Minimum Cost Path for a Shared Nothing Architecture

Maytham Safar
Computer Engineering Department, Kuwait University, Kuwait

Abstract: *Computing the minimum cost path is a key requirement in Intelligent Transportation Systems (ITS) and in some Geographical Information Systems (GIS) applications. The major characteristics of these systems are the facts that the underlying transportation graph is large in size and the computation is under time constraint. Due to the insufficiency of the classic algorithms under these settings, recent studies have focused on speeding the computation by employing alternative techniques such as heuristics, precomputation and parallelization. In this study, we investigate solutions assuming a shared nothing architecture (i. e., Teradata multimedia database system) as a way of speeding up the computation further. We build our algorithms on a recently developed graph model, Hierarchical mulTigraph (HiTi), and describe both concurrent and parallel versions of the algorithms. The concurrent algorithm allows simultaneous exploration of the search space by utilizing dynamically created agents across multiple disk nodes, which is efficiently supported by the Teradata multimedia database system architecture. The parallel algorithm breaks the problem into a set of smaller subproblems by exploiting a set of intermediate nodes that the shortest path passes through. We also investigate the impact of replicating subgraphs in the performance of our algorithms. We evaluated our algorithms via a simulation study and demonstrated that our concurrent and parallel algorithms show almost a linear speedup as the number of disk/CPU nodes is increased. Concurrent algorithm exhibits better sizeup, and scaleup results than the parallel algorithm.*

Keywords: *Shortest path, GIS system, intelligent transportation systems, shared nothing architecture, teradata multimedia database.*

Received May 24, 2004; accepted July 31, 2004

1. Introduction

In this section, we first motivate our work based on two real-world applications. Subsequently, we compare our work with related studies. Finally, our major contributions are presented.

1.1. Motivation

Recently, new fields such as Intelligent Transportation Systems (ITS) and Geographical Information Systems (GIS) have gained much attention from different academic disciplines. An ITS can be considered as the integration of advanced technologies in areas such as communication, information and navigation in order to achieve an economically improved and safer transportation systems. A GIS can be considered as the integration of different types/formats of data, which are related through geographical information. One of the interesting queries that can utilize this rich set of information is to find a best path between a source and a destination. One particular problem that is identified, as a key requirement for ITS and some GIS applications is the efficient processing of minimum cost path queries. The underlying transportation graphs are typically large and the queries need to be processed very quickly due to the real time nature of the application. Due to these characteristics, the classic algorithms (such as Dijkstra) have been found to be insufficient. In order to remedy the situation alternative methods such as heuristics, precomputation and parallelization have been studied.

In this study, we investigate solutions assuming a shared nothing architecture as a way of speeding up the computation further. In particular, our target hardware and software platform is a Teradata Multimedia Database System (M-DBS) [5, 6], which is built on NCR WorldMark multi-disk multi-processor system. Teradata M-DBS software supports the concept of agents as pieces of code that can migrate and execute on the physical (disk/CPU) node(s) containing the object(s) referenced by the agent. It also provides users with the capability of implementing their User Defined Functions (UDFs) in order to extend the functionality of the M-DBS. We define each of our minimum cost path algorithms as one UDF on Teradata M-DBS. Each UDF forks many agents on multiple nodes in order to speedup computation by simultaneous execution of agents.

We have developed our algorithms on a famous graph model, Hierarchical mulTigraph (HiTi) [16, 17], because of several reasons. First, it is the most recent work on single pair shortest path problem, which is developed for large road graphs. Second, it performs better than the traditional A^* algorithm by exploiting

the hierarchical nature of road navigation. Third, it is based on partitioning the graph into small subgraphs, which is very suitable for a shared nothing system. This partitioning significantly reduces the search space for computing the minimum cost path over a very large topographical road map [17]. In this paper we discuss several partitioning techniques that can form the underlying structure of these HiTi graphs, which are interesting in their own nature.

1.2. Related Work

There have been extensive studies on the shortest path problem both in theory and in practice. For all-pairs shortest path problems, the classic algorithms are Floyd-Warshall [7, 20] for general graphs and Johnson's algorithm for sparse graphs [15]. For the single-source shortest path problem the classic ones are Bellman-Ford [3, 8] and Dijkstra [4].

Parallel and distributed algorithms are developed to take advantage of the technological developments in computer architecture. For example in a series of studies [9, 10, 18] the graph is fragmented to recursively decompose the problem into smaller tasks and assign each task to a different processor. However their disconnection sets are precomputed beforehand, and they don't depend on the *source* and *destination* nodes of the problem at hand. Moreover their methods mainly targets acyclic fragments formed after partitioning. This restriction makes it very unsuitable for transportation graphs, which have almost grid-like and highly cyclic structure. Furthermore, we are not aware of any study that investigates concurrent/parallel minimum cost path algorithms assuming a shared nothing architecture with agent-based methodology.

Recent works aim at finding efficient solutions to single pair shortest path problem. This problem is in fact the most important problem for Navigation system. Shekar in [19] developed a hierarchical A^* algorithm. Their motivation was that the classic algorithms (like Dijkstra) process all the nodes in the graph to reach to a solution that is more than unnecessary and time consuming in practice. Therefore they suggest a heuristic method that searches only a necessary subset of the nodes. Their method takes advantage of the specific nature of transportation networks, *i. e.*, the existence of high-speed roads like freeways and highways. One major drawback of this approach is that it may not find the optimal solution. It is also not suitable for a *direction-based* centralized architecture since the computation of the entire path is required to find the next optimal direction.

A recent research done in [12, 14] followed a different approach: path encoding. They tried to develop algorithms that respond to the path queries as fast as possible since the number of queries received at any time can be very large especially in busy hours of traffic. In this method each node stores the tuple *<destination, successor, weight>* for all (or subset of) the destination nodes in the graph. A hierarchical data structure by fragmenting the graph is studied in [13] and different fragmentation strategies are evaluated in [14].

