The Advantages of Multiple Parallelizations in Combinatorial Search*

Lawrence A. Crowl
Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

Mark E. Crovella Thomas J. LeBlanc Michael L. Scott
Computer Science Department
University of Rochester
Rochester, New York 14627-0226

September 19, 1994

Abstract

Applications typically have several potential sources of parallelism, and in choosing a particular parallelization, the programmer must balance the benefits of each source of parallelism with the corresponding overhead. The tradeoffs are often difficult to analyze, as they may depend on the hardware architecture, software environment, input data, and properties of the algorithm. An example of this dilemma occurs in a wide range of problems that involve processing trees, wherein processors can be assigned either to separate subtrees, or to parallelizing the work performed on individual tree nodes. We explore the complexity of the tradeoffs involved in this decision by considering alternative parallelizations of combinatorial search, examining the factors that determine the best-performing implementation for this important class of problems.

Using subgraph isomorphism as a representative search problem, we show how the density of the solution space, the number of solutions desired, the number of available processors, and the underlying architecture all affect the choice of an efficient parallelization. Our experiments, which span seven different shared-memory multiprocessors and a wide range of input graphs, indicate that relative performance depends on each of these factors. On some machines and for some inputs, a sequential depth-first search of the solution space, applying simple loop-level parallelism at each node in the search tree, performs best. On other machines or other inputs, parallel tree search performs best. In still other cases, a hybrid solution, containing both parallel tree search and loop parallelism, works best. We present a quantitative analysis that explains these results, and present experimental data culled from thousands of program executions that validates the analysis. From these experiences we conclude that there is no one "best" parallelization that suffices over a range of machines, inputs, and precise problem specifications. As a corollary, we provide quantitative evidence that programming environments and languages should not focus exclusively on flat data parallelism, since nested parallelism or hybrid forms of parallelism may be required for an efficient implementation of some applications.

*This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Mark Crovella is supported by a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.

1 Introduction

Most applications exhibit many different sources of parallelism that might be exploited by a given implementation. Examples include instruction-level parallelism within basic blocks, loop-level data parallelism (as implemented by many parallelizing compilers), nested instances of data parallelism, and thread-based parallel implementation of heterogeneous functions. The choice of parallelization for a given application can be very complex, since it involves balancing the costs and benefits of the different sources of parallelism in the context of a given system. Specific system characteristics that must be considered include the number of available processors, the overhead of process (thread) management, the cost of synchronization and communication, and the potential for load imbalance.

Unfortunately, parallel programming environments, languages, and machines often support only one kind of parallelism, and thereby bias the choice of parallelization at the outset. For example, vector processors can easily exploit the parallelism inherent in processing an array of data, but cannot exploit functional parallelism. Parallelizing compilers use dependency analysis to discover which loop iterations may safely execute in parallel, but do not in general focus on other sources of parallelism. Heavyweight processes implemented in the kernel of an operating system are appropriate only for coarse-grain parallelism, since both process creation and context switching tend to be prohibitively expensive.

The problem of choosing the proper parallelization occurs in a wide range of applications that involve processing trees. In these applications, processors can be assigned either to process separate subtrees, or they can be assigned to parallelizing the processing of individual tree nodes. In this paper we explore the complexity of the tradeoffs involved in this decision by considering alternative parallelizations of combinatorial search on several different shared-memory multiprocessors. We focus on the problem of finding *subgraph isomorphisms*, and present a quantitative analysis of the factors that determine the best-performing implementation for this important class of problems. We show that the choice between alternative parallelizations depends on the underlying architecture, the number of processors, the expected density of the solution space, and the number of solutions desired. On some machines and for some inputs, a sequential depth-first search of the solution space, applying simple loop-level parallelism at each node in the search tree, performs best. On other machines or other inputs, parallel tree search performs best. For certain combinations of parameters, we find that it pays to employ a hybrid approach that mixes parallelizations.

We conclude that there is no one "best" parallelization for this application that suffices for a range of machines and inputs, and therefore any portable implementation must encode multiple parallelizations. We also conclude that neither straightforward loop-level parallelism (as supported by most parallelizing compilers) nor thread-based functional parallelism (as supported by most lightweight thread packages) captures the full range of practical parallel algorithms, and that programming languages and systems with a heavy bias toward only one style of parallelization will be inadequate for this important class of problem on large-scale parallel machines.

We present the problem of subgraph isomorphism, and a parallel algorithm to solve it, in section 2. Although the algorithm has many sources of parallelism, we focus on two particularly important ones: simple loop-level parallelism over matrices at each node in a search tree, and parallel tree search. Using results culled from thousands of data points on seven different shared-memory machines, we argue the need for multiple parallelizations in section 3. We present the case for hybrid parallelizations that combine loop-level parallelism and parallel tree search in section 4. We summarize our conclusions in section 5.

2 Problem Description and Analysis

We will use *subgraph isomorphism* as an example problem requiring combinatorial search. Given two graphs, one small and one large, the problem is to find one or more isomorphisms from the small graph to arbitrary subgraphs of the large graph. An isomorphism is a mapping from each vertex in the small graph to a unique vertex in the large graph, such that if two vertices are connected by an edge in the small graph, then their corresponding vertices in the large graph are also connected by an edge. Though subgraph isomorphism is NP-complete, techniques for pruning the search space often allow solutions to be found in a reasonable amount of time.

2.1 An Algorithm for Finding Isomorphisms

Our algorithm is based on Ullman's sequential tree-search algorithm [Ullman, 1976]. This algorithm first postulates a mapping from one particular vertex in the small graph to a vertex in the large graph. This mapping constrains the possible mappings for other vertices of the small graph: they must map to distinct vertices in the large graph, and must have the same relationship to the first vertex in both graphs. (This notion of relationship is made more precise below.) The algorithm then postulates a mapping for a second vertex in the small graph, again constraining the possible mappings for the remaining vertices of the small graph. This process continues until an isomorphism is found, or until the constraints preclude such a mapping, at which point the algorithm postulates a different mapping for an earlier vertex. The search for isomorphisms takes the form of a tree, where nodes at level i correspond to a single postulated mapping for vertices 1 through i in the small graph and a *set* of possible mappings for each vertex $j > i$, and where the mappings at levels 1 through i constrain the possible mappings at levels $j > i$.

We illustrate this algorithm with a simple example. The small and large graphs are:

$$a \rightarrow b \rightarrow c \quad \text{and} \quad 1 \leftarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$$

Each node in the search tree is a *partial isomorphism*, which we represent by an $S \times L$ Boolean matrix, where S is the number of vertices in the small graph, L is the number of vertices in the large graph, and entry (i, j) is true if we are still considering the possibility of mapping vertex i of the small graph to vertex j of the large graph. Each row represents a set of possible mappings for vertex i. For example, the initial partial isomorphism for our sample problem is:

$$a : \{1,1,1,1,1\},\ b : \{1,1,1,1,1\},\ c : \{1,1,1,1,1\}$$

When all rows in the partial isomorphism contain exactly one true element, then each vertex in the small graph has a single postulated mapping and the isomorphism is complete. When any row in the partial isomorphism contains no true elements, then some vertex in the small graph has no acceptable mapping, the isomorphism is invalid, and we may prune that node from the search tree.

The children of a node are constructed by selecting one possible mapping at the next level of the tree and then removing any conflicting mappings. For example, we may choose to map vertex a to vertex 2. Since vertex a may map to only one vertex in the large graph, we remove all other mappings for vertex a. This yields the partial isomorphism:

$$a : \{0,1,0,0,0\},\ b : \{1,1,1,1,1\},\ c : \{1,1,1,1,1\}$$

In addition, no two vertices in the small graph may map to the same vertex in the large graph, so we remove vertex 2 from the possible mappings of all other small graph vertices. This yields the partial isomorphism:

$$a : \{0,1,0,0,0\},\ b : \{1,0,1,1,1\},\ c : \{1,0,1,1,1\}$$

Since the search space is very large, it is prudent to eliminate possible mappings early, before they are postulated in the search. We do this by applying a set of *filters* to the partial isomorphisms, reducing the number of elements in each mapping set, and pruning nodes in the search tree before they are visited. Though there is a large number of possible filters, each based on a relationship between vertices within the small graph and between vertices and their potential mappings, we apply only two filters, *vertex distance* and *vertex connectivity*. These filters prune the search space enough to make the problem tractable.

