Subgraph Isomorphism on the BBN Butterfly Multiprocessor*

John Costanzo Lawrence Crowl Laura Sanchis
Mandayam Srinivas

Department of Computer Science
University of Rochester
Rochester, NY 14627

September 14, 1995

Abstract

This report describes an algorithm for finding subgraph isomorphisms for a restricted class of graphs and a parallel implementation of the algorithm on the BBN Butterfly Multiprocessor. This effort was part of a larger project to assess the suitability of the Butterfly architecture for a variety of machine vision tasks. Our algorithm searches a tree in which each node represents a partial assignment of vertices in the smaller graph to vertices in the larger graph. The algorithm prunes the search tree using properties of the two graphs as constrained by the partial mapping. These properties are vertex connectivity, distance between vertices, and the local topology of vertex clusters. By carefully balancing the computational load and the contention for shared resources, our algorithm achieves almost linear speedup in the processing rate of search tree nodes. However, the speedup of isomorphism detection rate is poor when looking for few isomorphisms, and good only when looking for many isomorphisms. We present an analysis of why we believe this property is intrinsic to algorithms that parallelize the tree search without parallelizing the node computations. We also discuss the effectiveness of the Butterfly architecture and programming environment in implementing such parallel algorithms.

*This work was supported in part by the Defense Advanced Research Projects Agency U.S. Army Topographic Labs under grant number DACA76-85-C-0001, in part by the National Science Foundation under grant number DCR-8320136 and in part by an AT&T Foundation Fellowship.

1 Introduction

The report describes the results of a project to implement a parallel algorithm for subgraph isomorphism on the BBN Butterfly multiprocessor. This project is part of a larger effort to assess the suitability of the Butterfly for implementing parallel algorithms for machine vision [Br86]. Our results indicate that the Butterfly is suitable for implementing an efficient algorithm for our problem.

1.1 Problem Refinement

The original problem statement, under the title of "Graph Matching", is as follows:

> The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of occurrences of (an isomorphic image of) H as a subgraph of G. As a variation of this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of absolute differences between corresponding pairs of labels is minimum.

The above problem statement includes two problems. The first problem requires us to enumerate every isomorphism. Because of the regularity of G and H and because vertices in G have much larger degree than vertices in H, there are likely to be a very large number of isomorphisms. We suspect that the average number of isomorphisms is exponential in the size of H. Thus, any program running on a computer such as the Butterfly would require prohibitively large amounts of computation, even on the average. (Note that even the problem of determining the existence of an isomorphism is NP-complete, and is commonly referred to as the Subgraph Isomorphism Problem [GJ79]). Since we are mainly interested in assessing the suitability of the Butterfly architecture for this problem under a very tight development time constraint, we decided to focus on the average time required to find a small, fixed number of isomorphisms, (*e.g.*, 10 or 100) rather than the average time to enumerate all of them.

The second problem is a generalization of Subgraph Isomorphism to graphs with edge and vertex costs. Here each isomorphism is assigned a cost according to some formula, and it is required to find the minimum cost isomorphism. Again, because of development time constraints, we do not address this problem in this report.

Finally, a word about terminology. The word *matching* has a specific meaning in graph theory, namely a set of vertex-disjoint edges in a graph. Graph Matching would thus suggest one of the well-known problems that involve finding such matchings in graphs. We therefore prefer to use the standard terminology, and will hereafter refer to our problem as Subgraph Isomorphism.

1.2 Approach

Our parallel algorithm is based on a modification of Ullmann's sequential tree-search algorithm for subgraph isomorphism [Ul76]. Ullmann's method generates a search tree where each node in the tree represents a possible partial isomorphism. Let $v_1, \ldots, v_n$ be the vertices of H, ordered so as to correspond to the depth in the search tree. A node at depth k contains a single mapping for vertices $v_1, \ldots, v_k$ and a set of possible mappings for vertices $v_{k+1}, \ldots, v_n$ At the leaf nodes of the tree, all vertices, $v_1, \ldots, v_n$, have a single mapping and the node represents an isomorphism.

Ullmann's algorithm prunes the search tree by eliminating mappings that are infeasible because they violate connectivity requirements. We view this procedure as the application of a *connectivity filter*. The regularity of the graphs for our problem is such that the connectivity filter will be less effective than in the general case. Because of this, we have attempted to improve the pruning efficiency of Ullmann's algorithm by precomputing a good vertex traversal order and by adding two new filters, the *distance* and *configuration* filters.