Agrawal in [1, 2] used a branch and bound search algorithm to reduce the total number of nodes visited [1]. Their overall approach is to precompute some partial information and then use it at run time to prune the search space. The essential part of their construction is the domain partitioning since it determines the goodness of the bounding procedure. The ideal domain partitioning requires choosing a specified number of domain centers such that the average distance to (from) a node from (to) its domain center is minimized. However, since this is an NP-hard problem they use heuristics that may reduce the search efficiency. As was observed in [17], another drawback of this approach is that it searches all the unnecessary intermediate edges before reaching to the target, therefore it ignores the hierarchical nature of the navigation systems. A similar work was done in [10] showed that Transitive closure query in a distributed environment could be broken into several subqueries that can be processed in parallel. In [14] a Topological clustering technique SPC was proposed to exploit the unique GIS road map characteristics to achieve I/O optimization in path query processing but it had the problem of space and computation overhead for creating link tables. An important recent work in [16, 17] explains a HiTi (Hierarchical mulTigraph) model to speed up the single pair shortest path problem on a topological road map [17]. They successfully avoid the searching of unnecessary intermediate edges by the help of these graphs. We will explain HiTi graphs briefly in Section "Concurrent/Parallel HiTi" since our algorithms are developed for these graphs.

1.3. Major Contributions

In this study, we investigate solutions assuming a shared nothing architecture (i. e., Teradata multimedia database system) to quickly compute the minimum cost paths in large transportation graphs. Our algorithms are based on a recently developed graph model, Hierarchical mulTigraph (HiTi) [17], and describe both concurrent and parallel versions of the algorithms. Our concurrent algorithms allow simultaneous exploration of the search space by utilizing dynamically created agents across multiple disk nodes, which is efficiently supported by the Teradata multimedia database system architecture. Our parallel algorithm exploits the property of HiTi graphs that allows the computation to be recursively broken into smaller subproblems. We also investigate the impact of replicating subgraphs in the performance of our algorithms. We evaluated both of our algorithms via a simulation study and demonstrated an almost

linear speedup for the algorithms as we increase the number of disk/CPU nodes.

The rest of this paper is organized as follows. In Section "Data Structures", we describe grid-based partitioning technique. We also investigate the role of replication in Section "Replication". In Section "Concurrent/Parallel HiTi", we first describe the agent-based methodology, which is the underlying concept of all of our algorithms. We then briefly describe the HiTi graph model. Finally we describe our concurrent and parallel versions of the shortest path algorithm developed for HiTi graphs. Section "Performance Evaluation" evaluates our algorithms using a simulation model. Our conclusion and future research directions are contained in Section "Conclusion".

2. Data Structures

Most of the previous studies on computing shortest paths either ignored or used heuristic methods to partition the graph. However partitioning the graph is one of the most important part of any algorithms since it determines the effectiveness of search procedures and storage allocations [2, 11, 13]. Therefore, it is essential to provide a partitioning technique that is specifically tailored for transportation graphs and have proven complexity bounds which can be fine tuned for the underlying system. In the following sections, we explain grid-based partitioning technique and its adoption to parallel and distributed systems.

2.1. Grid-Based Declustering

Suppose a graph with n nodes and m *edges* which can be covered by a bounding box of width w and height h when embedded in the plane. Such an input is typical especially in GIS applications. A natural way of spatially partitioning the graph is by imposing a regular grid on top of this bounding box. Using an $r \times s$ sized grid results in tiles of sizes wh/rs that can estimate n/rs nodes in each tile. Considering the grid-like structure of transportation graphs (i. e., each node is usually a 4-way intersection), such a measure is a good approximation to the number of nodes in each tile, which is needed to decluster the graph. In our simulations (see Section "Performance Evaluation"), we partitioned the graph by using 8×8 dimensional grid. See Figure 1, Grid based graph.

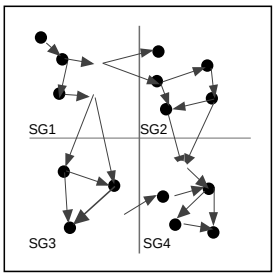

Figure 1. Grid based partitioning of a graph into 4 subgraphs of equal geometric space.

2.2. Replication

Each of the above partitioning method can be combined with replication. To discuss the role of replication, suppose we have p disk-nodes. If the graph is partitioned into exactly p subgraphs, then we obtain a one-to-one assignment of these subgraphs to the disk-nodes. This approach may appear to be the ideal declustering technique. However, in an agent-based system, depending on the nature of the particular algorithm being executed, some disk nodes might have a heavy I/O traffic if too many agents are trying to access them. To avoid such "*hot-spots*" a technique called replication is often used. In this technique, instead of partitioning the graph into p subgraphs, we may, for example, partition it into *fracp2* subgraphs and replicate each subgraph once to fill the remaining *fracp2* empty disk nodes. In general we can have *fracp2* subgraphs each of them with k copies, where $1 \leq k \leq p$.

In our experimental studies, we evaluated the impact of replicating the subgraphs over the disk nodes. We only tried the cases $k = 1, 2$ because of the following reasons.

1. *Space*: The more the replication, the more the space requirement is. We can see this by using the following simple calculation. If we assume that each subgraph has almost the same size then with k replication each subgraph has size nk/p where n is the graph size. Subsequently, the total space requirement is $(nk/p) \times p = nk$. So the total size increases by a factor of the replication number.
2. *Consistency*: Extra effort must be spent to make these multiple copies consistent if the data structures used are dynamic (which is the case in our algorithms). In general the cost is linear in the number of replicated copies.
3. *Load*: The more the replication, the less the difference between the loads on disk nodes. In this case we are facing the situation of over-replicating the subgraphs whose benefit becomes negligible compared to the costs associated to it.

