BEYOND DATA PARALLELISM
The Advantages of Multiple Parallelizations in Combinatorial Search

Lawrence A. Crowl Mark Crovella* Thomas J. LeBlanc* Michael L. Scott*

Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 92-80-06

December 1992

Abstract

In this paper we consider alternative schemes for parallelizing combinatorial search, and examine the factors that influence the choice of parallelization 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 affect the choice of an efficient parallelization. Our experiments, which span seven different shared-memory machines, eight parallelizations of the subgraph isomorphism algorithm, and a wide range of input graphs, show that there is no best parallelization for this problem: relative performance depends on each of these factors. In particular, on some machines and for some inputs, a sequential depth-first search of the solution space, applying data parallelism at each node in the search tree, performs best. On other machines or other inputs, parallel tree search, with no data parallelism, performs best. In still other cases, a hybrid solution, containing both parallel tree search and data parallelism works best. From these experiences we conclude that multiple parallelizations of a single algorithm may be needed to achieve the best performance over a range of machines, inputs, and problem demands, and that hybrid forms of parallelization (where data parallelism and tree parallelism are used in tandem) are needed in some applications.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, 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.

*Department of Computer Science, University of Rochester, Rochester, New York, 14627

1 Introduction

As parallel processors increase in size, it becomes harder and harder to develop applications with sufficient parallelism to make use of all the available processors. On small machines (e.g. multi-processor workstations), it is commonplace to run a modest number of heterogeneous processes, such as the independent programs of a typical time-sharing mix, or the pieces of a small-scale parallel application that has been divided along functional lines. On large-scale machines, however, successful exploitation of the hardware has generally required applications with a highly regular structure and, consequently, large amounts of *data parallelism*.

Two of the problem domains for which large-scale parallelism has proven most successful are on-line transaction processing and demanding scientific computations. The former domain is characterized by a very large number of small, mostly independent tasks, which can be executed in parallel by a collection of servers. The latter domain is characterized (at least in large part) by the parallel application of a local operation throughout one or more very large arrays. It has been argued [Gustafson *et al.*, 1988] that scalability depends on ever-increasing data sets, and it is tempting to suspect that problems with this sort of highly regular data parallelism are the only ones for which very large-scale machines will be practical. We argue in this paper, however, that there are important classes of parallelizable problems whose structure, while in some sense regular, is not data parallel in the usual sense of the word. Specifically, we argue that there are problems containing several fundamentally different kinds of potential parallelism, where no one parallelization is consistently best. Using combinatorial search as our problem class, and focusing in particular on the problem of finding *subgraph isomorphisms*, we show that the choice between alternative parallelizations depends on the underlying architecture, the expected density of the solution space, and the number of solutions desired. For certain combinations of parameters, we find that it pays to employ a hybrid approach that mixes parallelizations. We conclude that neither straightforward loop-based data parallelism nor thread-based functional parallelism 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 to express the full range of programs on large-scale parallel machines.

We present the problem of subgraph isomorphism, and a parallel algorithm to solve it, in Section 2. Using results culled from over 37,000 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 in Section 4, and 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-hard, 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 postulates a mapping from one 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 $j > i$.

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 verticies in the small graph, L is the number of verticies 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. 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. Formally, we assign a vertex i in the small graph to one of its remaining possible mappings j in the large graph. Since the vertex i in the small graph may map to only one vertex j in the large graph, we remove all other mappings for the small vertex, that is remove $(i, k \neq j)$ in the partial isomorphism. In addition, no two small vertices may map to the same large vertex, so we remove the postulated large vertex from the possible mappings of all other small vertices, that is remove $(k \neq i, j)$.

Since the search space is very large, it is prudent to elimiate 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 verticies within the small graph and between verticies and their potential mappings, we apply only two filters, *vertex distance* and *vertex connectivity*. These filters are not necessary, but they prune the search space enough to make the problem tractable.

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 removes entry (i, j) of the matrix if some vertex h in the small graph has already been mapped to a vertex k in the large graph and the distance from h to i is less than the distance from k to j. The distance filter uses precomputed distance matrices for the two input graphs.

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. Formally, the filter eliminates entry (i, j)

of the matrix if there exists a neighbor k of i in the small graph for which there is no remaining possible mapping to a neighbor of j in the large graph. Our vertex connectivity filter resembles the one employed by Ullman, except that for the sake of speed we do not iterate after eliminating a possible mapping to see whether its elimination would allow us to eliminate additional mappings.

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 Analysis of the Search Space

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 graph, 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. The expected degree of a vertex in the small graph is $(S-1)s$; the expected degree of a vertex in the large graph is $(L-1)l$.

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.

The expected number of solutions (i.e., isomorphisms) is the number of leaves times the probability that a given leaf is a solution. To determine this probability, which we refer to as the *density* of the problem space, we first note that the expected number of edges in the small graph is $\binom{S}{2}s$. Also, given a mapping from two connected vertices in the small graph to two vertices in the large graph, the probability that the two vertices in the large graph are connected is simply l. We assume that edge occurrences are independent events in both graphs, and therefore

$$density \equiv l^{\binom{S}{2}s}$$

In the experiments reported here, we use this definition of density to characterize the input graphs.