There are basically two aspects of the sequential algorithm that can be parallelized: the computation at each node and the exploration of the search tree. We have chosen to focus on the latter aspect for two reasons. Parallelizing the exploration of the search tree has the advantage of requiring minimal communication between processors, which implies low synchronization and contention costs. We also believe that it is the more interesting aspect, since the node computations tend to represent matrix computations, and parallelizing such computations on the Butterfly is better understood.

The basic algorithm is to generate the graph, perform the precomputations, and spawn off a number of processes to search the tree. The precomputations include ordering the vertices to increase search effectiveness, and precomputing constant information for the distance and configuration filters. Each process searching the tree contains a loop in which it obtains a search tree node from the shared stack, applies the filters to the node, and places the feasible children of the node, if any, on the stack. The loop continues until either the appropriate number of isomorphisms has been found, or there are no more nodes left to process. For a single processor, this algorithm is equivalent to a depth first search of the tree.

1.3 Results

Our algorithm was successfully implemented on the Butterfly. Experiments were conducted for finding between 1 and 100 isomorphisms, using between 1 and 96 processors. Figures 1-4 summarize the experimental findings.

The program exhibits almost linear speedup in the node processing rate. However, the solution rate (*i.e.*, the number of isomorphisms detected per second) shows good speedup only when the program is looking for 100 isomorphisms. The solution rate speedup is poor when looking for fewer isomorphisms, a phenomenon that we believe is intrinsic to algorithms that parallelize the tree search without parallelizing the node computation. Going in the other direction, we have evidence to suggest that the solution rate speedup will be almost linear if we are looking for a very large number of isomorphisms. However, the actual solution time would be prohibitive in that case.

Our effort indicates that the Butterfly architecture provides an effective balance of features for implementing parallel algorithms of the kind likely to be used for problems similar to ours. However, architectural suitability alone does not guarantee effective utilization of a parallel processor. The machine must also be equipped with a workable programming environment. We found that the Butterfly's programming environment, evidently reflecting the state of the art in parallel programming environments, is currently inadequate for rapid and reliable program development.

2 Algorithm Details

This section describes the details of our algorithm. We examine random graph generation, search vertex traversal order determination, and search tree node filtering.

2.1 Graph Generation

Generating a graph according to the given specifications is an interesting problem in itself. Consider an incremental algorithm to generate the graph H, which has 30 vertices each connected at random to 3 others. Thus H is a 3-regular graph. The 3 neighbors of some vertex v must be chosen in such a way that the triple that is chosen has the same probability as any other feasible vertex triple. (A triple is feasible if it does not violate the degree constraint for any vertex in the triple). Incrementally, we must ensure that the next vertex added to the triple is chosen at random from the remaining feasible neighbors.

It is not hard to see that such a procedure might not always yield a graph H, since we might run out of feasible neighbors prematurely, before all vertices have received their full complement of 3 edges each. We illustrate this phenomenon with a small example. Let K_4 be the complete graph on 4 vertices and $K_{3,3}$ be the complete bipartite graph with 3 vertices in each partition. Both K_4 and $K_{3,3}$ are 3-regular. Suppose it is required to generate a 3-regular graph with 6 vertices. The procedure described above might incrementally generate K_4. At that point, 4 vertices have degree 3 and 2 have degree 0. The only remaining feasible edge will connect the two zero-degree nodes, which will then each have degree 1. Since no further edges can be added, the procedure fails.

If we relax the requirement that the neighbors of a vertex must be chosen at random, it is simple to generate a 3-regular graph on n vertices, for n even. (Exercise for the reader: prove that for odd n, no 3-regular graph exists). We simply have $\lfloor n/4 \rfloor - 1$ copies of K_4 and one copy of either K_4 or $K_{3,3}$. However, it is required to choose neighbors at random. We therefore strengthen our definition of neighbor feasibility.

A vertex w is a feasible neighbor of v if adding the edge (v, w) does not violate the degree constraint for either v or w, and, furthermore, after adding the edge (v, w), the residual degree sequence is feasible. The *residual degree* d_i of a vertex i is the number of edges it needs to bring its degree up to 3. The *residual degree sequence* of a graph is formed by arranging the residual vertex degrees in non-increasing order. It can be shown [BM76] that the residual degree sequence is *feasible* if, for $1 \leq k \leq n$, $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min\{k, d_i\}$. If a residual degree sequence is feasible, then we know that the construction can be completed with the edge (v, w) included.