3. Concurrent/Parallel HiTi

In this section we discuss our algorithms that are based on a recently developed graph model, Hierarchical mulTigraph (HiTi) [17], and describe both concurrent and parallel versions of the algorithms. Our concurrent algorithms allow simultaneous exploration of the search space by utilizing dynamically created agents across multiple disk nodes which is efficiently supported by the Teradata multimedia database system architecture. Our parallel algorithm exploits the property of HiTi graphs which allows the computation to be recursively broken into smaller subproblems.

3.1. Agent Based Methodology

We will briefly describe the agent-based methodology. For details see [5]. In a multi-disk/CPU environment that stores large objects (such as multimedia), transferring the data to the tasks is costly. So instead of transferring the data across the nodes, the tasks (agents) are transferred to the disk/CPU node where the data resides.

In these systems the computation is based on the execution of agents. The execution of these agents is triggered by different events. An agent can be cloned or migrated. Agent cloning occurs when the data to be modified resides on other disk/CPU nodes. Agent migration is useful when the agent needs to access more than one object residing in different disk/CPU nodes. This can be done sequentially in a proper order. Agents are programmed by using a scripting language and depending on the operations performed these scripts range from very simple to very complex. They are transferred along with their context and a mailbox that is used for triggering events. In our algorithms, agents communicate by reading/writing information from/to disks.

3.2. HiTi Graphs

We start by briefly explaining the HiTi graph model that was originally designed in [17] and give the definitions and the notations that will be used by the algorithms. Subsequently, we describe the corresponding single pair shortest path algorithm.

3.2.1. Construction, Definitions and Notations

The construction of HiTi graph starts by first partitioning the original graph into disjoint subgraphs called Component Road Maps (CROM). (These are called level 1 subgraphs.) Such a partitioning can be obtained by using the method explained in Section "Data Structures". Then the *boundary* nodes between each neighbor CROMs are identified. A direct connection between two different CROMs, i. e., an edge between the boundary nodes, is called a *between* connection. For each CROM, the cost for each pair of connected boundary nodes is precomputed and if the corresponding path is solely contained within the CROM then this is called a *within* connection. A HiTi graph is then defined as a graph whose nodes are the boundary nodes and the edges are the *between* and *within* edges. This construction can be continued hierarchically by defining a level k CROM that contains a set of level k-1 CROMs for $k \geq 2$. The resulting structure can be viewed as a subgraph tree (ST) whose root is the entire graph. The *ith* subgraph at level j (of ST) is denoted as SG^j_i. A HiTi graph defined over level $k+1$ subgraph tree ST is called a level k HiTi graph and is denoted as $H^k (P^k, A^k)$ where P^k is the set of all the boundary nodes and A^k is the set of all the *within* and *between* edges.

Let X be a set of subgraphs in ST. Then $S_A (x)$ denotes the set of ancestors of X. $S_C (x)$ denotes the set of direct children of X. $S_N (x)$ denotes the set of boundary nodes in the subgraphs of X. $LUB_{SG} (SG^l_i, SG^l_j)$ denotes the least leveled common ancestor subgraph of SG^l_i and SG^l_j.

3.2.2. Shortest Path Algorithm Based on HiTi Graph (SPAH)

The Shortest Path algorithm (SPAH) is a variation of the A^* algorithm and it exploits the hierarchical structure of HiTi graphs. It consists of two phases: *ascending* and *descending*. The ascending phase is the period of traversal from the source node *START* to the boundary nodes $S_N (S^{l-1} (SG^l_j))$ where $l = LUB_{SG} (SG^l_i, SG^l_j)$. The descending phase is the period of traversal from the boundary nodes of $S_N (S^{l-1} (SG^l_j))$ to the destination node *DEST*. The following is the complete algorithm described in [17].

Algorithm Concurrent-SPAH (START, DEST)

begin

Assume we have a level k HiTi graph defined on a level k +1 ST;
Find SG_i^1 SG_j^1
$H^k (P^k, A^k)$
$LUB_{SG} (SG_i^1, SG_j^1)$
$P = k; p \geq 1; p$ - -
$S_N (S_C (S^P_A (SG_j^1)))$
$\lambda = 0$
λ (Agent) > λ ($Node_i$)
λ ($Node_i$) = λ ($Agent_i$)
$Node_i$
$Node_i$
$min (l - 1, r) \leq p \leq r$
$Node_i \xrightarrow{z} Node_j$
$\lambda = \lambda (Node_i) + z$
$Node_j$
$Agent_i$
$Node_i$
$Node_i \xrightarrow{z} Node_j$
$\lambda = \lambda$ (Node) + z
$Node_j$
$Agent_j$

Figure 2. Concurrent-SPAH.

3.3. Concurrent HiTi

Our concurrent algorithm exploits the agent-based methodology Teradata M-DBS that was described briefly in section "Agent based Methodology". The intuition is to simultaneously explore the search space by the help of dynamically created agents that can migrate across multiple disk/CPU nodes (see Figure 3). Since such technique is efficiently supported by the Teradata architecture, we expect significant speedup when the graphs have large size. Figure 2 gives the

complete algorithm that is based on SPAH and modified to take advantage of agent-based environment.

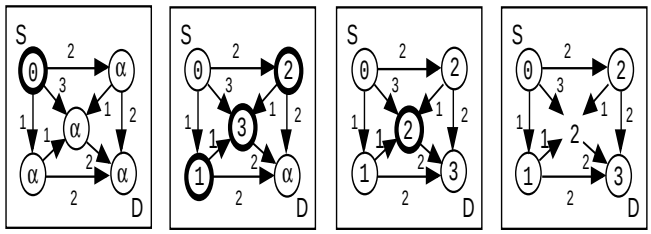

Figure 3. From left to right, the figures show how the concurrent algorithm runs. The nodes with thick black circle are the ones that have active agents running on. S: Source node, and D: Destination node.

