Adapting to Bandwidth Variations in Wide-Area Data Combination *

M.Ranganathan[†§], Anurag Acharya[‡†], Joel Saltz[†]

[†] Department of Computer Science
University of Maryland, College Park

[‡] Department of Computer Science
University of California, Santa Barbara

[§] National Institute of Standards and Technology, Gaithersburg, MD

mranga@snad.ncsl.nist.gov, acha@cs.ucsb.edu, saltz@cs.umd.edu

Abstract

Efficient data combination over wide-area networks is hard as wide-area networks have large variations in available network bandwidth. In this paper, we examine the utility of changing the location of combination operators as a technique to adapt to variations in wide-area network bandwidth. We try to answer the following questions. First, does relocation of operators provide a significant performance improvement? Second, is on-line relocation useful or does a one-time positioning at start-up time provide most if not all the benefits? If on-line relocation *is* useful, how frequently should it be done and is global knowledge of network performance required or can local knowledge and local relocation of operators sufficient? Fourth, does the effectiveness of operator relocation depend on the ordering of the combination operations. That is, are certain ways of ordering more amenable to adaptation than others? Finally, how do the results change as the number of data sources changes?

1 Introduction

The volume of data available over wide-area networks is growing rapidly. To be able to effectively utilize the large volumes of data that are being placed online, users need to be able to efficiently combine and/or digest data from multiple, geographically-distributed sources. Efficient combination of data from multiple sources is usually achieved by pre-planning based on information about costs of fetching the data and the costs of various operations to be performed on the data. The planning procedure decides: (1) the order in which data from different sources is to be combined, and (2) the location at which each of the combination operations is to be performed. The ordering of the combination operations is usually determined on the basis of intermediate result sizes; whereas the locations for these operations are usually determined by the network bandwidth between the data sources and the locations at which combination operations may be performed as well as the amount of processing to be done for the combination operations. Figure 1 shows an one such plan which corresponds to downloading all data from servers to the client and to perform all processing at the client – which is currently the dominant mode of combining data over wide-area networks.

*This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

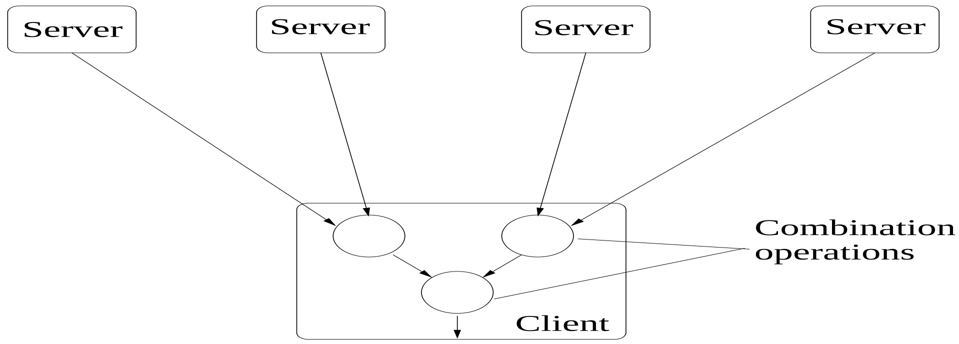
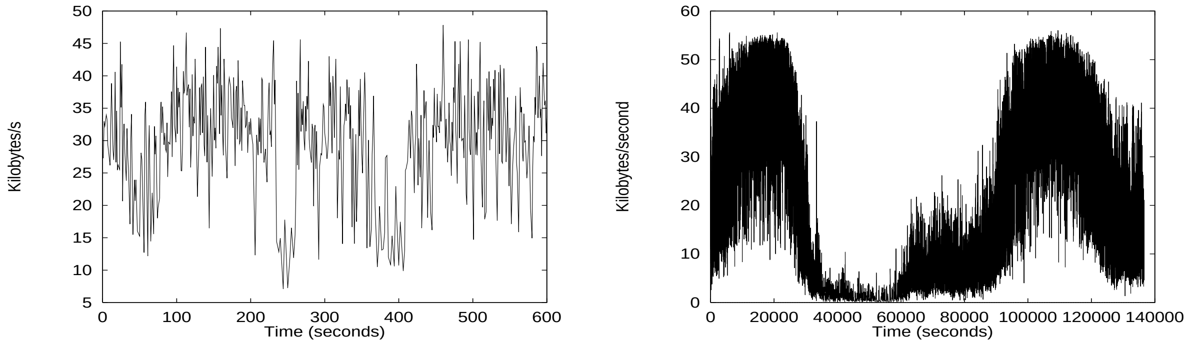


Figure 1: Example plan for data combination. This corresponds to downloading all data from servers to the client and performing all processing at the client.

Efficient data combination over wide-area networks is hard as wide-area networks have large variations in available network bandwidth. These variations are caused by congestion and cross-traffic and their occurrence cannot be predicted reliably. For example, Figure 2 shows the variation of application-level network bandwidth between the University of Wisconsin and UCLA. The first plot shows the bandwidth variation over the first ten minutes of a two-day trace; the second plot shows the bandwidth variation over the entire two-day trace.