Following the approach in [Ti79], our graph generator uses the neighbor feasibility test on-line. Every time a neighbor is examined, the degree constraint and the residual degree criterion are checked. The neighbor is deemed feasible only if it passes the test. Three feasible neighbors are then chosen at random for each vertex. The 10-regular 100-node graph G is similarly generated.

2.2 Vertex Ordering

The search procedure assumes a vertex traversal order. In this section we show how to choose this order to our advantage. Consider the situation in the algorithm after vertices $v_1, \ldots, v_k$ have been mapped. An unmapped vertex that is connected to a larger number of mapped vertices will have more restrictions on possible mappings. Thus, our goal is to choose the next vertex to visit in the search as the one that is maximally connected to vertices already mapped. We believe that this procedure will tend to cause pruning earlier rather than later, by applying restrictions to the edge mappings as soon as possible. This will make the search tree sparser and reduce the search space.

We compute a vertex ordering as follows. For each vertex we record a label, representing its position in the eventual ordering, and its residual degree. Initially every vertex is unlabeled and has residual degree 3. At each step $i, 1 \leq i \leq n$, we choose an unlabeled vertex of minimum residual degree, label it with i, and decrement the residual degree of its unlabeled neighbors. The procedure thus tends to choose vertices highly connected to the vertices already labeled.

2.3 Node Filters

We have implemented three filters, a connectivity filter modified from Ullmann's original filter, a distance filter based on the distances between vertices in the two graphs, and a configuration filter based on the possible configurations a vertex and its three neighbors can assume in the H graph. Note that the distance and configuration filters, when combined with use of the connectivity filter to restore consistency after 1's have been eliminated, are a generalization of, and hence more powerful than, the connectivity filter alone.

Ullmann's Approach

In Ullmann's implementation, each node of the search tree is associated with an m by n binary matrix M where m is the order of H and n is the order of G. The (i, j) entry of M is set to 1 if we are still considering the possibility that vertex v_i of H may be mapped to vertex w_j of G. At depth d in the search tree, M contains a single 1 in each of its first d rows. To generate the children of a node at depth d, we clear all but one of the 1's in row $(d+1)$ of M and zero out the rest of the column occupied by the remaining 1. So the branching factor is the number of 1's in row $d+1$. At the leaf nodes, exactly one 1 is set in each row of M and each column has at most one 1 set. To create the root matrix M we set the (i, j) entry of M to 1 unless we have reason to believe that vertex v_i of H cannot be mapped to vertex w_j of G.

Ullmann describes a refinement procedure which at each node attempts to get rid of 1's by checking connectivity requirements. If this removal results in a row of M losing all its 1's, the node can be pruned from the search tree. The connectivity check is as follows. For each 1 in M, say at position (i, j), the following check must be made. Let v_x be any neighbor of v_i in H. Then there must be some 1 in v_x's row in M corresponding to some neighbor of w_j in G. That is, there must be some y such that w_y is connected to w_j in G and entry (x, y) of M is set to 1. If no such y exists, then the mapping from v_i to w_j is impossible and we may set entry (i, j) of M to zero. Such a check must be done for every neighbor v_x of v_i. After all 1's in M have been checked in this way, the process must be repeated if any 1's were eliminated during the pass. Checking connectivity at a leaf node determines whether or not we have found an isomorphism.

Our Modifications

Ullmann's connectivity filter, or refinement procedure described above, can be made more efficient by making the following 2 observations:

1) It is not necessary at each node to check each 1 in the M matrix for connectivity consistency. In fact, it is only necessary to check at each stage the neighbors of those H vertices whose rows in the M matrix have lost 1's since the last connectivity check. Of course, since this check may result in some 1's being eliminated, it may recursively cause more checks. This implementation of the connectivity filter is described in more detail below.

2) At depth d, the vertices in rows 1 through d need not be checked even under the circumstances described in (1). This is because these vertices have only one 1 set in their rows in M, and by previous connectivity checks all 1's in their neighbors' rows must correspond in G to neighbors of the vertex indicated by the single 1. In particular, this means that at leaf nodes of the search tree no checking needs to be done at all, and whenever we reach such a leaf node we are guaranteed to have found an isomorphism.