3.4. Parallel HiTi

Observe that SPAH consists of two phases: *Ascending* and *descending*. In the *ascending* and *descending* phases the edge traversal is from the node *START* to some marked boundary nodes and from the marked boundary nodes to the node *END*. All the paths those pass through the marked boundary nodes are searched. This suggests computing the shortest distance from *START* to a boundary node and from a boundary node to *DEST*, for all the boundary nodes in parallel. Hence, if an agent is working on a node ($Node_i$), then the agent has to access the information (links connected to the node) that is stored on the same disk node. Subsequently, the agent has to clone it self and then migrate to all the nodes with links to $Node_i$, i. e., solve more than one shortest path problem simultaneously. To avoid cycles, we apply the same techniques described in section "Concurrent HiTi". Figure 4 demonstrates the Parallel-SPAH algorithm.

Algorithm Concurrent-SPAH (START, DEST)
begin

Assume we have a level k HiTi graph defined on a level k +1 ST;
Find SG_i^1 SG_j^1
$H^k(P^k, A^k)$
$LUB_{SG}(SG_i^1, SG_j^1)$
$P = k; p \geq 1; p$ - -
$S_N(S_C(S^P{}_A(SG_j^1)))$

Figure 4. Parallel-SPAH algorithm.

4. Performance Evaluation

In this section we evaluate our proposed techniques by applying a large set of experiments using a simulation system.

4.1. Simulation Setup and Measures

We have implemented the simulations in C++ and ran the performance comparison on a Sun workstation under Solaris to verify the practicality of our propositions. In our first set of experiments, Synthetic grid-graphs were used as data sets. The number of nodes was varied to obtain graphs of different sizes. Most of the experiments were performed with a graph of 1024 nodes, with average out-degree of 4 and a weight of [1, 5]. The function *f (u, DEST)* computes the Manhattan distance $|x_1 - x_2| + |y_1 - y_2|$, where ($x_1$, y_2) and (x_2, y_2) (are the locations of the points on the grid) between the node *u* and *DEST*. For our analysis, we create two dimensional grid graphs *G (V, E)* with 4 adjacent nodes. From *G (V, E)*, we create a level 4 subgraph tree ST. The level 4 subgraphs tree ST consists of 64 levels 1, 16 level2, 4 level 3, and 1 level 4 subgraphs.

In the second set of experiments, we use randomly generated graphs. The distance between two nodes was a random variable over an interval [10, 100]. The edge cost is generated based on a uniform distribution that was normalized to be consistent with the distance between two nodes. Each node has an out-degree of [3, 5] links. [2, 4] of the links were to nodes that have the shortest Euclidean distance from that node and the other links were for other nodes selected randomly. After creating the graph we spatially decluster the graph (Section "Grid-based Declustering") into smaller subgraphs and build the HiTi graph on the top of it. We use level 4 subgraph tree ST to generate level 3 HiTi graph for our analysis. The function *f (u, DEST)* was used to estimate the cost of the shortest path from node *u* to *DEST*. The function *f (u, DEST)* computes the Euclidean distance between the node *u* and *DEST* [17].

We create adjacency lists for each subgraph. Each node has the following information: all links attached to it plus (level, cost, destination). The ST graph is stored on one disk. The boundary (within and between) links are attached to the nodes of the original graph. We conduct single source shortest path search for randomly selected nodes. We compute 5 different shortest paths randomly pairs of source and destination nodes and averaged the results for the 5 runs.

In our experiments we use a measure that counts the Maximum number of links (*MLAC*) accessed among all the agents to reach the Destination (*DEST*). In sequential algorithm it measures the total number of links accessed to reach from Source (*SOURCE*) to Destination (*DEST*). All the parallel and concurrent algorithms links that are accessed simultaneously are considered as one *MLAC* measure.

In our performance comparison we use three performance measures. One is effect on *MLAC* as the size of the graph changes. The other one is the effect on *MLAC* as the number of CPU/disk nodes changes. The third one is the effect on *MLAC* as the limit on agents/disk accesses changes. For the Parallel algorithm, we change the limit on the total number of disk accesses at a time, and for the concurrent algorithm we change the limit on the total number of agents that a CPU could handle at a time. This measure is the same as the limit on the disk accesses for the

parallel algorithm. This is due to the fact that in our concurrent algorithm each agent created needs to access only one piece of data from the disk. We should note that our simulations ignore the typical optimization of real systems such as the memory residency of the data due to caching.

4.2. Results of the Experiments

We now discuss the performance of the ideas presented in the previous sections by conducting simulations. We study the influence of changing the number of graph nodes, the number of disks on *MLAC*. The structure of this section is as follows. In Sections "Speed-Up", "Size-Up", and "Scale-Up" we study the speedup, sizeup, and scaleup characteristics of our algorithms. Then, in Section "Replication", the benefits of replication are made obvious by presenting the simulation results. And in the last Section "Comparison of Parallel, Concurrent and SPAH" a comparison is made between *SPAH*, *concurrent*, and *parallel* algorithms. This section shows the graph size, and the system configuration in which each algorithm performs the best. And in the last Section "Random Graphs" we study the performance of the algorithms on randomly generated graphs.

4.2.1. Speed-Up

For our first set of experiments, we examined the speedup characteristics of the *parallel* and *concurrent* algorithms. We changed the number of disk nodes while keeping the graph size fixed. We did this for graph sizes of 16X16, 24X24 and 32X32 nodes.

Figure 5 and Figure 6 show the results of the speedup experiments. It is clear from the graphs that as we increase the number of disk/CPU nodes MLAC gets smaller. The speedup performance improved with larger graph sizes for the *parallel* algorithm. The *concurrent* algorithm achieved higher speedup performance than the *parallel* algorithm, but it was improved with smaller graph sizes. This result might seem surprising, but as the disk/CPU nodes increase more agents will run simultaneously and for the same number of agents the problem with less nodes (less links) gets solved faster. Total number of parallel threads depends on the HiTi graph levels, and the number of marked nodes in the graph and is not affected by the number of disk/CPU nodes. So, in our experiment we only changed the disk/CPU nodes configuration with fixing the HiTi graph (fixing graph size), which means that the total number of parallel threads was fixed. The only effect on the parallel algorithm was that more threads were executed simultaneously on the new disks and this explains the increase in speedup.