(a) Short-term variation

(b) Long-term variation

Figure 2: Variation in application-level network bandwidth. Network bandwidth was computed using the round-trip time for a 16KB message over TCP/IP.

Given the substantial fluctuation in network bandwidth, statically generated plans for data combination are unlikely to be effective. Data combination plans can adapt to bandwidth fluctuations in two ways: (1) change the order of combination operations, and (2) change the location of combination operations. Changing the order of combination operations[1] can adapt to transient bandwidth variations between the data sources and the site of a combination operator by delaying the operator till network performance between its location and the corresponding data sources improves. An example of this is *query-scrambling* [3] which tries to hide initial delay in data arrival as well as bursty data rates by reordering relational join operations. The effectiveness of changing just the order of the operators is, however, inherently limited as it is not able to reposition operators in response to persistent or long-term changes in bandwidth.

[1]We will refer to these as *operators* in the rest of the paper.

In this paper, we examine the utility of changing the location of combination operators as a technique to adapt to variations in wide-area network bandwidth. We try to answer the following questions. First, does relocation of operators provide a significant performance improvement? Second, is on-line relocation useful or does a one-time positioning at start-up time provide most if not all the benefits? If on-line relocation *is* useful, how frequently should it be done and is global knowledge of network performance required or can local knowledge and local relocation of operators sufficient? Fourth, does the effectiveness of operator relocation depend on the ordering of the combination operations. That is, are certain ways of ordering more amenable to adaptation than others? Finally, how do the results change as the number of data source changes?

We have addressed these questions in three different ways. First, we have developed three algorithms that use bandwidth information to adapt data combination plans. The first algorithm (called the *one-shot* algorithm) uses information available at the beginning of computation. The other two algorithms are on-line. The second algorithm is a *global* algorithm in the sense that it runs at only one location in the network, uses global information about network bandwidth and can freely reposition operators as needed whereas the third algorithm is *local* in that it runs on all hosts participating in the computation, uses local bandwidth information and considers only local re-adjustments in the plan.

Second, we have evaluated the performance of these algorithms in the context of a specific task – composition of satellite images from geographically distributed sites. In addition to comparing the end-to-end performance of these algorithms, we have tried to determine how frequently should the adaptation take place for on-line algorithms. Results of such an evaluation depend critically on the network variations. To ensure realistic conditions, we conducted a multi-day study of Internet bandwidth for a large number of host-pairs. This study included US hosts (east coast, west coast, midwest and south), European hosts (in Spain, France and Austria) and one host in Brazil. We used the bandwidth traces acquired in this study to emulate realistic network conditions.

Finally, we have tried to determine if certain ways of ordering combination operations are more amenable to adaptation than others by repeating our experiments with different orders.

We first describe the adaptation algorithms. We then describe the infrastructure needed to implement these algorithms including support for monitoring network bandwidth. Next, we describe our experiments and the study of Internet bandwidth variations used to drive the experiments. We then present the results from our experiments and a brief discussion of these results. Finally, we present our conclusions.

2 Placement algorithms

We first present the assumptions and the common features of all three algorithms. We describe the individual algorithms in subsequent subsections.

We make three assumptions about the servers that provide the data being processed: (1) servers can host computation – this may be on the server itself or on a companion machine located close to it; (2) servers have a single network interface – that is, they can send or receive at most one message at a time; and (3) data is not replicated. The first assumption is critical; relocation of operators is impossible without it. This capability is currently provided by a small number of servers that support server-side Java programs (eg, the Java Web Server, Jetty, Jack, Apache). The remaining assumptions can be relaxed – the algorithms presented in this paper can be easily adapted to to work without them. Both these assumptions, however, hold for most server sites currently online.

In addition, we make two assumptions about the application. First, that communication costs dominate the total execution time. Given the relatively low bandwidth provided by wide-area networks and the continuous improvement in processor speed, this assumption holds for a wide variety of applications. In particular, it holds for the image composition application that is used for the evaluation. Second, that the data being processed can be partitioned and individual partitions can be processed separately. This assumption holds for several application classes – eg, composition/comparison of a sequence of images where each image is a separate partition, hashed relational join where each hash bucket is a separate partition, merging sorted results from multiple search engines where a subsequence of sorted items from a search-engine is a separate partition.