We define the distance from one vertex to another to be the minimum number of edges in a directed path from the first vertex to the second. If there exists no path between the two vertices, the distance is infinite. The distance filter uses precomputed distance matrices for the two input graphs, which in our example are:

	a	b	c
a	∞	1	2
b	∞	∞	1
c	∞	∞	∞

and

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	1	2	3
3	∞	∞	∞	1	2
4	∞	∞	∞	∞	1
5	∞	∞	∞	∞	∞

The vertex distance filter eliminates mappings where the distance between two vertices in the small graph is less than the distance between two vertices in the large graph, which implies that there is some edge in the small graph that is not represented in the isomorphism. Formally, the filter at node (i, j) in the search tree (that is, the node at which we postulate a mapping from vertex i in the small graph to vertex j in the large graph) removes entry (k, l) of the matrix associated with that node if the distance from i to k is less than the distance from j to l. In our example, the distance from vertex a to vertex b is 1, which is less than the distance from vertex 2 to vertex 4, and so we can remove $(b, 4)$ from the partial isomorphism. Likewise, we can remove $(b, 5)$ and $(c, 5)$. This yields the partial isomorphism:

$$a : \{0, 1, 0, 0, 0\},\ b : \{1, 0, 1, 0, 0\},\ c : \{1, 0, 1, 1, 0\}$$

The vertex connectivity filter ensures that the possible mappings of a vertex in the small graph are consistent with the possible mappings of its neighbors.[1] Formally, the filter at node (i, j) of the search tree removes entry (k, l) of the matrix when there is an edge from i to k, but not an edge from j to l. The result of this elimination is simply the intersection of the mappings for k and the neighbors of j. In our example, $i = a$ and $j = 2$; $k = b$ is a neighbor of $i = a$; the neighbors of $j = 2$ are $\{1, 3\}$; the current mappings for $k = b$ are $\{1, 3\}$; and their intersection is $\{1, 3\}$. The partial isomorphism is therefore unchanged by this filter in this case.

Since neither of the above filters eliminates *all* invalid isomorphisms, we perform a final verification at each leaf node to ensure that every edge in the small graph is represented by an edge in the large graph, and therefore represents a valid isomorphism.

2.2 Sources of Parallelism

There are many ways to exploit parallelism in the implementation of our algorithm for subgraph isomorphism. (See Table 1.) The coarsest granularity of parallelism occurs in the tree search itself; we can search each subtree of the root node in parallel (hereafter referred to as *tree parallelism*),

[1] Our vertex connectivity filter resembles the one employed by Ullman, except that we do not iterate after eliminating a possible mapping to see whether its elimination would allow us to eliminate additional mappings.

with depth-first, sequential search at the remaining levels.[2] At each node of the tree, several filters must be applied so as to prune the search tree, and this set of filters could be executed in parallel.[3] We can also exploit parallelism when applying a filter to a candidate mapping.

Label	Source of Parallelism
Tree	search children of the root node in parallel
Filter	apply multiple filters at a node in parallel
Loop	examine "relatives" of the postulated mapping in parallel
Vector	constrain mappings for each "relative" in parallel
Word	constrain mappings for each "relative" in parallel

Table 1: Parallelism in subgraph isomorphism, from coarsest to finest.

Each filter removes potential mappings based on some relationship between the candidate vertex in the small graph and other vertices in the small graph. For a given candidate vertex, we can examine constraints on the remaining vertices of the small graph in parallel. We refer to this source of parallelism as *loop parallelism*, since both the granularity and structure of this source of parallelism resembles a single parallel loop. A finer-grain source of parallelism arises when removing mappings that violate the constraints of a filter. Since the primitive data elements in a mapping are of type Boolean, we can pack many booleans into a single word and use word-parallel bit operations, such as **and** and **or**. We refer to this source of parallelism as *word parallelism*. In addition, multiple word-parallel operations could be performed in parallel by a vector processor, thereby exploiting *vector parallelism*. Although both vector and word parallelism exploit the same source of parallelism (the possible mappings of a given small vertex), they can be used individually or in tandem.

In this paper, we focus on the tradeoffs between tree parallelism and loop parallelism. These two kinds of parallelism are particularly important because they occur in many problems that have a similar structure: expression evaluation [Eager and Zahorjan, 1993], parallel quicksort [Chen *et al.*, 1984], and the Barnes-Hut multibody algorithm [Singh *et al.*, 1992]. Although the presence of both tree parallelism and loop parallelism in backtracking search has been noted by other researchers [Wah *et al.*, 1985; Natarajan, 1987; Finkel and Manber, 1987; Rao and Kumar, 1989], there has been little previous attention given to the tradeoffs between loop and tree parallelism as a function of the particular machine, input, or problem. Since there are many possible parallelizations of the subgraph isomorphism algorithm, resulting from varying processor allocations to loop and tree parallelism, it is difficult to choose a particular parallelization without a better idea of how the various problem parameters affect the choice. Indeed, in an earlier study of subgraph isomorphism [Costanzo *et al.*, 1986] we chose to exploit tree parallelism because of its lower synchronization costs, even though (in retrospect) the problem parameters were such that tree parallelism was not very effective.

We study the tradeoff between loop and tree parallelism experimentally in Section 3, in which we show how the best choice of parallelization depends on the precise problem to be solved (that is, the number of isomorphisms to be found), the structure of the input graphs (i.e., the density of

[2] We could choose to implement tree parallelism at *any* depth in the tree, rather than at the root. A comparative evaluation of the alternative implementations of tree parallelism is beyond the scope of this paper, however.

[3] Our implementation uses only two filters, but others are possible. In general, we would expect combinatorial search problems to employ many filters, so much so that this source of parallelism could prove extremely valuable. However, in the experiments reported in this paper, we do not exploit this source of parallelism.

the solution space), the number of processors available, and the underlying architecture. In order to lay the foundation for that discussion, the next three subsections analyze the potential costs and benefits of the two parallelizations in the context of our implementation.

2.3 The Benefits of Speculative Search

When searching for only one solution, tree parallelism is *speculative*, in that we might not need to search every subtree of the root in order to find the required number of solutions. If the solution space is sparse, so that only some subtrees of the root contain solutions (as in Figure 1), then we might choose to search several subtrees in parallel rather than apply loop parallelism during a sequential depth-first search of a subtree that might contain no solutions. On the other hand, when searching in a very dense solution space (as in Figure 2), we can reasonably expect every subtree of the root to contain many solutions, and therefore could expect better results by using loop parallelism in an efficient depth-first search under a single child of the root.

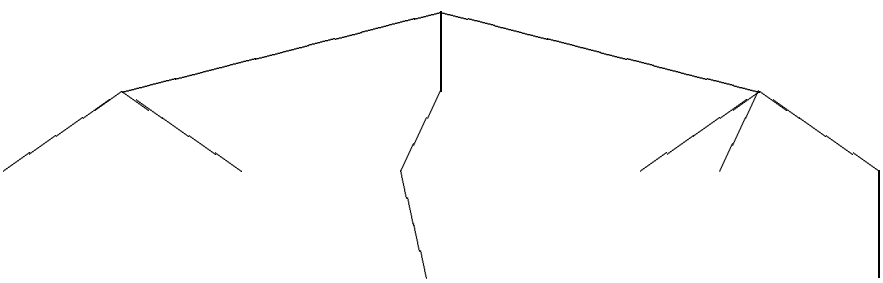

Figure 1: A sparse search tree.

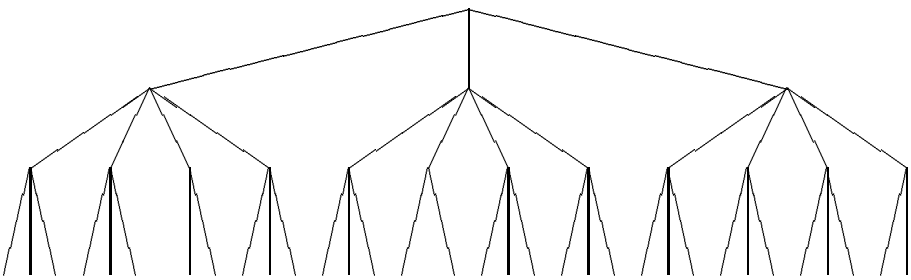

Figure 2: A dense search tree.

Even when loop parallelism is more effective than tree parallelism for a given input when searching for a single isomorphism, the reverse may be true when searching for multiple isomorphisms. This situation could occur because even a successful depth-first search of a single subtree might not yield enough solutions. The choice depends on the number of solutions desired, and the extent to which solutions are clumped in the search tree. In some cases, a combination of tree and loop parallelism might provide the best performance, assuming we have enough processors to implement loop parallelism within the context of tree parallelism.

Thus, the potential benefits of speculation are highly dependent on the nature of the search space. The remainder of this section describes the search space of our sample problem, and provides the necessary background to evaluate the potential benefits of speculation as we vary the input data.

We characterize the search problem in terms of the number of isomorphisms we want to find and the structure of the two input graphs. In our experiments input graphs are randomly generated from four parameters: the size of the small and large graphs, and the probability for each graph that a given pair of vertices will be joined by an edge. We use S for the number of vertices in the small graph, L for the number of vertices in the large graph, and s and l, respectively, for the edge probabilities in each of the graphs. There are S^2 potential edges in the small graph (the graphs are directed, and self-loops are permitted). The expected degree of a vertex in the small graph is $(2S-1)s$; the expected degree of a vertex in the large graph is $(2L-1)l$.