It is clear from the results that our concurrent and parallel algorithm shows almost a linear speedup as the number of disk/CPU nodes is increased.

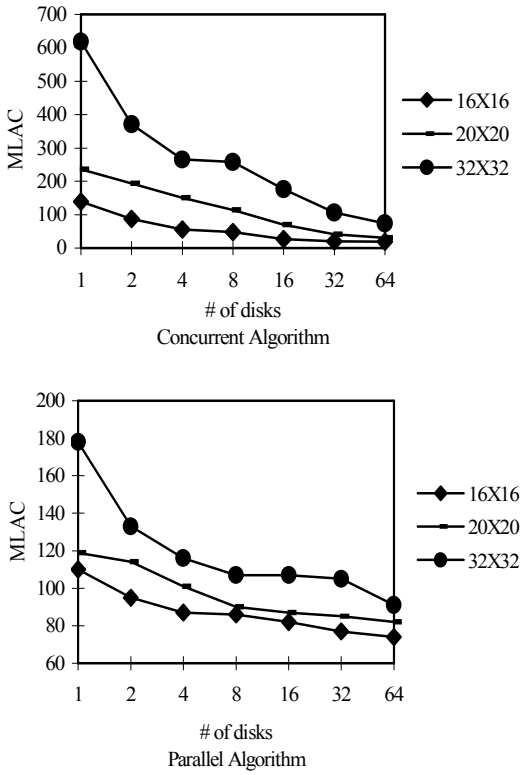

Figure 5. Speedup: Number of disk/CPU nodes vs. the maximum number of links accessed among all the agents for graph sizes of 16X16, 20X20, and 32X32 nodes.

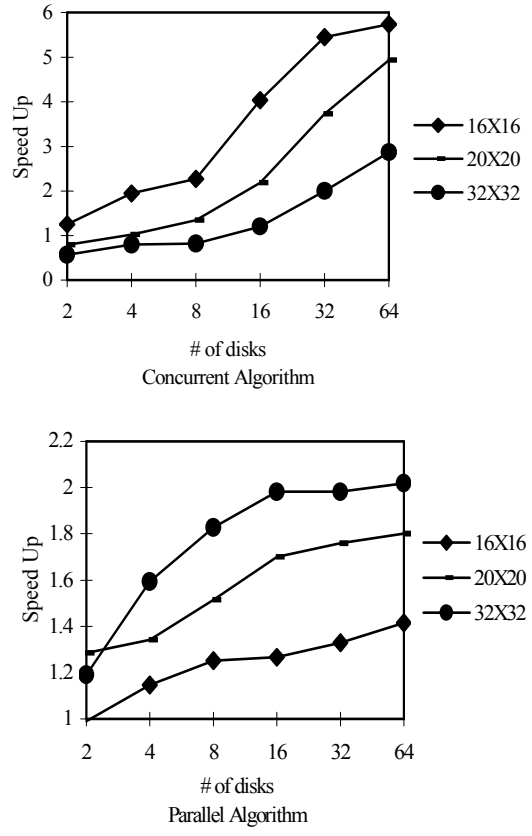

Figure 6. Speedup: Number of disk/CPU nodes vs. the speedup for graph sizes of 16X16, 20X20, and 32X32 nodes.

4.2.2. Size-Up

In sizeup experiments, we examine how the *parallel* and *concurrent* algorithms perform on a fixed number of disk/CPU nodes as we increase the size of the graphs. Figure 7 shows this for three different disk/CPU configurations where the graph is increased from 10X10 to 24X24 nodes. It is clear from the figures that as we throw in more disk/CPU nodes for the same graph size, we get a faster response (which is measured by *MLAC*). Concurrent algorithm exhibits better sizeup results than the parallel algorithm, i. e., by doubling the graph size we are not doubling the response time (*MLAC*) as we increase the disk/CPU nodes size. The same reasoning that was applied to the speedup case applies to the sizeup case for the differences between the performance of the parallel and concurrent algorithms.

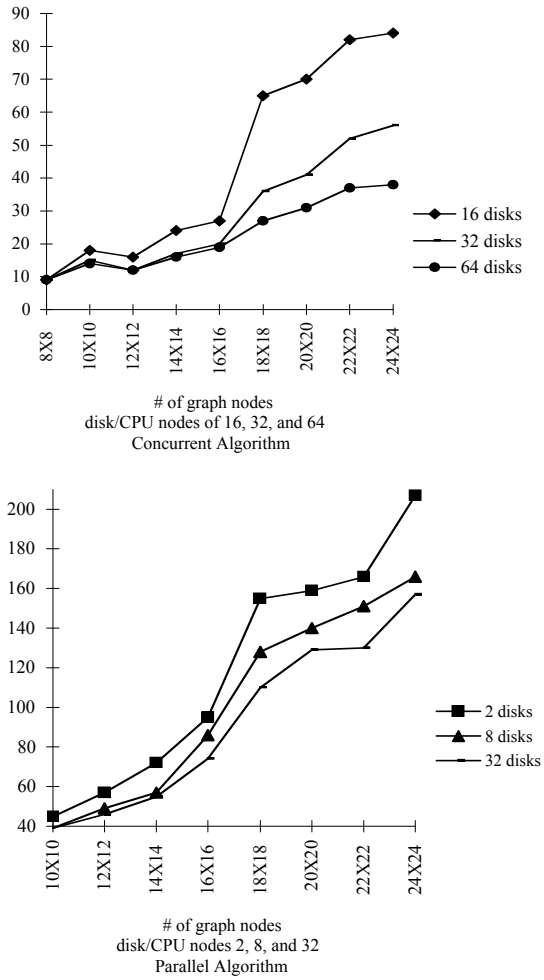

Figure 7. SizeUp: Number of graph nodes vs. the maximum number of links accessed among all the agents.