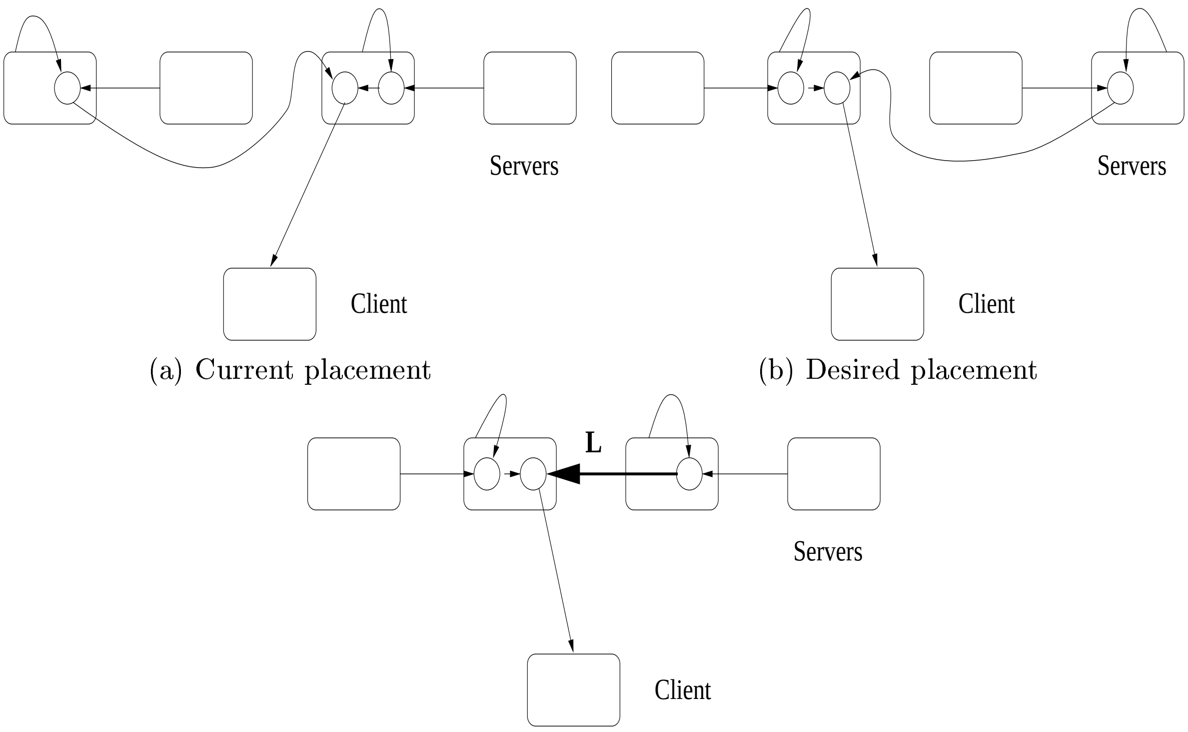
The input to the placement algorithms consists of the order of combination operations (represented as a data-flow tree) and information about network bandwidth (represented as a sparse matrix). The placement algorithms try to minimize the end-to-end execution time of the computation by suitably assigning the internal nodes of the tree (which correspond to combination operators) to the hosts participating in the computation. The execution time is governed by the length of the critical path of the data-flow tree. Critical path is defined as the length of the longest path from a server to the final destination (the client). All three algorithms attempt to iteratively reduce the critical path by repositioning operators on the critical path. Note that the problem of finding an optimal assignment of operators to hosts is NP-complete; the algorithms we present are heuristics.

The *one-shot* algorithm is run once at the beginning of the computation. The on-line algorithms, *global* and *local*, reposition operators during the computation and must satisfy three additional requirements.

Light-move requirement: relocation of operators must be done only when the size of their state is small; otherwise, the cost of moving the operators may outweigh the benefits. To meet this requirement, the computation is structured as a demand-driven data-flow tree with the servers as the leaves, combination operators as internal nodes and the client as the root. Each node in the tree holds its output (original data for the servers, processed data for combination operators) until its consumer requests it. An operator requests data from its producers only after it has dispatched its output to its consumer. Relocation of an operator can occur *after* it has dispatched its output and *before* it requests new data.

Concurrency requirement: these algorithms must run concurrently with the actual computation; stopping the computation to run the placement algorithm can cause large delays. This can be achieved in two ways, either by running the algorithm on a lightly loaded node or by interleaving the execution of the placement algorithm with the actual computation.

Coordination requirement: relocation of operators must be done in a coordinated fashion; uncoordinated moves can reduce the effectiveness of the placement by causing transfers over links with poor bandwidth. For example, suppose that the current placement is as in Figure 3(a) and the desired placement is as in Figure 3(b). As shown, the change-over requires relocation of all three operators. If the change-over is not coordinated, it is possible that in an intermediate stage, two of the operators have been relocated but the third has not (as in Figure 3(c)). This could lead to data transfer over a link that exists neither in the original placement nor in the desired placement (presumably because its bandwidth was poor!). An example of such a link is shown in Figure 3(c) by the thick line marked **L**.



(a) Current placement (b) Desired placement

(c) Undesirable intermediate state

Figure 3: Example of an undesirable intermediate placement that occurs due to uncoordinated relocation of operators.

2.1 One-shot algorithm

Initialization: all operators are placed at the client (eg Figure 1).

Iterative step: First, compute the critical path using a branch and bound algorithm. Let N be the current placement, K the critical path in this placement and C the cost of the placement (that is, cost of the critical path); let N' be the cheapest placement yet found and C' the cost of this placement.

> $C' \leftarrow C$; $N \leftarrow$ `current placement`
> **foreach** `operator` **in** K
>   `consider all alternative locations for operator`
>   `let` C_{min} `be the cost of the cheapest alternative placement`
>   **if** $(C_{min} \leq C')$
> $C' \leftarrow C_{min}$; $N' \leftarrow$ `cheapest placement`
> **end**
> **if** $(C' \leq C)$ $N \leftarrow N'$; $C \leftarrow C'$

In the worst case, this algorithm requires bandwidth to be measured for all links. In practice, however, due to the branch and bound nature of the algorithm only a subset of the links need to be measured.