As described above, the problem of finding isomorphisms can be reduced to searching a very bushy S-level tree of possible vertex matchings. In this bushy (unpruned) search tree, the L children of the root represent the L associations of vertex 1 of the small graph with a vertex of the large graph. The $L-1$ children of a level-1 vertex represent the $L-1$ associations of vertex 2 of the small graph with one of the remaining vertices of the larger graph, and so on. The total number of leaves in the problem space tree is

$$\binom{L}{S} S!$$

This number is very large; for $S = 32$ and $L = 128$ (the size of our experiments), there are approximately 4×10^{65} leaves. Brute-force search of this space is not feasible.

We define the *density* d of the search space to be the probability that a randomly-selected leaf will be a solution. We can express this as

$$d \equiv (1 - s + sl)^{S^2},$$

where S^2 is the number of potential edges in the small graph and $1 - s + sl$ is the probability that a given potential edge will be successfully matched—that it will either be missing in the small graph, or present in both graphs. We assume that edge occurrences are independent events in both graphs. In the experiments reported here, we use this definition of density to characterize the input graphs.

The search algorithm employs loop parallelism to speed up the search along a single path through the tree, and tree parallelism to search alternative paths. Which approach can be expected to be most productive depends in large part on the variance in the time required to find a solution on different paths. The higher the variance, the more likely that tree-parallel search of multiple paths will turn up a solution sooner than loop-parallel search of a single path.

From a qualitative point of view, this expectation is consistent with the findings of Rao and Kumar [Rao and Kumar, 1993], who used the notion of solution density to describe the situations under which tree-parallel search exhibits superlinear speedup in comparison to sequential search. Their primary finding was that the expected speedup of tree-parallel search is linear at worst, and superlinear when the portions of the tree searched by different processors have different solution densities, or when heuristic ordering of child nodes is wrong near the root of the tree. Since our loop parallelism can display no more than linear speedup, these are the situations in which one would expect tree parallelism to dominate.

Unfortunately, the quantitative aspects of Rao and Kumar's work do not apply directly to our work. They consider two subcases. The first case assumes that the algorithm does not prune, and that within the portion of the tree searched by a single processor, the probability of a given leaf being a solution is independent of the probabilities of any other leaves being solutions. The second

case models both pruning and heuristic ordering of children at each node of the tree. It assumes that the tree is binary, and that the probability of pruning is uniform throughout the tree. None of these assumptions apply to the algorithm employed in our experiments.

Precisely analyzing the tradeoff between loop and tree parallelism requires characterizing the behavior of the filters, which is quite difficult. However, we can obtain an intuitive understanding of the tradeoff between loop and tree parallelism by considering the probability that solutions lie in various subtrees of the search space.

Define a *successful* node to be one that 1) is the root node or the child of a successful node, and 2) whose mapping of a vertex in the small graph to a vertex in the large graph introduces no unsuccessful edges. A successful edge is one which connects nodes in the small graph that have allowable mappings (given the mappings postulated by the current search) that are also connected. If arbitrary children of successful nodes at most levels of the tree are likely to have a solution under them, then little backtracking can be expected to occur, and loop-parallel exploration of a single path is likely to find a solution faster than tree-parallel exploration of multiple paths. If, on the other hand, at most levels of the tree an arbitrarily-chosen child is unlikely to have a solution beneath it, then significant amounts of backtracking are likely, the variance in the solution-finding times of different paths is likely to be high, and tree parallelism is likely to work well.

Thus, we would like to estimate the probability that children of a given successful node have at least one solution under them. Assume the root is level 0. At a successful node at level k of the tree, we will have successfully matched k nodes and k^2 potential edges in the small graph with counterparts in the large graph. $S-k$ nodes and S^2-k^2 potential edges will remain. The probability that a randomly-chosen leaf below this point will turn out to be a solution is therefore $t^{S^2-k^2}$, where $t = 1-s+sl$. We cannot use this directly to answer our question, because solutions are not independent. In fact, solutions tend to come in clumps, since nearby leaves share many successfully-matched ancestors; as a result, the probability of finding *at least one* solution in a given subtree is much lower than it would be if solutions were evenly distributed. Thus we must consider the structure of a subtree in determining the likelihood that it contains a solution.

Suppose that $\mathbf{v}$ is a leaf whose parent is successful. The probability that $\mathbf{v}$ is a solution is $p_S = t^{2S-1}$. ($2S-1$ is the number of potential edges induced by matching the last vertex.) Now suppose that $\mathbf{v}'$ is a node at level $S-1$ whose parent is successful. The probability that $\mathbf{v}'$ itself will be successful is $t^{2(S-2)+1}$. Assuming it is successful, the probability that a given child of $\mathbf{v}'$ (a leaf) is *not* a solution is $1-t^{2S-1}$. Then the probability that *no* leaf under $\mathbf{v}'$ is a solution is $(1-t^{2S-1})^{L-S+1}$, since $\mathbf{v}'$ has $L-S+1$ children. Finally, the probability that at least one leaf under $\mathbf{v}'$ is a solution is $p_{S-1} = t^{2(S-2)+1}(1-(1-p_S)^{L-S+1})$.

In general, if we assume that the children of a successful node have independent probabilities of being successful (this is only an approximation above the final level), we have

$$p_k \approx t^{2(k-1)+1}(1-(1-p_{k+1})^{L-k}).$$

On the assumption that nodes are pruned iff they are not successful, we can also compute the expected number of nodes that will be visited in the process of searching for a solution in various subtrees. This number turns out not to be useful, however, because the filters eliminate the vast majority of nodes before they are ever visited. For example, when $S=32$, $L=128$, $s=0.9$, and $l=0.2$ ($d \approx 10^{-567}$), our implementation visits only 105 nodes of the more than 10^{65} nodes in the search tree, in the process of determining that no isomorphisms exist. A search that pruned only when nodes were unsuccessful would be expected to visit nearly 20,000 nodes.

Figure 3 plots p_k against k for various values of d. At densities greater than about 10^{-6}, the search has better than even odds of finding a solution under an arbitrarily-chosen child all the way

down to the leaves. Under these circumstances, tree-parallel search of multiple paths in the tree is unlikely to uncover a solution much faster than sequential search on a single path, and probably much slower than loop-parallel search on that path. By contrast, at densities lower than about 10^{-48}, the odds are worse than 50-50 only 5 levels down in the tree. Extensive backtracking is likely, and the variance in the amount of time required to find a solution on different paths down the tree is likely to be high. Under these circumstances, tree-parallel search is probably a very good idea.

