [5], MPICH [6] and ANU/Fujistu MPI [7]. We did find some differences between implementations such as different type for MPI_comm (which is an *int* in some systems). Various bugs in some of the implementations also caused some problems. Using MPI (with GNU C) has allowed us to port our software to different types of architectures such as a Network of Workstations (NOWS), the Fujistu AP1000 and AP3000 parallel machines and an Alpha Farm. We have completed a port to a Cluster of 16 PC's connected by a supercomputing interconnect running the Windows NT operating system: MPI was invaluable in this process.

7.2 Programming with MPI

Our experience with using different MPI implementations and architectures has been positive. We have found the rank system a useful abstraction particularly when used with collective operations: maintaining code is made easier than other methods such as OCCAM-2 (any topology change would require the re-write of hard wired collective operations). MPI is much more flexible. However this flexibility has it price. The requirement that implementation can vary part of the message passing semantics to cope with lack of buffering space led directly to the termination problem in indexing described above. There is a fairly large set of routines and ideas to learn in order to use MPI to the full, much more so than OCCAM-2. We have not used PVM so cannot compare it with MPI, but we would use MPI rather than OCCAM-2.

7.3 Performance

Results on the architectures currently supported are being generated and examined, but early indications show that there is a performance gain to be had on IR systems by using parallelism. Currently PLIERS can index 14/15 Gigabytes of raw text per hour on 8 Alpha nodes if no position data is needed and term weighting search shows near linear speedup. We will report these and other results in more detail at a later date.

8. Summary

We have found that MPI is a useful method for implementing portable and efficient parallel Information Retrieval systems. In particular we have found that many collective operations are useful for both Query transaction service in IR and Term Selection in Information Filtering. Other message passing facilities have been found to be useful however. Our experience of using different implementations of MPI on different architectures has been positive.

References

1. MacFarlane, A., Robertson, S.E., McCann, J.A.: Parallel Computing in Information Retrieval - An updated Review. Journal of Documentation. Vol. 53 No. 3 (1997) 274-315
2. S. E. Robertson: Overview of the OKAPI projects. Journal of Documentation. Vol. 53 No. 1 (1997) 3-7
3. Jeong, B., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. IEEE Transactions on Parallel and Distributed Systems. Vol. 6 No. 2 (1995) 142-153
4. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M.M. Gatford, M.: Okapi at TREC-3. In: Harman, D.K. (ed): Proceedings of Third Text Retrieval Conference, Gaithersburg, USA (1994) 109-126
5. Alasdair, R., Bruce, A., Mills J.G., Smith, A.G.: CHIMP/MPI User Guide Version 1.2, EPCC-KTP-CHIMP-V2-USER1.2. Edinburgh Parallel Computing Centre (1994)
6. Gropp, W., Lusk, E.: Users guide for MPICH, a portable Implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago (1998)
7. MPI User's Guide. ANU/Fujitsu CAP Research Program, Department of Computer Science, Australian National University (1994)