We now describe the procedures used to eliminate 1's at the beginning and throughout the search.

Connectivity Filter

This filter is invoked by giving it a list of vertices in H whose rows in M have recently lost 1's. This can happen, for instance, when new nodes are generated by branching in the search tree, or when other filters, described below, have been applied, resulting in the elimination of 1's.

The filter maintains a stack containing the numbers of the vertices whose neighbors must still be checked. This stack is initialized with the list of vertices passed in to the procedure. The algorithm proceeds by repeatedly taking vertices off the stack and processing them; this may result in more vertices being pushed onto the stack. The procedure ends when the stack is empty.

The following is done for each vertex v_i popped off the stack. If a 1 is in position (i, j) of M, then the jth row of the incidence matrix of G is retrieved. This vector will have a 1 corresponding to each neighbor of w_j in G. All such vectors are retrieved for each j such that (i, j) is 1 in M, and these vectors are ORed together. The resulting vector is ANDed with each row in M corresponding to a neighbor of v_i whose vertex number is not less than d, the current depth. For each such neighbor of v_i, the result of the AND operation is installed as the new row in M for this vertex. If the row for the vertex in M has been zeroed out, the corresponding node is pruned from the search tree. If one or more 1's have been eliminated from the row, the vertex is pushed onto the stack if it is not already there.

Distance Filter

We define the distance between two vertices in a graph to be the smallest number of edges that must be crossed in order to get from one vertex to the other. Clearly, the distance between the isomorphic images in G of vertices u and v must be less than or equal to the distance between u and v in H, since G has an image for every edge in H and potentially contains some shortcuts between the images of u and v.

This observation forms the basis of the distance filter, which we describe in this section. The distance filter requires knowledge of the distance between every pair of vertices in both G and H. We precompute these in two matrices by repeated breadth-first search starting at each vertex in G and H.

The distance filter may be applied every time a new node has been generated. Suppose a node has been generated at depth i by zeroing out all 1's except one in row i of M. Assume the remaining 1 is in column j. For each 1 in each row of M after row i, the following is done. Let the 1 be in position (l, m) of M $(l > i)$. That is, the 1 signifies that vertex v_l in H may be mapped to vertex w_m of G. For this to be so the distance in G from w_j to w_m must be less than or equal to the distance in H from v_i to v_l. If this check fails, then the 1 at (l, m) is zeroed out.

If the distance filter application results in elimination of 1's in one or more rows of M, the connectivity filter is invoked to restore connectivity consistency.

Configuration Filter

The configuration filter is based on the fact that because every vertex in the H graph has degree 3, one can enumerate the 4 types of configurations that a vertex v and its 3 neighbors can be in.

Type 0 configuration occurs when none of the neighbors of v are connected to each other by an edge. Type 1 configuration occurs when exactly two of the neighbors of v are connected to each other. Type 2 configuration occurs when exactly two pairs of the neighbors of v are connected to each other. Type 3 configuration occurs when all three neighbors of v are connected to each other, forming an isolated clique of size 4. Each vertex in H can be determined to be in exactly one of these 4 types.

In the graph G, each vertex has degree 10. If a vertex v in H is mapped to a vertex w in G, then the 3 neighbors of v in H must be mapped to three neighbors of w which together with w form a configuration having at least the connectivity of v's configuration. In other words, if v has configuration type k, v and its three neighbors must be mapped to a vertex w and 3 of w's neighbors forming configuration type l where $l \geq k$. Each vertex w in G has 120 3-tuples of neighbors and a configuration type corresponding to each of these 120 3-tuples.

Application of the configuration information requires knowledge of the configuration type of each vertex in H and of the types associated with each vertex in G and each of its 3-tuples of neighbors. This information is computed and stored at the beginning of the algorithm.

One application of the configuration information takes place when the matrix M is initialized at the root node of the search tree. At this point each feasible mapping of a vertex v_i to a vertex w_j in G is checked, to make sure that for some 3-tuple of neighbors of

w_j, w_j has configuration type at least as large as that of v_i. If this is not the case, the (i, j) entry of M is zeroed out.