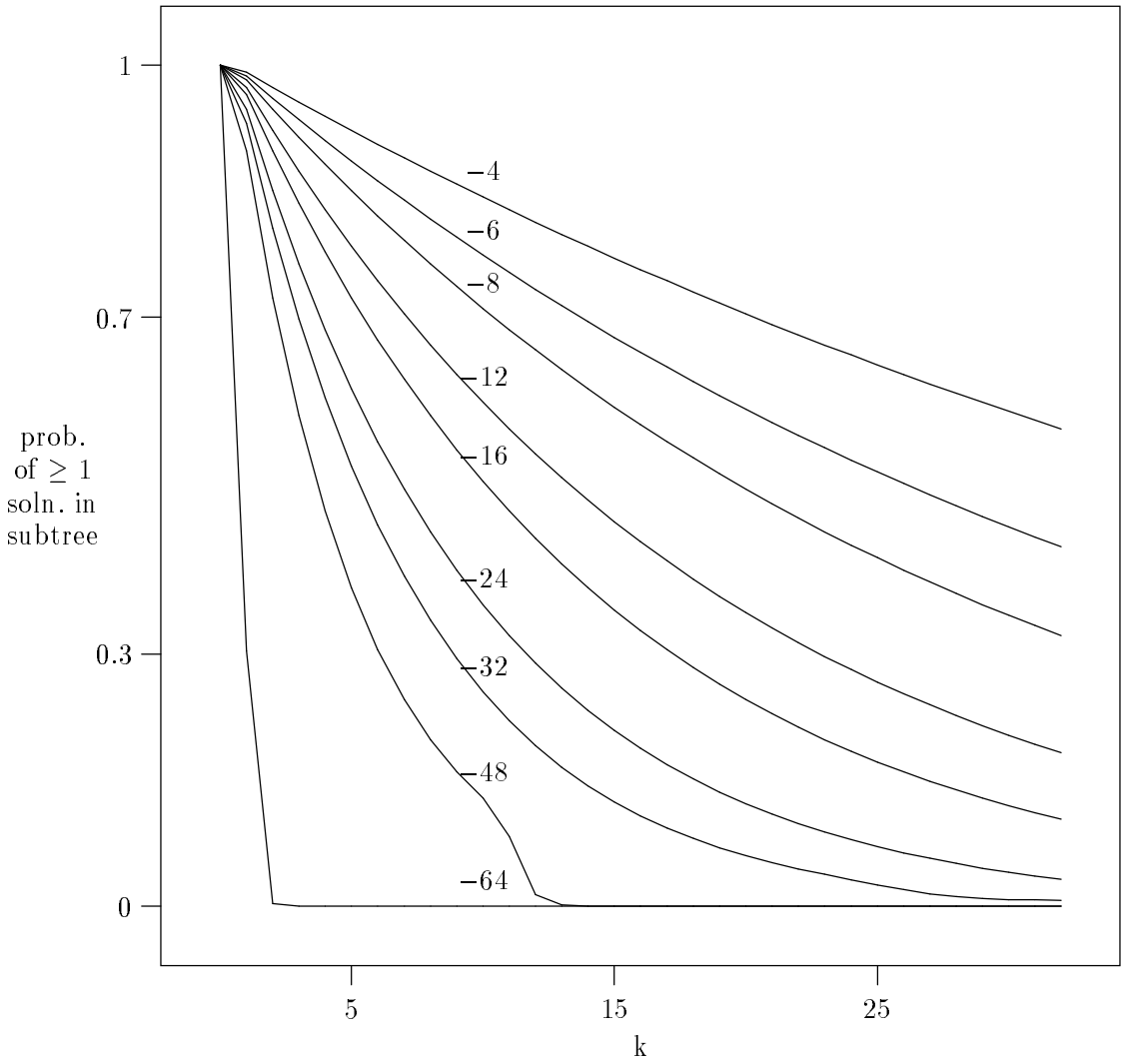

Figure 3: Probability that a randomly-selected node at level k has at least one solution under it, given that its parent is successful, for various logs (base ten) of the solution space density.

2.4 Implementing Tree and Loop Parallelism

To make our discussion of the remaining performance effects concrete, we first describe our implementation of tree and loop parallelism in subgraph isomorphism. Subsequently, in section 2.5, we describe the significant performance effects other than speculation that relate to the tradeoff

between loop and tree parallelism.

2.4.1 Tree Parallelism

In our experiments, the small graph has 32 nodes, and the large graph has 128 nodes. The root of the search tree therefore has 128 children (the number of possible mappings between a particular vertex in the small graph to a vertex in the large graph). To implement tree parallelism, we place the 128 partial isomorphisms of the root node in a central work queue, and create a single worker process on each processor. Each worker process has a local copy of the source graphs and the distance matrices. During its lifetime a worker process removes an entry from the work queue, searches that subtree of the root until a sufficient number of solutions are found or the subtree is exhausted, and then takes another partial isomorphism from the work queue. The work performed by one worker process is entirely independent of every other worker process, except that all processes terminate when the desired number of solutions has been found.

We used a central work queue so as to balance the load among processors; the high variation in search time among subtrees would likely cause load imbalance under any static distribution of work. We did not bother to distribute the work queue, since access to the shared queue is very infrequent. We maintain the number of solutions found so far in a shared variable, which each worker process examines after processing a node in the search tree; we could increase the amount of *braking loss* (that is, time spent continuing to search for solutions even after all required solutions have been found) and decrease contention for this shared variable by examining its value less frequently. Although each of these design choices has slightly different costs on the different machines used in our study, the overall effect of the underlying architecture on these design decisions is not significant.

The machines used in our experiments all had fewer than 128 processors, and most program executions terminated without any process retrieving several entries from the work queue. As a result, no load imbalance occurred using tree parallelism. However, on machines with a larger number of processors, it would be possible for processors to be idle due to an absence of work in the work queue. In such cases, we could place second-level (or even third-level) children of the root in the work queue, so as to create enough parallelism to exploit all available processors.

2.4.2 Loop Parallelism

Loop parallelism occurs in the implementation of the filters that are applied at each node in the search tree. Each filter iterates over the vertices of the small graph in parallel, so loop parallelism offers at most 32-way parallelism in our experiments. To implement this parallelism, we create a single worker process on each processor that is responsible for executing iterations of parallel loops. A single master process is responsible for copying the loop parameters into the local memory of each worker process, and for synchronizing all of the worker processes at the end of loop execution. At the start of loop execution, we divide the n iterations of the loop into blocks of size n/p and assign each block to a processor. This assignment of iterations to processors ensures that a processor always executes the same loop iterations for every parallel loop. As described in [Markatos and LeBlanc, 1992], this technique avoids the need to load the rows of the partial isomorphism matrix into local memory at the start of loop execution, since the i_{th} iteration of every parallel loop is always executed by the processor whose local memory or cache already contains the i_{th} row of the matrix.

There can be significant load imbalance among loop iterations because the amount of work performed by a filter depends on the number of potential isomorphisms in a partial isomorphism;

iterations with many active mappings do more work than iterations with few active mappings. In addition, an iteration of the connectivity filter, which checks for consistency across postulated mappings, terminates if the corresponding vertex in the small graph does not yet have a postulated mapping. Thus, the variation in running time among iterations is high. When using a small number of processors, we assign several iterations to each processor, and the imbalance among iterations tends to average out. However, as the number of processors increases, the number of iterations per processor decreases, and the imbalance among iterations becomes significant. As discussed below, dynamic scheduling of loop iterations cannot eliminate this imbalance and therefore we use static scheduling. We verified this choice with experiments on a Silicon Graphics multiprocessor workstation, which showed that our static distribution of blocks of iterations performed as well as or better than dynamic self-scheduling of loops.

2.4.3 Combining Tree and Loop Parallelism

Some of our experiments use both tree and loop parallelism simultaneously. To implement this hybrid form of parallelism we statically divide the available processors into two categories: processors used for tree search and processors used for loop parallelism. As before, we create a worker process for tree search on each of the processors dedicated to tree search; we create a worker processor for parallel loops on the remaining processors. The implementations of tree and loop parallelism within each group of processors is the same as described above.

2.5 Evaluating Performance

In this section we identify the sources of overhead in tree and loop parallelism and describe our methodology for measuring these overheads during program execution.

2.5.1 Sources of Overhead

The primary source of overhead in the implementation based on tree parallelism is wasted speculation. Section 2.3 provides a quantitative analysis of the benefit of speculative search, and an assessment of how that benefit varies with the density of the solution space. This analysis gives insight into the potential benefits of speculation as we increase the number of processors, and is particularly appropriate for a large number of processors. For a small number of processors (on the order of $2-4$), the benefit of speculation is uncertain. We have observed that even in sparse solution spaces, small numbers of processors do not usually exploit speculation effectively.

There are two primary sources of overhead under loop parallelism: load imbalance and communication (and synchronization) costs. Load imbalance is difficult to avoid: the nature of the algorithm dictates that loop parallelism is used within a large number of small loops that have high variation in iteration running time. Since the loops are small, any load imbalance is large in comparison to the work done. Alternative loop scheduling schemes cannot significantly improve load balance — there are too few iterations available to allow load balancing to be successful.

Scaling the size of the problem to decrease load imbalance is not likely to be successful either. The number of iterations in the loops is equal to the number of nodes in the small or large graph, and the problem difficulty increases exponentially in the sizes of these graphs. The problem would become intractable well before the loops could be made large enough to eliminate load imbalance. As a result, we conclude that loop parallelism in subgraph isomorphism is inherently not scalable.

Communication costs are present under both forms of parallelism, but are not significant under tree parallelism since each task proceeds independently and no inter-task communication is necessary. Communication costs are significant under loop parallelism however, since even very small

loops require extensive coordination between the master and worker processes. As a result, the cost of communication in the underlying machine has a noticeable effect on the performance of loop parallelism, but has little effect on the performance of tree parallelism.

In summary, speculation is increasingly beneficial as the number of processors increases, and as the solution space grows more sparse. On small numbers of processors, speculation gives unreliable benefits, while loop parallelism provides fairly predictable benefits. However, loop parallelism suffers from load imbalance and communication overhead, which increase with an increase in the cost of communication.

2.5.2 Measuring Overheads

To explore the relationship between problem parameters and the performance of different parallelizations, we measured the causes of poor performance in each implementation of subgraph isomorphism. Our goal was to gain understanding of the way that machine characteristics, problem definition, and input choice affect the various kinds of overhead that can occur in parallel combinatorial search.

To help develop insight, we assigned the various overhead costs to categories that are meaningful to the programmer. The particular categories were chosen so as to be *complete* and *orthogonal*. By completeness we mean that in the absence of overheads in these categories, the program would have exhibited linear speedup. This is verified empirically: if, after measuring all overhead in a multiprocessor execution, the remaining computation equals that of the uniprocessor case, then the set is complete for that execution. Our set was found to be complete for all executions we measured. By *orthogonal* we mean that no segment of a single processor's time can be simultaneously assigned to two different overhead categories. This ensures that we can measure, add, and subtract overhead values meaningfully.