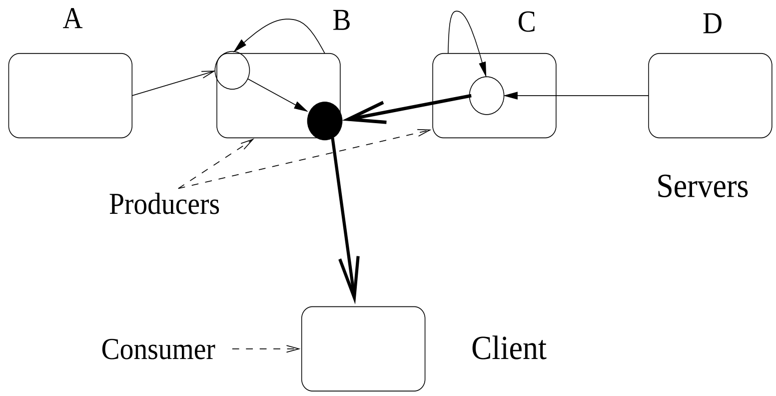


Figure 4: Illustration for the local algorithm. The operator in question is marked by the filled circle; the local critical path around it is marked by lines.

2.2 Global algorithm

This algorithm uses the one-shot algorithm as a procedure to compute new placements; the only modification is in the initialization step where the *current placement* is used as the initial placement instead of placing all the operators at the client. The client computes new placements and coordinates the change-over from an existing placement to a new one. The placement algorithm is triggered periodically and is run concurrently with the actual computation.

To meet the *coordination* requirement, this algorithm uses a barrier-based scheme. Each host maintains an iteration number which is incremented when a new partition of the dataset is served. The client initiates the change-over from a current placement to a new placement by propagating the new placement along with its next demand for data. This placement is propagated up the data-flow tree. When a placement reaches a server for the first time, it sends a message to the client containing its current iteration number and suspends its processing. Once the client receives replies from all servers participating in the computation, it computes the maximum iteration number and broadcasts it to all servers as the iteration at which the change-over should happen. Each of the servers resumes processing once it receives this message. When it reaches the iteration specified in the message, it switches atomically from the old placement to the new placement.

This scheme has the potential disadvantage that barriers might take a long time; for example, if a barrier message is enqueued behind a large (data-transfer) message. To get around this problem, barrier messages are assigned a higher priority than other messages. If multiple messages are enqueued, barrier messages get priority.

2.3 Local algorithm

The local algorithm uses the one-shot algorithm to compute a good initial placement. It then periodically adjusts the location of each operator using a distributed placement algorithm that relies only on local information – the relocation decision for each operator is made based on the bandwidth to the nodes that provide its inputs and the node that consumes its output. For example, in Figure 4, decisions for the operator marked by the filled circle would be based on the bandwidths between B and C and B and the client.

Since no host has global information about the network, it is no longer possible to compute a critical path at a single host. Instead, the critical path is computed in a distributed fashion – each operator determines for itself if it is on the critical path.

The local algorithm has two parts: the first part is used to determine if an operator is on the critical path; if it is, the second part is used to compute a new location. The information required for these algorithms is piggybacked onto the demand messages used for the actual computation.

This algorithm divides time into *epochs* and associates an epoch-timer with each operator. It uses epochs to help determine if an operator is on the critical path and as well to coordinate the relocation of the operators. Each operator decides if it is on the critical path as follows: (1) each time it asks for data from its producers, it keeps track which one delivered the data later – this information is propagated to the producers on the next request for data; (2) at the end of an epoch, an operator counts the number of times it was marked "later"; (3) an operator decides that it is on the critical path iff it was marked the "later" producer more than half the times it sent data during the epoch *and* its consumer was also on the critical path. To ground this recursion, root of the operator tree is always on the critical path (by definition).

At the end of an epoch, an operator currently on the critical path checks if it is possible to improve the *local critical path* around it. The local critical path for an operator is defined as the longest path from either of its producers to its consumer (In Figure 4, a local critical path for the operator marked by the filled circle is shown by bold lines). It considers the locations of the two producers, location of the the consumer and the current location as alternative sites for the operator in question and picks the location that minimizes the local critical path. To fulfill the *coordination* requirement, the epochs for operators on different levels are staggered – e.g. in a combination tree of depth 3, operators on level 0 use epochs 0,3,6,... (3n), operators on level 1 use epoch 1,4,7,.. (3n+1), and operators on level 2 use epochs 2,5,8,... (3n+2). The decision-making, thus, passes up the data-flow tree as a wavefront.

To keep track of the locations of operators, we use a timestamp vector. All participating hosts maintain two vectors – a timestamp vector and a location vector. Each vector has one entry for each operator. When an operator is repositioned, the original site updates the corresponding entry in the location vector and increments the corresponding entry in the timestamp vector. The new information is propagated to peers of the original site by piggybacking it on outgoing messages. When such a message is received, the timestamp vector associated with the message is compared with the current timestamp vector at the receiver. If the incoming timestamp vector dominates[2] the timestamp vector at the receiver, both the vectors at the receiver are overwritten by the incoming vectors and the propagation continues.