In addition, a configuration filter may be applied whenever a new node is generated in the search tree. Suppose a new node has been created at depth i. This has been done by zeroing out all 1's in row i of M except one, say in column j. Thus for all isomorphisms that may be found in this branch of the search tree vertex v_i of H is mapped to vertex w_j of G. We want to make sure that the neighbors of v_i are all mapped to neighbors of w_j which form the correct type of configuration with w_j. In order to do this the filter takes each 3-tuple of neighbors of w_j whose configuration type is at least as large as v_i's configuration type, and records the fact that each of v_i's neighbors may be mapped to any one of the vertices in the 3-tuple being considered. After this is done for each applicable 3-tuple of neighbors of w_j, the 1's corresponding to mappings for the neighbors of v_i which were not recorded in the above procedure are zeroed out in the matrix M. Since this process can result in elimination of 1's in the rows of M corresponding to v_i's neighbors, the connectivity filter should be then called to restore connectivity consistency.

3 Implementation

The program is implemented on the Butterfly using the C programming language and calling the Chrysalis operating system [BB85a] directly. Programming with Chrysalis is appropriate when there is need for shared data but not for a shared address space, and when the granularity of processes is high. The algorithm chosen has these characteristics.

The implementation has a main process to setup the computation, and several server processes to perform the tree search. The main process initializes a shared stack, consisting of tree nodes which have yet to be searched, to the set of possible tree nodes at depth one. The server processes extract a node from the shared stack, filter it, and place any feasible children back on the shared stack.

The main process: sets up the shared memory environment through which all processes will communicate; generates the random graph and first search tree node; precomputes the vertex order, static distance filter information and static configuration information; spawns the server processes; signals the processes to start searching; and waits for the processes to finish.

Each server process: connects to the shared memory environment; waits for the start signal; copies the constant problem and precomputation information into local memory; loops processing search tree nodes until the appropriate number of isomorphisms have been found; and signals the main process upon termination. To process a search tree node, the server process: retrieves a tree node from the shared stack; applies the filters to the node; and places any feasible child nodes back on the shared stack.

3.1 Sequential Operations

Not all elements of the program could be parallelized in the short time available to our project. As a result, some parts were implemented sequentially. This section discusses the reasons each was left sequential.

Graph Generation

We did not consider the random graph generation as part of the problem statement, so no effort was made to parallelize it.

Precomputation

The precomputation is graph dependent, and therefore is necessarily part of the problem statement. We did not parallelize the precomputation since it took only about 12 seconds on a single Butterfly node, which represents a small fraction of the total work. In addition, the precomputation is a relatively straightforward matrix computation, and parallelizing such computations on the Butterfly is relatively well understood. Therefore we did not feel that the coding effort would be justified. This precomputation time is not counted in the solution times cited later.

Spawning Processes

The server processes are sequentially started by the main process. We made no attempt to parallelize process startup because we feel that, in the context of vision, the isomorphism search is likely to be repeatedly called on a stream of problems. In such an environment, the processes would be started and left started, and the relative cost of process startup would become insignificant. Because of the time to start processes, a process spawned early could conceivably do a substantial amount of work on the search while other processes were still being started. This would adversely affect the timing results, so all processes start computation only when receiving a broadcasted "start" signal.

3.2 Parallel Operations

This section describes those aspects of our implementation that we have parallelized. These parallelizations strive to keep processors as independent as possible while keeping them as busy as possible.

Copying Precomputation Results

The precomputation results are placed in a central location by the main process. While the server processes could use this central location to access the appropriate data, such accesses would be subject to contention with the other server processes attempting to access the information. Such contention can result in a substantial performance degradation. Therefore, we have each server process copy the precomputation results to the local processor for the duration of the search. The access costs to the data are minimal with the local copy.

Tree Search

An obvious approach for parallelizing a tree search with such a large branching factor is to assign a processor to each top level subtree and have each processor then execute a sequential algorithm on the smaller problem. This approach has the advantage that there is almost no communication between processors, and hence they will not contend for information resources. There is, however, a disadvantage in that if a processor finishes early (because it eliminates a subtree), there is no work left for the processor and it becomes ineffective.

The first approach implemented had a central stack for all tree nodes. This eliminated the ineffectiveness of a processor finishing early. Unfortunately, the stack tended to grow very large. Due to the parallel, random, nature of stack access, the algorithm no longer has the space preserving properties of a depth first search. Indeed, the stack may grow as quickly as it would for a breadth first search, which would store a prohibitively large number of tree nodes for this problem. In addition, processors were always contending for access to the shared stack.

The second approach implemented used the shared stack only for search tree nodes at depth one and two. Below that, a local stack was used for searching a subtree in a strictly depth first manner. This reduced overall contention for the shared stack while still leaving plenty of work for a processor which finished a subtree early.