The categories we used are: *Load Imbalance, Idling, Synchronization Loss, Memory Loss* and *Wasted Computation*. *Load Imbalance* is defined as the processor cycles spent idling, while parallel tasks exist and are not yet completed. *Idling* is defined as the processor cycles spent idling, while there are no parallel tasks available. *Synchronization Loss* is defined as the time spent executing synchronization instructions (e.g., waiting in a barrier or spinning on a lock). *Memory Loss* is defined as time processors spend stalled, waiting for memory to supply needed operands. Finally, *Wasted Computation* is defined as algorithmic work done by the program that did not contribute to finding the problem solution(s). In combinatorial search, this category measures time spent searching subtrees that do not contain solutions. In this case, we refer to this category as *wasted speculation*.

We developed a uniform method for quantitatively evaluating each of these overhead categories, and applied it to our implementations of subgraph isomorphism. This general approach to performance evaluation, along with its implementation, is described in more detail in [Crovella and LeBlanc, 1993]. We found that these categories provided the right level of abstraction for examining how performance depends on the underlying architecture, the structure of the input graphs, and the number of desired isomorphisms, on a range of shared-memory multiprocessors. The next section presents the results of those examinations.

3 Multiple Parallelizations

In this section we show that good performance in the subgraph isomorphism computation sometimes requires that tree search be parallelizable, and sometimes requires that filters be parallelizable. (Section 4 discusses situations for which parallelizing both search and filters is useful.) As a result,

good performance requires the ability to parallelize the problem using both loop parallelism and tree parallelism.

This result holds no matter which aspect of the problem is varied. Both parallelizations are needed whether one is concerned with:

1. porting a given problem and input to a different machine,
2. running a given problem on a given machine while varying inputs, or
3. for a fixed input and machine, searching for a varying number of solutions.

We show multiple examples for each of these points based on our implementation, which currently runs on seven shared-memory multiprocessors. These machines are as described in Section 3.1. We study four input data sets: one in which solutions are extremely plentiful; two in which solutions exist but are relatively rare (among all leaf nodes); and one in which no solutions exist. The problems we consider are: finding a single isomorphism; finding 128 isomorphisms; and finding 256 isomorphisms. We have collected data for these seven machines, four classes of inputs, three problems, and several parallelizations, amounting to several thousand data points.

In this presentation, we will not address variability in performance due to the number of processors used on a given machine or the choice of whether or not to exploit word parallelism. We report the minimum execution time (in seconds) achieved over the entire range of processors whether exploiting word parallelism or not.

3.1 Machines Used

Our implementation runs on seven shared-memory multiprocessors, covering a range of processor and interconnect technologies. Table 2 summarizes them.

Label	Number of Processors	Machine
Balance	20	Sequent Balance
Symmetry	19	Sequent Symmetry
Iris	8	Silicon Graphics Iris
Butterfly	39	BBN Butterfly One
TC2000	21	BBN Butterfly TC2000
8CE	7	IBM 8CE
KSR1	32	Kendall Square Research 1

Table 2: Shared-memory multiprocessors.

Three of the machines support a global coherent memory accessed through a shared bus. The Sequent Balance uses the National Semiconductor 32032 processor. The bus in the Balance has a sustained bandwidth of 26.7 MB/sec. Each processor has an 8 KB write-through cache. The Sequent Symmetry uses Intel 80386 processors running at 20 MHz, which are between 3–7 times faster than the processors in the Balance. The bus has a sustained bandwidth of 53.3 MB/sec, and each processor has a 64 KB write-back cache. The Silicon Graphics Iris uses 40 MHz MIPS R3000 processors on a bus with 64 MB/sec bandwidth. Each of the processors has a 64 KB first-level cache and a 1 MB second-level cache.

Two of the machines are from the BBN Butterfly family of scalable multiprocessors. The BBN Butterfly I uses 8 MHz Motorola 68000 processors connected by a 4 MB/sec (per link) Butterfly switch. Each processor has 1 MB of local memory, and a processor may access another's memory through the switching network; there are no data caches. The access time ratio between local and non-local memory is 1:5. The BBN TC2000 uses Motorola 88100 processors, which are about 60 times faster than the processors used in the Butterfly I. The peak switch bandwidth in the TC2000 is 38 MB/sec (per link). Although the TC2000 does have processor data caches, these caches are not kept consistent automatically, and therefore are not used for shared data. In our experiments on the Butterfly I and the TC2000, data is moved into local memory explicitly by the program.

The IBM 8CE is similar to the Butterfly family in that each processor has its own local memory, and can access the memory of every other processor. In addition, the 8CE has a single global shared memory. As in the Butterfly, there is no data cache, and the programmer (or operating system) must manage local and global memory explicitly. Access to non-local memory is through a shared bus, and the access time ratio between local, global and remote memory is 1:2:5. The 8CE uses the ROMP-C processor (as found in the IBM PC/RT).

The Kendall Square KSR 1 uses a custom 64-bit processor, which is roughly twice as fast as the processors used in the TC2000. All memory in the system is managed as a set of caches, with each processor containing a 32 MB cache (i.e., local memory) and a 512 KB subcache. Access times to the subcache, the local cache, and a remote cache are in the ratio 1:9:88. Cache block movement uses a high-speed ring network.[4]

3.2 Varying the Machine

For a given problem and input, each parallelization outperforms the other on some machine(s). Here we show two selected examples. In each example, loop parallelization is best for more than one machine, and tree parallelization is best for more than one machine. This shows that the differences we are noting are significant; there are multiple machines in each category.

The first example in Table 3 is searching for multiple solutions in a sparse solution space. (Specifically, looking for 128 isomorphisms in a solution space with density 10^{-21}). We see that loop parallelism is best on two machines, tree parallelism is best on four other machines, and one machine (the Balance) is a toss-up. For this and subsequent tables, we underline the time taken by the better parallelization in those cases where the difference in execution time is significant.[5]

		8CE	Butterfly	Balance	Iris	Symmetry	TC2000	KSR1
sparse	loop	24.6673	75.7317	91.6722	2.0559	29.7714	11.0429	10.68
	tree	36.0518	12.5988	86.7278	2.5880	15.8381	3.7843	2.24
dense	loop	1.0644	4.0369	4.2056	0.0933	1.3143	0.5518	0.46
	tree	1.5251	2.9804	6.5167	0.1087	1.6690	0.5177	0.26

Table 3: Time in seconds when searching for 128 solutions; varying machines.

The second example in Table 3 is searching for multiple solutions in a dense solution space. (That is, looking for 128 isomorphisms with solution space density 10^{-5}). In this case, there are

[4]The KSR 1 has a multi-level ring architecture, but all our tests were done on a single ring.

[5]We consider differences in running time significant if the slower version takes at least 25% more time than the faster version.

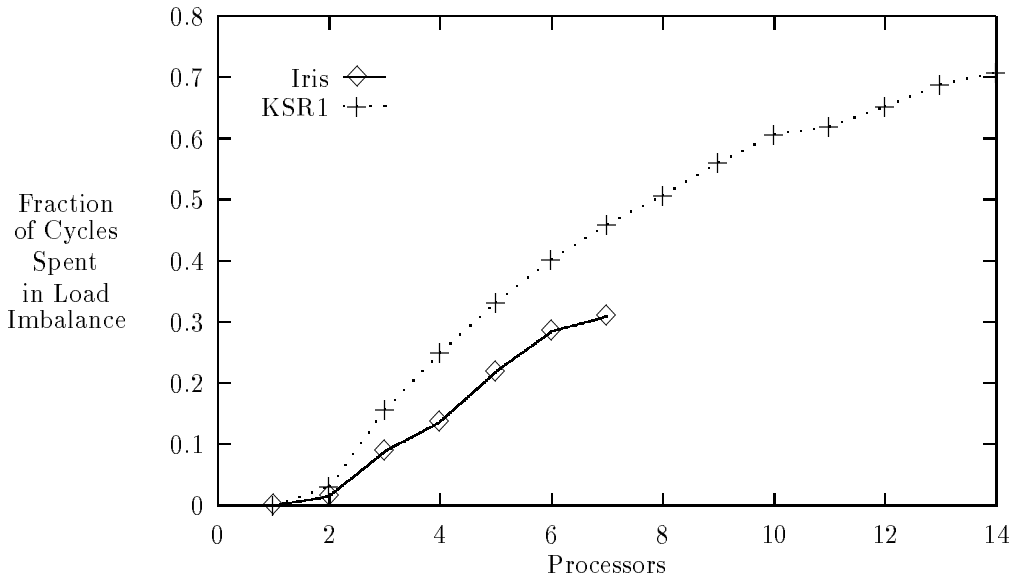

Figure 4: Increasing load imbalance in loop parallelism.

three machines for which loop parallelism is best, and two for which tree parallelism is best. The two parallelizations are a toss-up on the remaining two machines.

The first two lines in Table 3 show that in a sparse solution space, loop parallelism outperforms tree parallelism on the Iris and 8CE, while tree parallelism outperforms loop parallelism on the KSR1, TC2000, Symmetry, and Butterfly; the two parallelizations are comparable (within 6%) on the Balance. This result is somewhat surprising, given that the Balance, Symmetry, and Iris have such similar architectures (all are coherent, bus-based machines). We can determine why one parallelization outperforms the other by using our performance evaluation method, combined with considerations of inherent scalability. To focus our discussion, we concentrate on the Iris and the KSR1 as examples of architectures that favor one parallelization over the other.