The local algorithm does not require a central coordinator. However, it has other limitations. First, local re-positioning of the operator may not result in an overall improvement. Second, the relocation alternatives for an operator are limited. Finally, computation of the placement is interleaved with the actual computation instead of being concurrently.

3 Infra-structural requirements

There are two main infra-structural requirements for implementing these algorithms: (1) the placement algorithm should able to specify the location of combination operations and to move operators during computation; and (2) the placement algorithm should be able to request bandwidth information for any pair of participating hosts.

The mobility support can be provided by mobile object systems like Sumatra [1, 14], Aglets [11], Mole [17] or Telescript [18]. For frequently used servers, operator mobility can also be implemented

[2] A timestamp vector dominates another iff every entry in the first vector is greater than or equal to the corresponding entry in the second and there exists at least one entry in the first vector that is strictly greater than the corresponding entry in the second vector.

by installing all the code at all servers and using control messages to transfer operators between hosts.

The monitoring support can be provided by user-level distributed network monitoring systems like Komodo [13] and the Network Weather Service [19]. Since the placement algorithms run periodically, only on-demand monitoring is needed. Applications that run the placement algorithm infrequently can even do their own monitoring without significantly impacting the results.

4 Experiments

We conducted our experiments in the context of a specific task – composition of satellite images from geographically distributed sites. Composition is done pair-wise; the images being composed are compared pixel-by-pixel; pixels for the resulting image are generated by selecting one of the two corresponding pixels. The two images being composed need not be of the same size; if the images are of different sizes, the smaller image is expanded to the size of the larger image. The resulting image is the same size as the larger image. Each site delivers a sequence of 180 images. Corresponding images from all participating servers are composed and a sequence of 180 images is delivered to the client. The composition operations are arranged as a complete binary tree. This application is based on the AVHRR Pathfinder program used by NASA Goddard to process data from the NOAA satellites [2].

We implemented a detailed discrete event simulation of the system using CSIM [12]. The simulation models the complete protocol and includes the effects of end-point congestion, message buffering and message startup cost, retrieval of images from disk, image composition and transmission delays. The simulation also models high-priority messages, such as barrier messages, which are preferentially processed. In our experiments, the disk bandwidth was set to 3MB/s, the composition operation was assumed to take 7 microseconds per pixel and the message startup cost was set to 50 milliseconds. The message transmission time was computed using this startup cost and actual Internet bandwidth traces. The bandwidth traces were acquired by a multi-day study of Internet bandwidth for a large number of host-pairs. This study included US hosts (east coast, west coast, midwest and south), European hosts (in Spain, France and Austria) and one host in Brazil. The traces were generated by repeated round-trip transfers of 16KB messages over two-day periods. For the experiments described in this paper, we extracted trace segments starting at noon (all experiments were run as if they started at noon).

In addition to modeling the communication and the computation, the simulation also models the network monitoring. It models an on-demand monitoring scheme with the following features: (1) if node A sends node B a message of size greater than S_{thres} both node A and node B know the bandwidth between A and B (passive monitoring); (2) each node maintains a bandwidth measurement cache; entries are timed out after T_{thres} seconds; (3) when a message is sent between two nodes, the most recent bandwidth values (those that fit within 1KB) are piggybacked onto the message. In our experiments, S_{thres} was set to 16KB. For the main set of experiments, T_{thres} was set to 40 sec. This value was chosen based on our analysis of the bandwidth traces which indicated that the expected time between significant changes in the bandwidth ($\geq 10\%$)was about 2 minutes; we picked 40 sec as a conservative value (a little less than half the expected period). We varied this value in our experiments to determine how frequently should the placement algorithm be run.

To determine a good image size, we downloaded over 1000 images from 15 web sites that provide hurricane images. We found that the image sizes fit a normal distribution with a mean close to 128KB and a variance of 25%. For our experiments, we generated a sequence of 180 images for each server; the image sizes being selected from a normal distribution with a mean of 128KB and a variance of 25%.

To answer the first three questions raised in the introduction (does relocation help; is on-line relocation needed; and if so is global network knowledge important) we evaluated four placement algorithms on 300 different network configurations each with 8 servers and one client. The placement algorithms included the three algorithms described above and a trivial algorithm (called *download-all*) which places all operators at the client. As mentioned earlier, this is currently the dominant mode of data combination over wide-area networks. For these experiments, the online placement algorithms (global and local) were run once every 10 minutes. We generated the network configurations by different assignments of the Internet bandwidth traces to the links in a complete graph of nine nodes (eight servers and one client). The assignments were generated using a uniform random number generator. We chose to run our experiments on 300 network configurations after preliminary experiments showed that using more configurations (up to 600) did not cause a significant change in the results. In our experiments, the *download-all* placement algorithm is used as the base-case; performance of the other algorithms is presented relative to the performance of this algorithm.

To answer the fourth question (are certain ways of ordering more amenable to adaptation than others), we reran our experiments using a left-deep tree to order the composition operations (see Figure 5 for an illustration). A left-deep combination tree is linear whereas a complete binary tree is maximally bushy. Left-deep trees are often used for database query plans.