4.2.3. Scale-Up

In the next experiments, we increased the size of the graph in proportion to the number of disk/CPU nodes in the system; i. e., each disk/CPU node has a fixed number of graph nodes. The performance results of these experiments are shown in Figure 8. It is clear that both algorithms scales well, i. e., *MLAC* doesn't increase dramatically as the graph size increases with the increase of disk/CPU nodes. The increases in *MLAC* are due to the fact that when we increase the number of disk/CPU nodes we are increasing the graph size. This leads to an increase in the average length of shortest paths in the graph; i. e., *MLAC*. It is true that the amount of data per disk/CPU node does not change for a given experiment, but the graph size changes and the graph nodes in different disk/CPU nodes are not completely independent. So, *MLAC* is not remaining constant; as it should ideally; as the disk/CPU nodes is increased. The results show nice scaleup, especially for the concurrent algorithm. Overall, we can conclude that concurrent and parallel algorithms can indeed be used for large graph sizes.

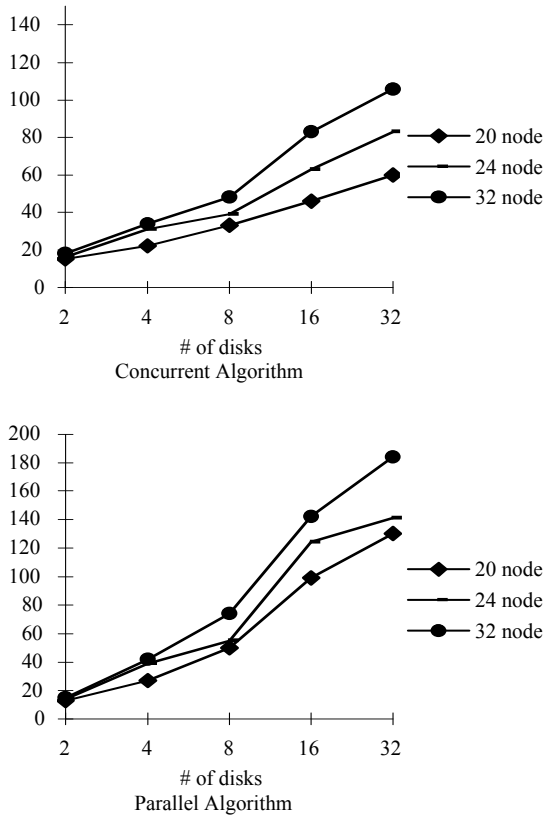

Figure 8. Scale-up: Number of disk/CPU nodes vs. the maximum number of links accessed among all the agents.

4.2.4. Replication

In this section we show the results of replication of the graph nodes. We restricted the experiment to only one replication of the graph. Figure 9 shows the results of replication effect as the graph size increases from 12X12 to 22X22 nodes, and the effect of increasing agents/disk accesses. In the figure on the left, the graph size ranges from 12X12 to 22X22 nodes for two agents/disk accesses limits of 4, and 16. In the right figure, we change the agents/disk limits from 2 to 32 for two graph sizes of 16X16, and 32X32 nodes. From the results, we may conclude that as the graph size increases, we see a better replication effect on the performance of the concurrent algorithm, especially

when we have smaller agents/disk accesses limit. And as the agents/disk accesses increasing the effect of replication reduce, since we are able to execute more agents simultaneously (which was the mean reason for using replication).

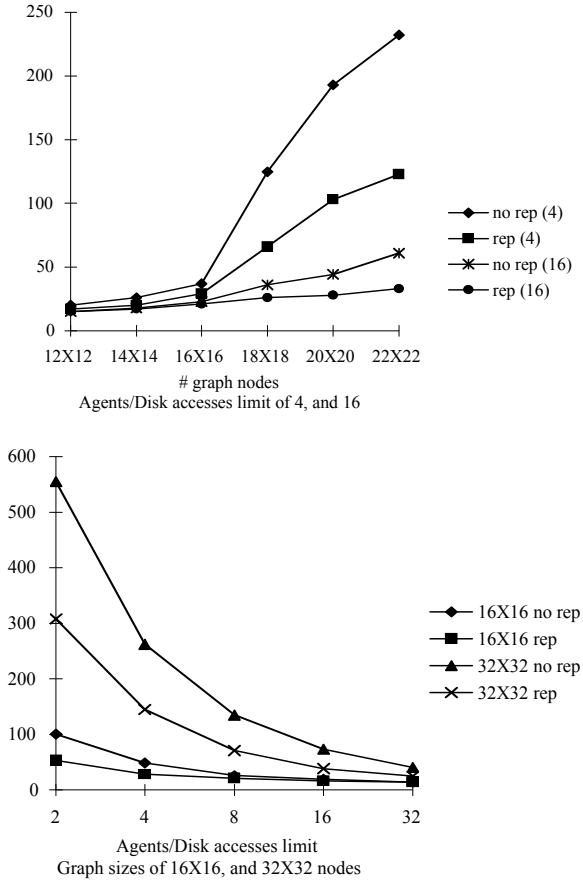

Figure 9. Replication - Concurrent Algorithm.

4.2.5. Comparison of Parallel, Concurrent and SPAH

In this section we compare the original sequential algorithm (*SPAH*) to the parallel and concurrent algorithms. In these experiments, the graph sizes ranges from 24X24 to 32X32 nodes, the number of disk/CPU nodes varied from 2 to 64, and the agents/disk accesses were limited to 8, 16, and 32. The results for the run are shown in Figure 10. Both of the parallel and concurrent algorithms outperformed the SPAH algorithm in terms of *MLAC*. For small graph size the concurrent algorithm outperforms the parallel algorithm in terms of *MLAC* and scalability. For larger graph size the parallel algorithm outperforms the concurrent algorithm for smaller disk/CPU nodes configuration. As the disk/CPU nodes, and the agent/disk accesses limit increases the concurrent algorithm performs the best.