Loop parallelism executes faster on the Iris than on the KSR1, while tree parallelism executes faster on the KSR1 than on the Iris. As discussed in Section 2.5, the principal issues in comparing these two parallelizations are the overhead of load imbalance and the benefit of speculation. These factors interact with the number of processors available and the speed of those processors.

To understand why the Iris outperforms the KSR1 under loop parallelism, we first note that the uniprocessor (sequential) running time of the program is 21.88 seconds on the KSR1, while it is 8.66 seconds on the Iris. Although the Iris is faster at solving this problem on a single processor, the Iris only has 8 processors, while our KSR1 configuration has 32 processors (much larger machines are available). Unfortunately our measurements of load imbalance show that for this problem, on these machines, the degree of load imbalance under loop parallelism grows quite large with an increase in the number of processors. Figure 4 shows the fraction of total processor cycles lost due to load imbalance for loop parallelism on this problem on both machines. The figure indicates that beyond about 8 processors, the fraction of cycles lost due to load imbalance grows very large. In fact, the benefit of adding additional processors beyond this point is completely counteracted by the increase in load imbalance, precluding the KSR1 from benefiting from its larger supply of processors.

On the other hand, the KSR1 outperforms the Iris under tree parallelism. As before, the single processor case favors the Iris (7.03 seconds on the Iris, 18.86 on the KSR1). However, there is no

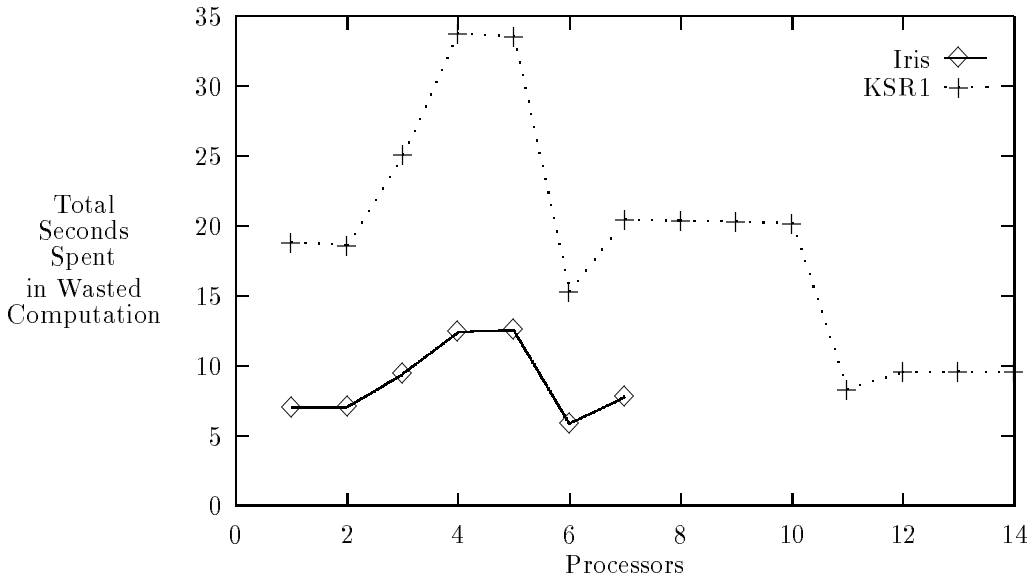

Figure 5: Decreasing wasted computation in tree parallelism.

load imbalance under tree parallelism on this problem; the dominant source of overhead is wasted computation due to speculation. Figure 5 shows the total time spent on wasted speculation for this problem on both machines, in seconds. As the number of processors increases, each time the line does not rise, the program has benefited from an increase in processing power; when the line stays flat, a constant amount of work has been divided among a larger number of processors; and when the line drops, the program has found a cheaper set of solutions via speculative parallelism. The figure shows that increasing processors for this problem continues to yield significant benefits beyond 8 processors; as a result, the KSR1 is able to exploit its larger number of processors to advantage and outperform the Iris.

We can extend these observations to all of the machines in Table 3. On machines with large numbers of processors ($P \geq 32$ — KSR1 and Butterfly), tree parallelism does much better than loop parallelism. On machine with a small number of processors ($P \leq 8$ — 8CE and Iris) loop parallelism performs better than tree parallelism. On machines with about 20 processors (Balance, Symmetry, and TC2000) tree parallelism does better than loop parallelism, although the difference between the two is smaller than on the machines with more processors.

On this last set of machines (Balance, Symmetry, and TC2000), the relative benefit of tree parallelism varies considerably, even though all three machines have roughly the same number of processors. On the Balance, tree parallelism is only marginally better than loop, while on the TC2000, tree parallelism is much better than loop. This variation in the benefit of tree parallelism is due to an increase in the cost of communication (relative to the cost of computation) when moving from the Balance to the Symmetry to the TC2000. As communication costs rise, loop parallelism becomes less effective relative to tree parallelism. As discussed in Section 2.5, this occurs because only loop parallelism incurs any significant communication.

The second set of data in Table 3 confirms our expectation that the tradeoff between loop and tree parallelism shifts in favor of loop parallelism when the solution space is very dense. When the solution space density is 10^{-5}, the variance in search time among different subtrees is so small that tree parallelism is preferable only on machines with large numbers of processors ($P \geq 32$). As

expected, the benefits of loop parallelism depend on the relative cost of communication, and are most significant on older machines with relatively slow processors, low latency, and high bandwidth (e.g., the Balance and Symmetry) and less significant on recent machines with very fast processors, high latency, or limited bandwidth (e.g., the Iris and TC2000).

Thus we have shown that the relative performance of the two parallelizations across machines depends on processor speed, the relative cost of communication, and the number of processors available. Given two machines with a comparable number of processors, communication costs can tip the balance from loop parallelism to tree parallelism (as occurred with the Symmetry and TC2000 in a dense solution space). Given two machines with very similar architectures but a different number of processors, beneficial speculation can cause tree parallelism to dominate on the machine with more processors, while loop parallelism performs better on the machine with fewer processors (as occurred with the Symmetry and Iris in a sparse solution space). These effects are the result of the scalability of tree parallelism, the non-scalability of loop parallelism, and the need for communication in loop parallelism, as discussed in Section 2.5.

3.3 Varying the Input

For a given machine and problem, each parallelization outperforms the other on some inputs. We show selected examples in Table 4. In the examples, we are searching for one solution while varying the density of the solution space (inputs). We see that for each machine, loop is best in one case, tree is best in two cases, and one case is about even:

		10^{-5}	10^{-21}	10^{-35}	empty
8CE	loop	0.2912	13.7951	163.8678	0.6636
	tree	1.1243	11.0926	3.0933	0.5906
Butterfly	loop	0.7298	33.7188	541.5130	1.7667
	tree	2.3331	3.7634	8.0026	1.4941
Iris	loop	0.0227	1.0993	13.2450	0.0513
	tree	0.0753	0.7400	0.2407	0.0413

Table 4: Time in seconds searching for one solution; varying inputs.

Unlike the problems in the previous section, the results in Table 4 are consistent across all the machines we studied. Thus we see that the best parallelization for the "find multiple solutions" problem is machine-dependent, yet the best parallelization for the "find one solution problem" is not. This surprising result occurs because tree parallelism is highly speculative when only a single solution is required. The processor cycles spent on speculative computation are beneficial in a sparse solution space, but are wasted in a dense solution space, and are neutral in an empty solution space.

We can verify this by comparing the overheads experienced on the Iris by loop and tree parallelism when seeking one solution in a sparse (10^{-21}) solution space. Figure 6 shows the three most significant sources of overhead for both parallelizations, along with the amount of time spent in productive computation. The figure shows total time spent by all processors, so the height of the bar must be divided by the number of processors to get the actual running time. It shows that tree parallelism exploits speculation well, with cheaper solutions found when the third and the sixth

processors are added. It also shows that increasing processors in the loop parallelization increases communication and load imbalance enough that it cannot compete with tree parallelization.