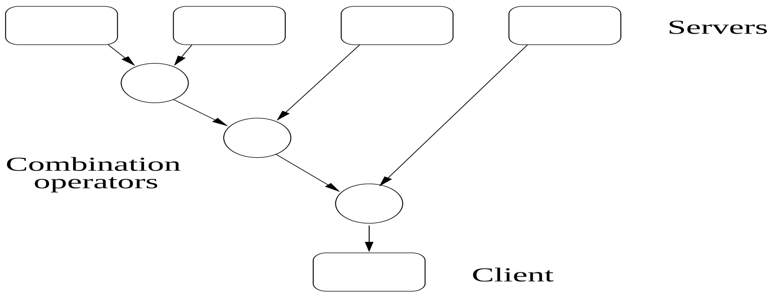
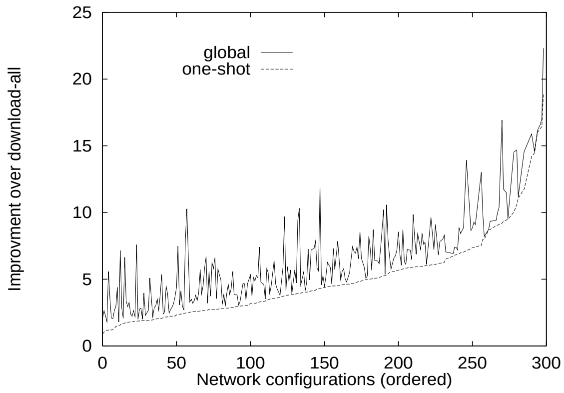


Figure 5: Example of a left-deep combination tree.

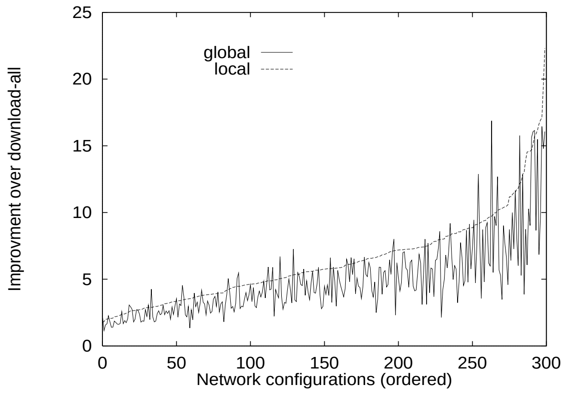
To answer the fifth question (how do the results change as the number of data sources changes), we varied the number of servers from four to thirty-two. Finally, to determine how frequently should the placement algorithm be run, we studied the performance for time periods ranging from one minute to one hour.

5 Results and Discussion

Figure 6 presents the performance of the proposed placement algorithms for the 300 network configurations. Both graphs use the download-all strategy as the base-case – the performance of the other algorithms is measured as the speedup they achieve over the download-all strategy. In each graph, the network configurations have been sorted by the performance of one of the algorithms being compared – to make it easier to compare the results. There are several points to note. First, all relocation algorithms significantly outperform the download-all strategy. Second, while one-shot placement achieves large gains, on-line relocation provides a consistent and substantial additional improvement - in particular, the global algorithm achieves a median improvement of 40% over and above the speedup achieved by the one-shot algorithm. Third, except in a few cases, the global algorithm performs better than the local one – the median ratio in this case is about 1.25. A more direct sense of the impact of relocation algorithms can be seen from reduction in the average inter-arrival time for processed images at the client from 101.2s for *download-all* to 24.6s for *one-shot*, 22s for *local* and 17.1s for *global*.



(a) Global vs one-shot placement (sorted by performance of one-shot)



(b) Global vs local placement (sorted by performance of global)

Figure 6: Performance of operator relocation algorithms for 300 network configurations. The performance of an algorithm on a particular configuration is measured as the speedup it achieves over the download-all strategy. In each graph, the network configurations have been ordered to make it easier to compare them. Both graphs are on the same scale.

To understand why the local algorithm is unable to match the performance of the global algorithm, we studied the relocation traces we obtained from the simulations. We found two reasons. First, each operator moves in a locally optimal greedy fashion regardless of whether the move actually results in an overall reduction in the critical path. Second, the local algorithm is unable to react quickly and effectively to changes in network bandwidth. This happens because it only makes local adjustments and often needs several steps to converge to a desirable state. In the intervening period, it is often in a less efficient state. Furthermore, in many cases, by the time it is able to achieve the desirable state, the network changes again. The first problem is inherent to the distributed nature of the algorithm; to mitigate the effect of slow convergence, we considered a strategy that considers up to k additional locations for possible placement of the operator being repositioned. These k locations are in addition to locations already under consideration and are chosen randomly (uniform distribution) from the remaining hosts. Note that adding additional locations increases the cost of running the algorithm as additional links have to be monitored. We ran experiments allowing between one to six additional locations. These experiments were run on all 300 configurations. Figure 7 shows the impact of this modification on performance of the local algorithm. We see that there was no significant difference in performance

To explore if this conclusion (the global algorithm performs better than the local algorithm) would hold if the number of servers was changed, we varied the number of servers from four to thirty-two. These experiments were run on all 300 configurations and were run for all four placement algorithms – download-all, one-shot, global and local. The local algorithm did not consider any additional locations. In all cases, the workload consisted of 180 images/server and 180 images delivered to the client. Figure 8 presents the results. We were surprised to note that the global