Shared Stack

The shared stack is implemented as a data structure in shared memory. Each server process is responsible for consistently updating the stack. Because server processes will start execution of the stack at arbitrary times relative to each other, these operations must insure that they do not interfere with each other. We solve this problem by having the server processes lock the stack using an atomic test and set operation when performing the critical data structure updates. When a server process finds the stack locked, the process busy waits until it finds the stack unlocked. Locking the stack produces contention for the stack. This contention in turn reduces the amount of effective parallelism that can be obtained from the program. Therefore, the program should seek to minimize the amount of time a process locks the stack relative to the amount of time processing away from the stack.

4 Analysis of Results

Our algorithm was successfully implemented on the Butterfly. Data was collected for s = 1,5,10,20 or 100, where s is the number of isomorphisms being sought, with p = 1,2,4,8,16,32,64 or 96 processors. For comparison purposes, each Butterfly node is about 50% slower than a Sun 2/50 workstation. Figures 1-4 summarize the experimental findings. Each data point represents between 6 and 10 trials. The majority of our experiments were run on a single, randomly generated instance of each of the graphs G and H. More robust results would be obtained if we were to run our program on different random graphs.

Our experiments indicate that using both the distance and the configuration filter is only marginally more useful than using either of these filters alone. In addition, we found that the distance filter took less time to run. Because this latter fact was observed too late in the experimentation, the configuration filter alone was used for all of our trials. However, the nature of the parallelization is such that using any combination of filters at each node should yield the same qualitative results in terms of speedup.

4.1 General Analysis

This section provides a general analysis of the parallel algorithm. Analysis specific to our implementation is discussed in the next section.

Node Processing Rate

Figure 1 shows the number of search tree nodes processed per second as a function of the number of processors. Our program achieves a near linear speedup in the node processing rate, indicating that our program is an effective parallelization of the tree search algorithm.

Solution Rate

Figure 2 shows the number of solutions generated per second as a function of the number of processors. This figure clearly shows that the program did not achieve linear speedup in problem solution time. One may observe, however, that the speedup gets better as larger numbers of isomorphisms are sought. This seems to confirm the conjecture that near linear speedup would be achieved by our program if we were to look for a sufficiently large number of isomorphisms. This conjecture is supported by the following analysis.

Suppose we are looking for one isomorphism. Consider a search tree with two subtrees. A sequential algorithm will explore one subtree first, and then, if a solution is not found, will explore the second subtree. In parallelizing this search for two processors, we would send one processor down each of the subtrees. Now, assume the solution is found in the first subtree. The first processor will find the solution while the second processor will not. However, the first processor will find the solution in the same amount of time as the sequential version. The second processor will contribute nothing to the problem solution time.

This argument can be extended to multiple processors, and multiple solutions. As different processors explore different branches of the search tree, some will find solutions and some may not. Whether or not a processor finds a solution depends on whether or not the processor can find a solution before the required number of solutions are found by other processors. The extent to which processors search without yielding solutions determines the amount of unprofitable work expended. If some of this work would not take place when using only one processor, then this work represents a limit to the speedup that can be achieved with parallelism.

Consider the case where we are searching for a few isomorphisms and the search tree contains a large number of isomorphisms distributed evenly among its branches. A sequential algorithm would only need to go down a few branches to find the required isomorphisms. The parallel implementation, however, would explore many branches at once, and would terminate when an appropriate (small) number of processors have found isomorphisms. In the meantime, the remaining processors would have expended a considerable amount of work. Thus parallelization in this case is only marginally productive.

Now consider the case where we are searching for a large number of isomorphisms and the search tree contains a large number of isomorphisms distributed evenly among its branches. Most processors will contribute solutions before the appropriate number of isomorphisms have been found. Thus, processors will contribute a larger percentage of their total work towards the set of solutions, and the efficiency of the parallel algorithm will be higher.

Effective Processors

Figure 3 shows the number of effective processors as a function of the actual number of processors. This figure again points out the greater effectiveness of parallelization when a greater number of isomorphisms is being sought.

Solution Time

Figure 4 shows the time in seconds for finding a solution as a function of the number of solutions being sought. Note that the unit cost of finding a solution does not increase as more isomorphisms are sought, but instead seems to slightly decrease.