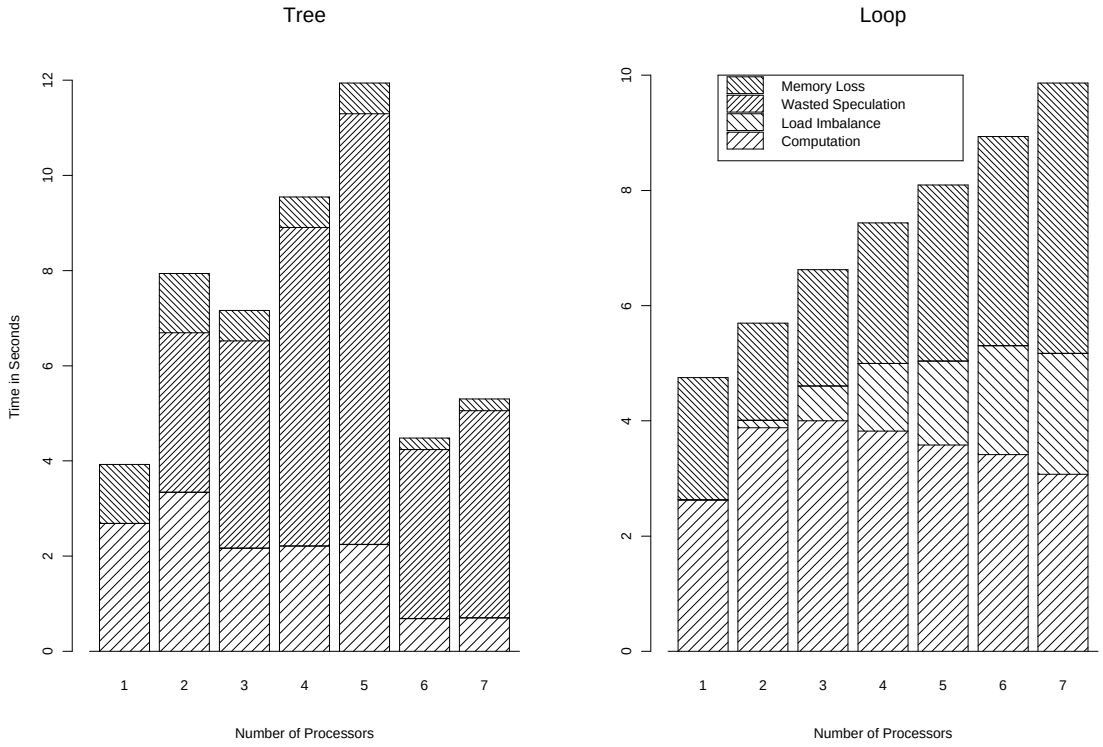

Figure 6: Overheads of loop and tree parallelism, seeking one solution in a sparse solution space.

3.4 Varying the Problem

For a given machine and input, each parallelization outperforms the other on some problems. As seen in Table 5, on each machine loop parallelization is best in at least one category, and tree parallelization is best in at least one category. The example is searching a dense solution space (10^{-5}).

		desire 1	desire 128	desire 256
Butterfly	loop	0.7298	4.0369	7.1970
	tree	2.3331	2.9804	3.2637
Symmetry	loop	0.3190	1.3143	2.3190
	tree	1.3214	1.6690	1.8000
Iris	loop	0.0227	0.0933	0.1660
	tree	0.0753	0.1087	0.1320

Table 5: Time in seconds searching a dense solution space; varying the problem.

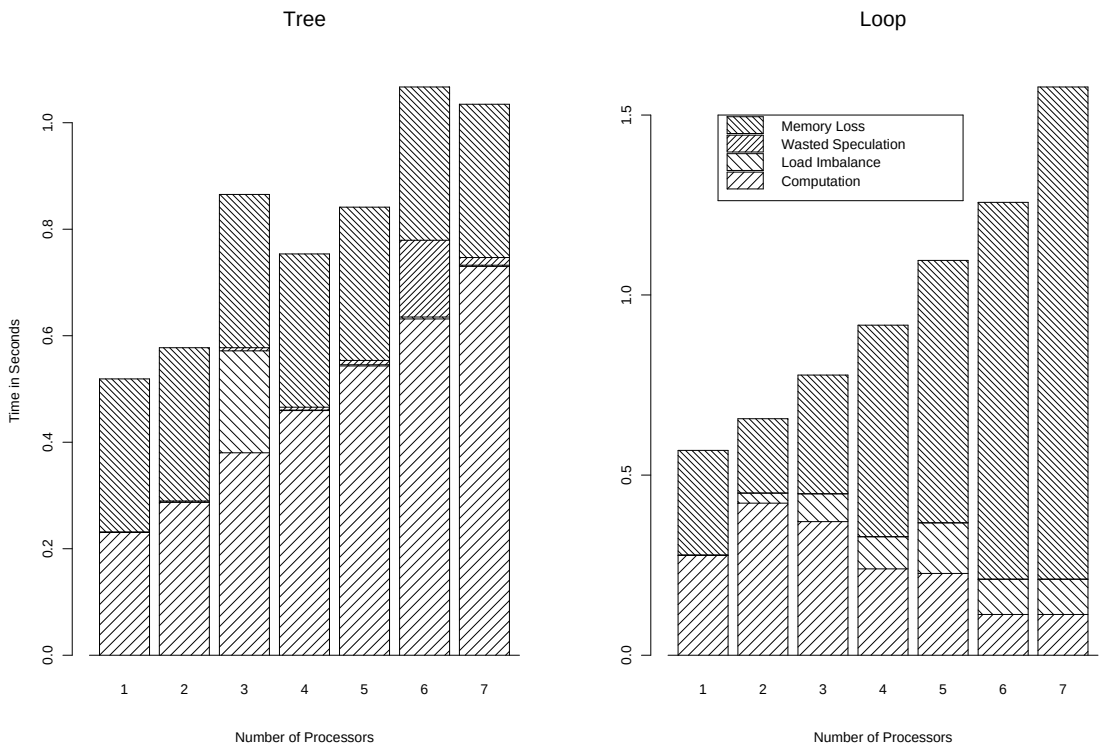

Figure 7: Overheads of loop and tree parallelism, searching a dense space for many solutions.

We can understand these performance figures by examining why tree parallelism outperforms loop parallelism on the Iris when seeking many solutions in a dense solution space. Figure 7 shows the three most significant overheads for this case. It shows that, for tree parallelism, the overhead due to communication stays roughly constant as we increase processors, and that the increased processing power is spent in useful computation. For loop parallelism, it shows a large increase in communication costs as we increase the number of processors. This occurs because node filtering is rapid in a dense space; all nodes are likely to lead to solutions. As a result, communication occurs more frequently as parallel loops are entered and exited more quickly.

Here the cost and benefits of speculative parallelism can be seen in another way; speculative parallelism wastes cycles when searching for a few solutions in a dense space, but when the number of solutions desired is much greater than the number of processors, speculative parallelism hurts very little, and loop parallelism incurs additional communication costs. In other words, when little processing power is wasted on speculation, tree parallelism excels because of its larger grain size.

3.5 Summary: Loop vs. Tree

The results in this section stand in sharp contrast to the notion that one "best" parallelization exists for this broad class of problems. We have shown that this notion does not hold even if any two of the three computation characteristics are held constant: machine, input, or problem. We have not used any characteristics specific to subgraph isomorphism in reaching this conclusion; the performance effects that we show here are due to the density of the solution space, simple variants of the problem definition, and the performance characteristics of the machine being used.

The two kinds of parallelism used in this program, loop parallelism and tree parallelism, are not

often well supported in the same language and runtime environment. These data argue that any language and runtime environment intended for use on this broad class of problems should provide good support for both kinds of parallelism.

4 Hybrid Parallelization

The previous section showed that, in parallel combinatorial search, tree parallelism and loop parallelism are both important and serve different roles. Loop parallelism speeds descent of the search tree, while tree parallelism helps find the best subtree as early as possible. The data in the previous section suggest that, for some inputs and problem spaces, a combination of both approaches may perform better than either alone.

Such a hybrid algorithm would be expected to do best on inputs and problems in which speculative parallelism is beneficial, and time spent in searching subtrees is significant. This type of problem could be considered midway between inputs with a dense solution space, in which speculation doesn't help, and inputs with a sparse or empty solution space, in which the whole graph must be searched and coarse grain size is most important.

We implemented such a hybrid algorithm to test this hypothesis. Our implementation partitions processors into groups of two; each group works together using loop parallelism on a single subtree. We ran this hybrid program on the Iris, using the graph inputs discussed in Section 3. Table 6 presents running times when searching for a single solution in a sparse space (which benefits from speculation). The table shows that the notable benefit obtained from speculation when the sixth processor is added is evident in both the tree and hybrid versions; however the hybrid version also benefits from loop parallelism, making it the best choice in this case.[6]

Processors	Loop	Tree	Hybrid
1	17.159	16.679	n/a
2	8.769	16.749	8.849
4	4.659	16.659	10.189
6	3.359	3.479	1.459
8	2.649	3.469	1.459

Table 6: Time in seconds for loop, tree, and hybrid programs.

These results indicate that there are problems which benefit from hybrid parallelism. To test our hypothesis that these problems occur in the inputs ranges between sparse and dense, but not near the endpoints, we ran the hybrid program on the Iris for a larger set of input graphs. The larger size of the input graphs helps show more markedly the relative performance of the three programs. In these tests, we varied the density of the solution space, starting out dense (where dense is approximately 10^{-38} for these larger graphs) and moving into the sparse range (about 10^{-62}). The results are shown in Figure 8. This figure shows solution space density decreasing to the left. As the solution space grows more sparse, loop parallelism and tree parallelism both increase in running time. However hybrid parallelism does not increase as fast as the other two, because it is exploiting speculation (this effect was verified by examining the raw data).