The next experiments measures the effect of changing the agents/disk accesses, and the change in disk/CPU nodes on *MLAC* for both parallel, and concurrent algorithms. The results of the experiments are shown in Figure 11. Both algorithms show a good scaleup. They both perform better with increasing the disk/CPU nodes, the agents/disk accesses, or both. The increase in agents/disk accesses has a larger affect (reducing response time) on the concurrent algorithm than on the parallel, especially for small disk/CPU configuration. On the other hand, in the parallel algorithm the agents/disk accesses effect reduces as the disk/CPU increases. This is where we reach a point that every parallel thread is executed on a different disk, and each thread performance is limited to its sequential base. That is why the increase in agents/disk accesses does not affect it anymore. Figure 12 shows that as we increase the agents/disk accesses limit, the total number of agents created stays almost constant for larger disk/CPU nodes configuration. This means that we get better performance by increasing the number of the disk/CPU nodes and at the same time we are not increasing the load on the system (since the total agents created remains almost constant).

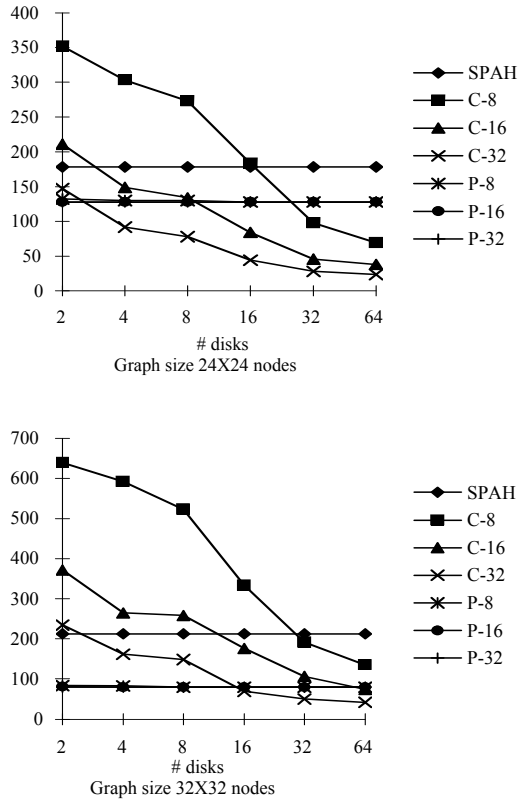

Figure 10. Comparison: Number of disk/CPU nodes vs. the maximum number of links accessed among all the agents for agents/disk accesses of 8, 16, and 32.

4.2.4. Random Graphs

The last set of experiments is based on a randomly generated graph with 24X24 nodes, and an average out-degree of 3. We conducted the same single-source path search experiments described in the previous sections. Interestingly, the experiments show the same performance as we had for the synthetic graphs. SPAH remains the worst, while the concurrent shows better performance for larger disk/CPU nodes and parallel

algorithm shows better performance for smaller disk/CPU nodes configuration or smaller agents/disk accesses limits.

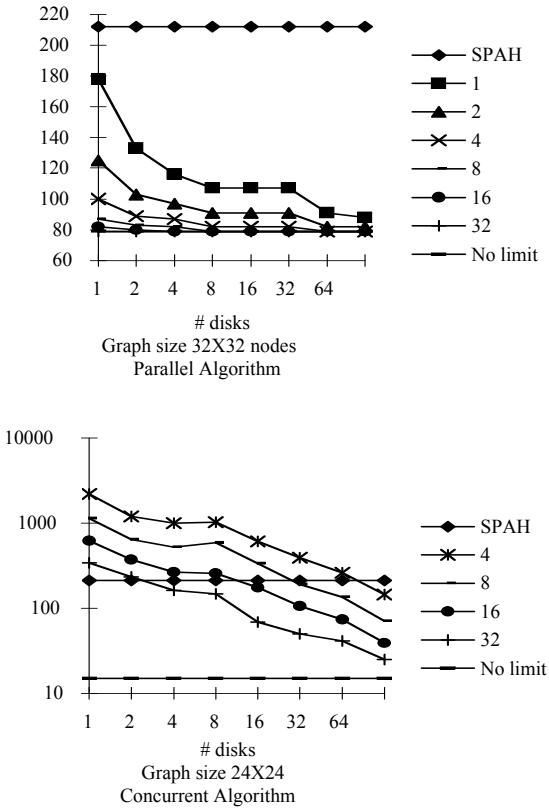

Figure 11. Comparison: Number of disk/CPU nodes vs. the maximum number of links accessed among all the agents for agents/disks accesses of 4, 8, 16, 32, and no limit.

5. Conclusion

This study proposed both concurrent and parallel versions of a recently developed graph model, Hierarchical mulTigraph (HiTi), as a solution for speeding up the computation of shortest path problem on a shared nothing architecture (i. e., Teradata multimedia database system). The concurrent algorithm allows simultaneous exploration of the search space by utilizing dynamically created agents across multiple disk nodes, which is efficiently supported by the Teradata multimedia database system architecture. The parallel algorithm breaks the problem into a set of smaller subproblems by exploiting a set of intermediate nodes that the shortest path passes through. We evaluated our algorithms via a simulation study and demonstrated that our concurrent and parallel algorithms show almost a linear speedup as the number of disk/CPU nodes is increased. Concurrent algorithm exhibited better sizeup, and scaleup results than the parallel algorithm.