4.2 Implementation Analysis

A high degree of contention for a shared resource can severely degrade overall problem solution time. Much of the art of parallel programming consists of balancing the need for minimizing contention at shared resources with the necessity of providing enough shared information for the processors to work effectively.

The current program still has contention for the shared stack, especially at startup, which is probably why we achieve slightly less than linear speedup for the node processing rate. In addition, when looking for 1 isomorphism, the number of nodes evaluated per second actually decreases beyond $p = 64$. We suspect this may be because there is an insufficient amount of work to overcome the initial shared stack contention.

Given more time, we would revise our technique for assigning the initial tree nodes to processors. The first approach we would take at this point would be to arbitrarily assign the tree nodes at depth one to the processors as in the naive approach. However, instead of keeping an entire subtree to itself, each processor would place on the shared stack those tree nodes at depths two or three that it will not immediately process. This would restrict startup contention to that caused by placing children on the shared stack, which is significantly less expensive than retrieving children from the shared stack.

4.3 Other Parallelizations

This section discusses some approaches that we might take in parallelizing some other aspects of the computation, such as the precomputations and the node filtering.

The node filters and the precomputations are array based and may generally be parallelized with a "parallel for-loop". As such, they are more suitable for implementation under the Uniform System [BB85b]. Parallelizing these computations can be effective. However, the precomputation results and individual tree search nodes would become shared resources and contention for these resources could become significant.

Since the number of available processors is limited, an interesting question arises regarding the tradeoff between the two aspects of parallelization. In other words, what fraction of the processors should we devote to speeding up the node computations, and what fraction should we devote to exploring more nodes. One advantage of parallelizing node computations is that it does not entail sending processors down unproductive subtrees.

The following subsections describe some of the ways in which these parallelizations could be performed.

Vertex Ordering

The current sequential vertex ordering algorithm chooses arbitrarily among vertices contained in the bin with the minimal residual degree. This algorithm may be parallelized by having several processors extract nodes from the appropriate bin. This approach requires care to ensure that the actions of one processor reducing a vertex's residual degree do not adversely affect the processing of another vertex.

Distance Precomputation

The distance matrix may be parallelized by assigning each processor to a distinct vertex in the two graphs and having the processor compute the distances to all other vertices. The parallelized distance computations are more appropriate for implementation under the Uniform System.

Configuration Precomputation

Each processor is assigned a vertex in each of the graphs. Each processor then independently computes the vertex's configurations.

Connectivity Filtering

The stack based connectivity filter would be parallelized in much the same manner as the stack based tree search. However, such parallelization would require careful thought to ensure that the algorithm would not be affected by one processor turning 1's into 0's while another processor examines that information.

Distance Filtering

The distance comparisons are independent and may be distributed among the different processors.

Configuration Filtering

Parallelization of the configuration filter may be achieved by distributing the work associated with each 3-tuple of the neighbors of the mapped G vertex among the different processors.

5 Conclusions

The purpose of this exercise has been to examine the suitability of the Butterfly computer for solving certain versions of the subgraph isomorphism problem involving fixed size regular graphs. Since our interest is mainly in studying the effectiveness of parallelization, we decided to restrict ourselves to finding a small number of isomorphisms. We also chose to parallelize the exploration of the search tree since the effect of such a parallelization is interesting but relatively unexplored, both in theoretical and practical terms.

5.1 Performance

The program showed effective processor utilization in terms of the gross amount of work done. The average number of search tree nodes processed per second increased almost linearly with the number of processors. This supports our claim that our program is an effective parallelization of our algorithm.

We also found that using more processors resulted in faster detection of isomorphisms, although the speedup is not linear in the number of processors. The speedup is reasonably good for $s = 100$, but is poor for smaller values of s. Our results suggest that the speedup will be almost linear if s is very large. We believe that this behavior is inherent in the nature of the approach to the problem, and independent of the architecture used to implement the approach.

We also found that in general, the unit cost of finding an isomorphism decreased slightly as the number of isomorphisms sought increased. We did not look for more than 100 isomorphisms, however, since those alone took the program more than an hour to find using 1 processor.

5.2 Butterfly Suitability

The Butterfly is suited to the types of parallelization techniques employed in our program. In general, we find the Butterfly architecture is an effective parallel architecture, but that the programming environment needs significant work. The weakness of the programming environment should be considered more of a statement on the state of the art in parallel programming rather than a statement on BBN or the Butterfly.