[6] The increase in running time from two to four processors in the hybrid case results from bus saturation interfering with loop parallelism, without tree parallelism providing any benefit.

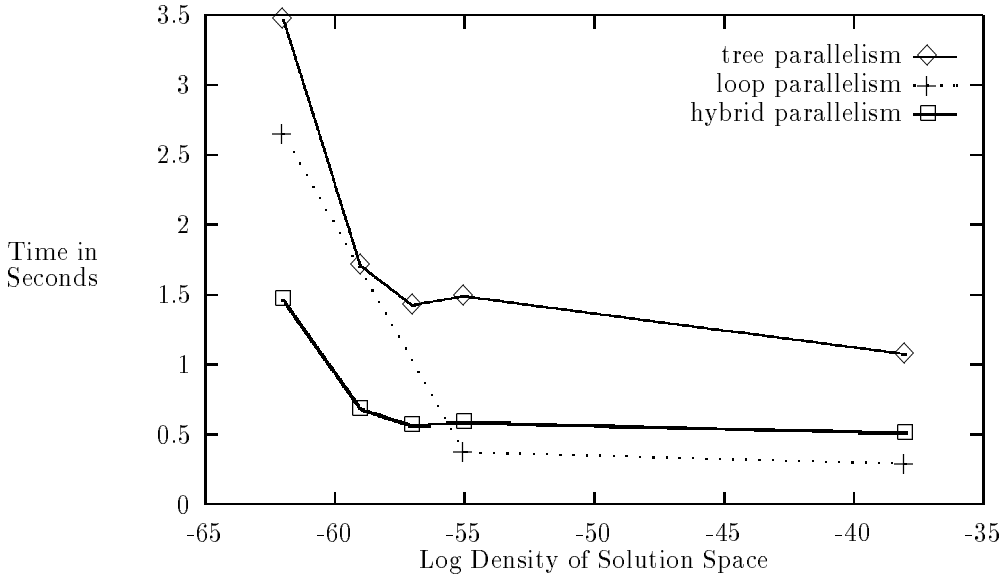

Figure 8: Performance of the three approaches over varying solution space densities.

These data suggest that not only is there need for both loop-parallel and tree-parallel versions of combinatorial search programs, but that there is a need for both parallelizations to be in use simultaneously. This suggests that any language or runtime environment supporting both parallelizations should also be able to support their simultaneous use.

We have extended some hypotheses about when each kind of parallelism is most useful in an application, but there is more work to be done before this problem is well understood. The inability to predict precisely in every case which static allocation of processors will perform best (all loop, all tree, or a hybrid) suggests the need for a dynamic allocation of processing power at runtime. Such an approach could exploit speculative parallelism when looking for few solutions in a sparse graph, using some measure such as non-uniformity of solutions in subtrees to determine when a subtree is promising and should be explored more quickly using loop parallelism. In a dense graph, when many solutions are required, such a dynamic approach could search the top levels of the tree quickly using loop parallelism, then descend the most promising subtrees in parallel using tree parallelism.

5 Conclusion

Our experiences with combinatorial search for subgraph isomorphisms have shown that the best choice of parallelization depends on several factors, including the problem, the input, and the machine. In particular, the choice between loop and tree parallelism can be difficult to resolve, and yet the choice can have a significant impact on performance. In some cases, the performance of loop parallelism dominates the performance of tree parallelism; varying the machine, the problem, or the input can cause the opposite to occur. In other cases, a combination of loop and tree parallelism performs best. Clearly, for this class of problem, there is no "best" parallelization.

In addition, our results clearly show that flat data parallelism (which roughly corresponds to what we've called loop parallelism) is not the sole source of parallelism, nor even the best source of parallelism, for this class of problem. A data parallel programming environment lacking support for nested parallelism would be unable to exploit all the readily available parallelism in this problem. This result is consistent with the observations in [Tichy *et al.*, 1992].

In order to exploit the various types of parallelism we've considered here, the language, compiler, or runtime system must have the facilities necessary to express (or find) each type of parallelism, and the mechanisms needed to implement each form of parallelism efficiently. The VCODE compiler for the Encore Multimax [Chatterjee, 1993] and the NESL compiler for the Connection Machine CM-2 [Blelloch *et al.*, 1993] support nested data parallelism, which can be used to express both types of parallelism in subgraph isomorphism. The Chores runtime system [Eager and Zahorjan, 1993] and the parallelizing Fortran compiler for the iWarp [Subhlok *et al.*, 1993] are recent examples of programming systems designed to support both data parallelism (e.g., parallel loops) and task (or functional) parallelism (e.g., tree search).

Our work with control abstraction in parallel programming languages [Crowl and LeBlanc, 1992; Crowl and LeBlanc, 1993] represents one approach to expressing and implementing multiple parallelizations within a single source code program. A control construct defined using control abstraction may have several different implementations, each of which exploits different sources of parallelism. Programmers can choose appropriate exploitations of parallelism for a specific use of a construct on a given architecture by selecting among the implementations. Using annotations, programmers can easily select implementations of control constructs (and hence the parallelism to be exploited) without changing the meaning of the program.

Our experiences with subgraph isomorphism illustrate the need for multiple parallelizations, and also indicate the need to tune the parallelization based on the problem, the input, or the machine. An approach that incorporates multiple parallelizations within a single source program, and allows the programmer (or compiler) to select alternative parallelizations easily, greatly simplifies the process of developing an efficient implementation of combinatorial search.

Acknowledgements

We thank Argonne National Laboratories for the use of their TC2000, International Business Machines for providing the 8CE, Donna Bergmark and the Cornell Theory Center for their assistance and the use of their KSR1, and Sequent Computer Systems for providing the Balance and Symmetry.

References

[Blelloch *et al.*, 1993] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha, "Implementation of a Portable Nested Data-Parallel Language," In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 102–111, May 1993.

[Chatterjee, 1993] Siddhartha Chatterjee, "Compiling Nested Data-Parallel Programs for Shared-Memory Multiprocessors," *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1993.

[Chen *et al.*, 1984] J. Chen, E. L. Daglass, and Y. Guo, "Performance Measurements of Scheduling Strategies and Parallel Algorithms for a Multiprocessor Quicksort," *IEEE Proceedings Part E*, 131(2):45–54, March 1984.

[Costanzo *et al.*, 1986] John Costanzo, Lawrence Crowl, Laura Sanchis, and Mandayam Srinivas, "Subgraph Isomorphism on the BBN Butterfly Multiprocessor," Butterfly Project Report 14, Computer Science Department, University of Rochester, October 1986.

[Crovella and LeBlanc, 1993] Mark E. Crovella and Thomas J. LeBlanc, "Performance Debugging using Parallel Performance Predicates," In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, May 1993.

[Crowl and LeBlanc, 1993] Lawrence A. Crowl and Thomas J. LeBlanc, "Parallel Programming with Control Abstraction," Technical Report 93-60-15, Computer Science Department, Oregon State University, June 1993.

[Crowl and LeBlanc, 1992] Lawrence A. Crowl and Thomas J. LeBlanc, "Control Abstraction in Parallel Programming Languages," In *Proc. 4th International Conference on Computer Languages*, pages 44–53, April 1992.

[Eager and Zahorjan, 1993] Derek L. Eager and John Zahorjan, "Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing," *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[Finkel and Manber, 1987] Raphael Finkel and Udi Manber, "DIB — A Distributed Implementation of Backtracking," *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[Markatos and LeBlanc, 1992] Evangelos P. Markatos and Thomas J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," In *Proc. Supercomputing '92*, pages 104–113, November 1992.

[Natarajan, 1987] K. S. Natarajan, "An Empirical Study of Parallel Search for Constraint Satisfaction Problems," Technical Report RC 13320, IBM T.J. Watson Research Center, December 1987.

[Rao and Kumar, 1989] V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search," *International Journal of Parallel Processing*, 16(6), 1989.

[Rao and Kumar, 1993] V. Nageshwara Rao and Vipin Kumar, "On the Efficiency of Parallel Backtracking," *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[Singh *et al.*, 1992] Jaswinder P. Singh, Wolf-Dietrich Weber, and Anoop Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20:5–44, March 1992.

[Subhlok *et al.*, 1993] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Programming Task and Data Parallelism on a Multicomputer," In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[Tichy *et al.*, 1992] Walter F. Tichy, Michael Phillippsen, and Phil Hatcher, "A Critique of the Programming Language C*," *Communications of the ACM*, 35(6):21–24, June 1992.

[Ullman, 1976] J. R. Ullman, "An Algorithm for Subgraph Isomorphism," *Journal of the ACM*, 23:31–42, 1976.

[Wah *et al.*, 1985] Benjamin W. Wah, Guo jie Li, and Chee Fen Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, pages 93–108, June 1985.