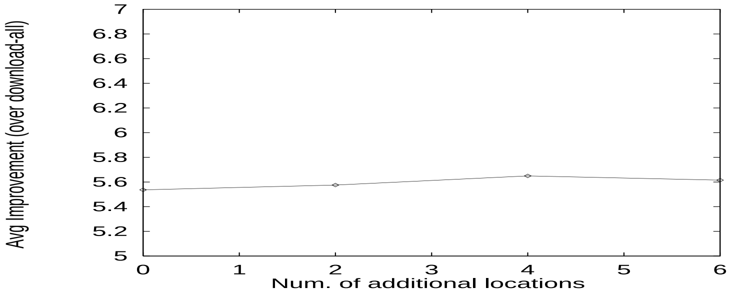


Figure 7: Impact of considering additional (randomly selected) locations on the performance of the local relocation algorithm. Each point is an average of the speedup achieved by the algorithm over all 300 configurations.

algorithm scaled better than both the one-shot algorithm and the local algorithm. An analysis of the simulation traces showed that the convergence problem that limits the performance of the local algorithm for 8 servers impacts its performance even more for larger configurations as achieving desirable placements takes even more adjustment steps.

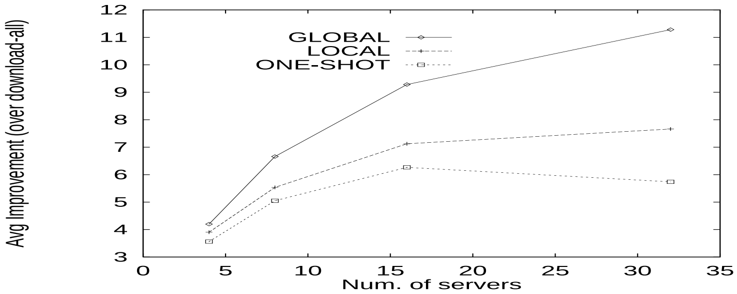


Figure 8: Impact of variation in the number of servers on the performance of relocation algorithms. Each point is an average of the speedup achieved by one algorithm over all 300 configurations.

To study the impact of repositioning period on the performance of relocation algorithms, we ran the global algorithm with five relocation periods between two minutes and an hour. These experiments were run on all 300 configurations. Figure 9 presents the results. We see that a 5-10 minute relocation period provides the best performance.

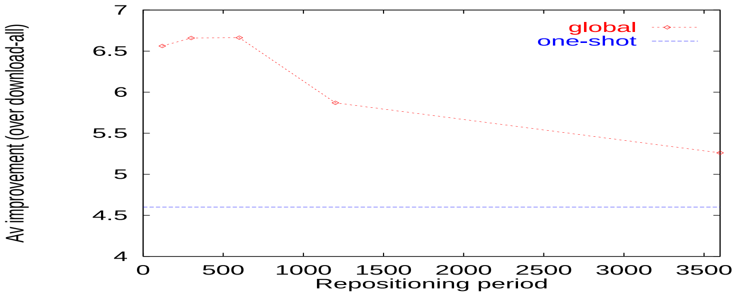
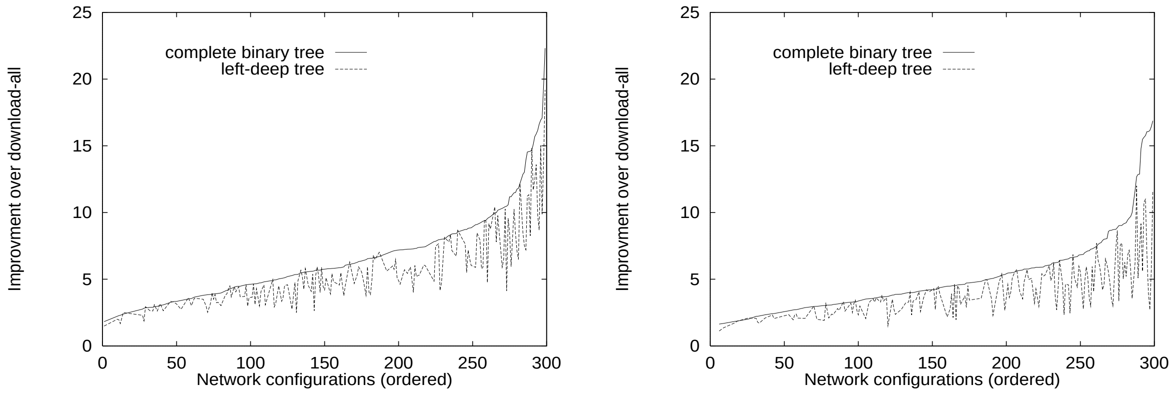


Figure 9: Impact of variation in relocation frequency. Each point is an average of the speedup achieved over all 300 configurations.

To study the impact of combination order, we reran experiments with global, local and download-all algorithms using a left-deep tree to order the composition operations instead of a complete binary tree. These experiments were run on all 300 network configurations. Figure 10 presents the results. We note that an order based on a complete binary tree allowed either relocation algorithm to achieve a better performance than an order based on a left-deep tree.