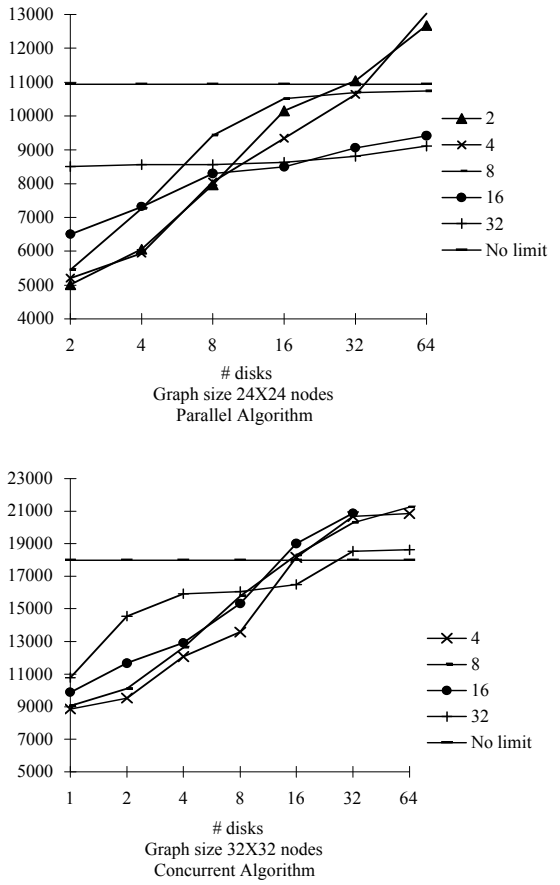

Figure 12. Comparison: Number of disk/CPU nodes vs. total number of created agents for agents/disks accesses of 2,4,8,16,32, and no limit.

Acknowledgments

Research Administration at Kuwait University (Project No EO02/01) has funded this research.

References

[1] Agrawal R., and Jagadish H. V., "Algorithms for Searching Massive Graphs," *Transactions on Knowledge and Data Engineering*, vol. 6, no. 2, pp. 225-238, 1994.

[2] Agrawal R., Dar S., and Jagadish H. V., "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM TODS*, vol. 15, no. 3, pp. 427-458, 1990.

[3] Bellman R., "On a Routing Problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87-90, 1958.

[4] Dijkstra E. W., "A Note on Two Problems in Connection with Graph Theory," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[5] Connell W. O., Ieong I. T., Schrader D., Watson C., Au G., Biliris A., Choo S., Colin P., Linderman G., Panagos E., Wang J., and Walter T., "A Teradata Content-Based Multimedia Object Manager for Massively Parallel Architectures," *in Proceedings of the ACM SIGMOD*, pp. 68-78, 1996.

[6] Connell W. O., Schrader D., and Che H. H., "The Teradata *SQL3 Multimedia Database Server*," pp. 68-78, 1996.

[7] Floyd R. W., "Algorithm 97 (Shortest Path)," *Communications of the ACM*, vol. 5, no. 6, pp. 345, 1962.

[8] Ford L. R., and Fulkerson D. R., "Flows in Networks," *Princeton University Press*, 1962.

[9] Houtsma M. A. W., Apers P., and Ceri S., "Data Fragmentation for Parallel Transitive Closure Strategies," *in Proceedings of the 9th International Conference on Data Engineering*, pp. 447-456, 1993.

[10] Houtsma M. A. W., Apers P., and Ceri S., "Distributed Transitive Closure Computations: The Disconnection Set Approach," *in Proceedings of the 16th VLDB Conference*, pp. 335-346, 1990.

[11] Houtsma M. A. W., Wilschut A., and Flokstra J., "Implementation and Performance Evaluation of a Parallel Transitive Closure Algorithm on Prisma/db," *in Proceedings of the 19th VLDB Conference*, 1993.

[12] Huang W., Jing N., and Rundensteincr E., "A Semi-Materialized View Approach for Route Maintenance in IVHS," *in Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, pp. 144-151, 1994.

[13] Huang W., Jing N., and Rundensteincr E., "Effective Graph Clustering for Path Queries in Digital Map Databases," *in Proceedings of the 5th International Conference on Information and Knowledge Management*, 1996.

[14] Huang W., Jing N., and Rundensteincr E., "Hierarchical Optimization of Optimal Path Finding for Transportation Applications," *in Proceedings of the 5th International Conference on Information and Knowledge Management*, 1996.

[15] Johnson D. B., "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, vol. 24, no. 1, pp. 1-13, 1977.

[16] Jung S., and Pramanik S., "An Efficient Path Computation Model for Hierarchically Structured Topological Road Maps," *IEEE Transaction on Knowledge and Data Engineering*, 2002.

[17] Jung S. and Pramanik S., "HiTi Graph Model of Topological Road Maps in Navigation Systems," *Proceedings of the 12th International Conference on Data Engineering*, pp. 76-84, 1996.

[18] Shahabi C., Kolahdouzan M., and Sharifzadeh M., "A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases," *ACMGIS*, McLean, VA, USA, 2002.

[19] Shekar S., Kohli A., and Coyle M., "Path Computation Algorithms for Advanced Traveler Information Systems," *in Proceedings of the 9th International Conference on Data Engineering*, pp. 31-39, 1993.

[20] Warshall S., "A Theorem on Boolean Matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11-12, 1962.

Maytham Safar is currently an assistant professor at the Computer Engineering Department at Kuwait University. He received his PhD degree in computer science from the University of Southern California in 2000. He has one book and more than 12 articles, book chapters, and conference/journal papers in the areas of databases and multimedia. His research interests include peer-to-peer networks and multidimensional databases. He served on many conferences as a reviewer and/or a scientific program committee member such as ICDCS 2000, EURASIA-ICT 2002, ICWI 2002, ICME 2002, ICWI 2003, AINA 2003, ICWI 2004, iiWAS 2004. He also served as a member on the editorial board or a reviewer for many journals such as IEEE Transactions on Multimedia Journal, ACM Computing Reviews, Journal of Digital Information Management (JDIM), Multimedia Tools and Applications Journal (MTAP).