Architecture

There is a cost associated with ability to share information, regardless of whether or not information is actually shared. Unfortunately, architectures which seek to minimize this cost tend to raise the cost of sharing information. A balance must be struck between the cost of having access to shared information when it is not being used, and the cost of accessing shared information when it is used. We believe the Butterfly architecture strikes a good balance.

The Butterfly provides a number of processor nodes, each containing a processor and local memory, which are connected via a high-speed omega (butterfly, fft) network. The memory management hardware determines whether or not an address references the local memory, or the memory on another node. When all processors are accessing local memory, there is no contention for memory resources, and the cost of the ability to share information is limited to a simple test in the address translation. This model has significantly lower cost for local accesses than those models which provide a single global memory for all processors. In addition, it scales to larger numbers of processors far more easily.

When access to shared data is needed, processes can place memory on remote nodes into their address space. This allows access to shared memory, at the cost of a higher memory cycle time. While this remote access is more costly than those architectures with a single global memory, the cost to access shared information is lower than message based architectures. The use of shared memory allows concurrent access to data structures which avoids the process interaction and synchronization costs that message based systems inherently force.

When transferring large amounts of data from one processor's memory to another's, message passing implementations may be more appropriate. This is because the extended cycle time of remote memory references and repeated instruction interpretation can become expensive. For such situations, the Butterfly architecture provides a hardware implemented block memory copy between processors. With this hardware, message based communication between processors can be efficiently implemented. In addition, with the shared memory capability discussed earlier, a message based communication does not require interrupting the destination processor.

In summary, the Butterfly architecture provides local memory for efficient access to information that is not shared, an interconnection network for efficient access to remote memories for small amounts of shared information, and block memory copy for efficient sharing of large amounts of information. Any single feature could doubtless be implemented at a lower cost, but it is the balance of these features which determines the effectiveness of a parallel architecture.

Programming Environment

We find the programming environment for the Butterfly is its weakest attribute. The Butterfly architecture allows efficient implementation of algorithms based on a variety of models of computation, unlike many machines where the model of computation is built into the machine. On the Butterfly, programmers are not locked into a model of computation, but may choose that which seems most natural to the problem at hand. Unfortunately, in adopting a model of computation, Butterfly programmers must choose a programming environment which supports that model. Currently, all programming environments support a single model of computation. These environments then have a severe, often restricting, impact on the algorithms chosen. In other words, the current programming environments do not reflect the flexibility of the architecture.

We chose to program using Chrysalis directly rather than using the Uniform System. Chrysalis is more suitable for the large processes communicating through relatively disjoint regions of shared memory. This characterizes our shared stack tree search algorithm. The Uniform System is suitable for those instances where one thinks in terms of shared matrices and "parallel for loops". This characterizes the types of computations needed for parallelizing the precomputations and tree node filters. Further efforts at parallelizing would probably involve re-coding the program under the Uniform System for the precomputations and node filters and then working the tree search in on top of that. This brings up the issue of mixing models within a single program. These issues are an active research area, and more work is needed in this area.

References

[BB85a] BBN Laboratories; *Chrysalis Programmer's Manual*; Version 2.2, June 1985

[BB85b] BBN Laboratories; *The Uniform System Approach to Programming the Butterfly Parallel Processor*; Version 1, October 1985

[BM76] J. A. Bondy and U. S. R. Murty; *Graph Theory and Applications*; North-Holland, 1976.

[Br86] C. Brown, R. Fowler, T. LeBlanc, M. Scott, M. Srinivas, L. Bukys, J. Costanzo, L. Crowl, P. Dibble, N. Gafter, B. Marsh, T. Olson, L. Sanchis; *DARPA Parallel Architecture Benchmark Study*; Prepared for the DARPA Architecture Workshop, November 1986; Butterfly Project Report 13; University of Rochester, Computer Science Department, Rochester, NY, 14627; October 1986

[GJ79] M. R. Garey and D. S. Johnson; *Computers and Intractability: A Guide to the Theory of NP-Completeness*; W. H. Freeman, 1979.

[Ti79] G. Tinhofer; *On the generation of random graphs with given properties and known distributions*; Applied Computer Science (Munich), 13 (1979), pp. 265-297.

[Ul76] J. R. Ullmann; *An Algorithm for Subgraph Isomorphism*; J. ACM, 23 (1976), pp. 31-42.