(a) global placement algorithm (b) local placement algorithm

Figure 10: Impact of combination order. Each point is an average of the speedup achieved over all 300 configurations. The network configurations for both graphs have been sorted by by the performance of the complete binary tree version. Both graphs are on the same scale.

6 Related Work

There are two major areas of related research – dynamic query optimization and mapping task graphs onto networks of workstations. Dynamic query optimization has been proposed for dealing with uncertainties in the size of intermediate results and in the availability of memory and other resources. Bodorik et al [5, 6] apply critical path analysis and on-line correction of query plans to deal with mispredictions in the size of intermediate results. Their primary technique is to reorder the query plan.

Franklin et al. [9] have proposed dynamic query optimization for dealing with uncertainty in the amount and configuration of memory available. The primary technique, similar to the *one-shot* algorithm proposed in this paper, determines suitable locations for various operators at query startup time. The amount of memory available is assumed to not change for the duration of the query.

Amsaleg and Franklin [3] have proposed *query scrambling* to deal with unexpected delays in data arrival in a wide-area environment. The primary technique, once again, is to reorder the query plan. They did not consider relocation of operators.

Various scheduling heuristics for task graphs have been proposed [4, 16, 20, 21]. Comparisons of some of these schemes appear in [8, 10]. While the task graph scheduling problem is similar to the problem considered in this paper, it differs in several important ways. First, task heuristics assume that the network speed is constant and is the same for all links. Second, they assume that the amount of data to be communicated and the time taken to process this data are known a-priori. Finally, they assume that computation time dominates communication time. These assumptions do not hold in a wide-area environment. Nevertheless, the algorithms proposed in this paper bear a strong similarity to task graphing scheduling algorithms – in particular, the idea of trying to repeatedly shorten the critical path.

Our scheme of partitioning the dataset and processing a set of partitions at a time is similar in spirit to pipelined hash-joins which are used in parallel and distributed databases [7, 15]. The advantage of pipelining in both these scenarios is that intermediate results do not have to be stored. For our algorithms, pipelining provides an additional advantage in that it allows individual stages in the pipeline to be relocated after each data partition. To our knowledge, adaptive pipelined joins have not been considered.

7 Conclusions

There are five main conclusions of our study. First, all relocation algorithms significantly outperform the currently used strategy of downloading all data to the client and processing it there. Second, while one-shot placement achieves large gains, on-line relocation provides a consistent and substantial additional improvement. Third the global algorithm performs better than the local one – at least up to 32 data sources. Fourth, a 5-10 minute relocation period provides the best performance. Finally, using a complete binary tree to order the combination allows either relocation algorithm to achieve a better performance than an order based on a left-deep tree.

The infra-structural support required for achieving this is not demanding. The primary requirements are (1) ability to specify the location of combination operations and to move operators during computation; and (2) ability to get bandwidth information for any pair of participating hosts. The mobility support can be provided by one of many mobile object systems currently available. For frequently used servers, operator mobility can also be implemented by installing all the code at all servers and using control messages to transfer operators between hosts. The monitoring support can be provided by user-level distributed network monitoring systems; given the 5-10 minute relocation period, applications can do their own monitoring.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. *Sumatra: A Language for Resource-Aware Mobile Programs*, chapter ˙Springer Verlag Lecture Notes in Computer Science, 1997. J. Vitek and C. Tschudin (eds).

[2] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the 4th Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, May 1996.

[3] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, December 1996.

[4] F. Anger, J. Hwang, and Y. Chow. Scheduling with sufficiently loosely-coupled processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[5] P. Bodorik, J. Riordan, and J. Pyra. Deciding to correct distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):12–21, June 1992.

[6] P. Bodorik and J. Riordon. A threshold mechanism for distributed query processing. In *ACM Computer Science Conference*, pages 616–625, Atlanta, Georgia, Feb. 1988.

[7] M.-S. Chen, M. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the exeucution of pipelined hash joins. In *18th International Conference on Very Large Databases*, Vancouver, Aug. 1992.

[8] H. El-Rewini and T. Lewis. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[9] M. J. Franklin, B. Jonsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *ACM SIGMOD 96*, Montreal, Canada, June 1996.

[10] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–91, Dec 1992.

[11] D. Lange and M. Oshima. *Programming Mobile Agents in Java.* In progress, 1996.

[12] Mesquite Software. CSIM Simulation Software. `http://www.mesquite.com/`.

[13] M. Ranganathan, A. Acharya, and J. Saltz. Distributed resource monitors for mobile objects. In *International Workshop on Operating System Support for Object Oriented Systems*, 1996.

[14] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 91–104, Jan 1997.

[15] J. Richardson, J. Lu, and K. Mikkilineni. Design and evaluation of parallel pipelined join algorithms. In *ACM SIGMOD*, pages 399–409, May 1987.

[16] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, 1989.

[17] M. Straβer, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In *Proceedings of the ECOOP'96 workshop on Mobile Object Systems*, 1996.

[18] J. White. Telescript Technology: Mobile Agents, 1996. *http://www.genmagic.com/Telescript-/Whitepapers.*

[19] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. Technical Report TR-CS96-494, University of California at San Deigo, Oct 1996.

[20] M. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–43, 1990.

[21] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–67, 1994.