Solutions are not randomly distributed among the leaves of the search tree. The filters described earlier are highly effective in practice because many subtrees contain no solutions while others contain many solutions. For example, when $S = 32$, $L = 128$, $s = 0.9$, and $l = 0.2$ ($density \approx 10^{-312}$), 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.

When the search reaches level k in the tree, there are $S-k$ vertices remaining to be matched in the partial isomorphism. We can expect $\binom{k}{2}s$ of the $\binom{S}{2}s$ edges in the subgraph to connect vertices seen so far, and $\binom{S}{2}s - \binom{k}{2}s$ edges to remain to be matched. If all edges joining matched vertices in the small graph are also present in the large graph (note that this is not always guaranteed by the filters we employ), then the probability that any particular leaf below us in the search space represents an isomorphism is

$$l^{\binom{S}{2}s-\binom{k}{2}s}$$

a quantity that can be considerably larger than the density of the problem space as a whole. For example, when $S = 32$, $L = 128$, and overall density $\approx 10^{-18}$, subtree density is over 1000 times the overall density only 11 levels down in the tree (again given that all edges joining matched vertices in the small graph are also present in the large graph). This analysis shows that pruning the search tree can be highly effective in reducing the number of nodes searched.

2.3 Parallel Implementations of the Subgraph Isomorphism Algorithm

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;

Label	Source of Parallelism
tree	search of different children of the root node
filter	application of different filters
loop	examining small "relatives" of the postulated mapping
vector	constraining mappings for each "relative"
word	constraining mappings for each "relative"

Table 1: Exploitable parallelism, from coarsest to finest.

we can search each subtree of the root node in parallel (hereafter referred to as *tree parallelism*), with depth-first, sequential search at the remaining levels. 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.[1] We can also exploit parallelism when applying a filter to a candidate mapping.

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*.[2] Although both vector and word parallelism exploit the same

[1] 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.

[2] None of the machines in our study support vector operations, so we did not exploit this form of parallelism.

source of parallelism (the possible mappings of a given small vertex), they can be used individually or in tandem.

The presence of both tree parallelism and parallelism within the tree-pruning step has been noted by other researchers [Wah *et al.*, 1985; Natarajan, 1987; Finkel and Manber, 1987; Rao and Kumar, 1989]. However, there has been little previous attention given to whether there is a single best parallelization of branch-and-bound search for a particular machine, input, or problem. In fact, we find that 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 the solution space), and the underlying architecture.

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

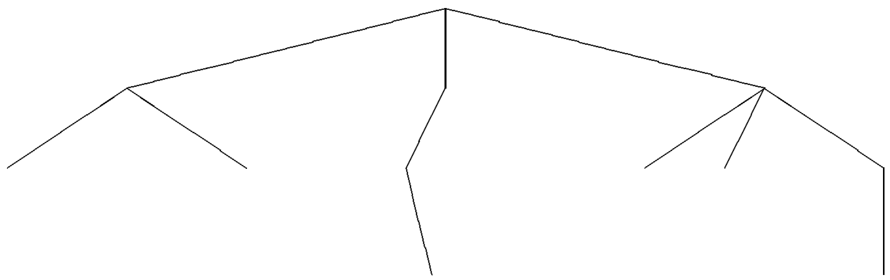

Figure 1: A sparse search tree.

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.

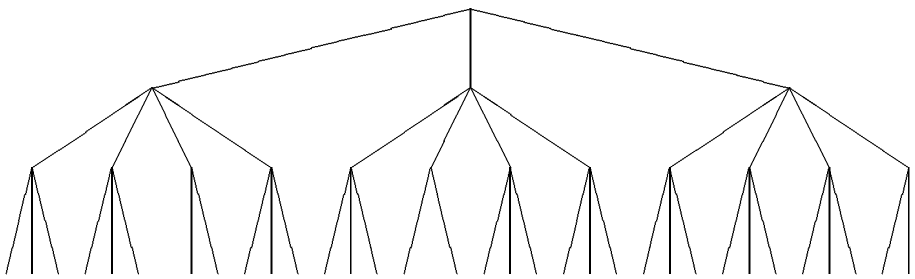

Figure 2: A dense search tree.

Even when, as above, 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 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.

The underlying architecture (hardware and software) is also an important factor to consider when choosing a parallelization. For example, both vector and word parallelism require the appropriate operations in hardware. The performance effects of word parallelism depend on several factors, including the time required to pack and unpack the representation of mappings into a single word, the number of parallel operations on mappings, and the time required to extract a single bit from the representation. Also, since the packed representation of mappings reduces the overall bandwidth required to read and write mappings, word parallelism will lower communication costs (and perhaps even contention overhead), and therefore have a significant performance effect on machines where communication is expensive.

The underlying architecture may also affect the choice between loop and tree parallelism. Loop parallelism is particularly appropriate on machines and software systems that support fine-grain parallelism, while the coarser-grain tree parallelism could be used on any shared-memory multiprocessor (or even distributed-memory multiprocessors) using operating system processes or lightweight threads.

Since there are so many possible parallelizations of the subgraph isomorphism algorithm, it is difficult to choose a particular parallelization without a better idea of how various problem parameters affect the choice. Indeed, our first effort in parallelizing subgraph isomorphism [Costanzo *et al.*, 1986] chose to exploit tree parallelism (because of its lower synchronization costs) when the problem parameters were such that tree parallelism is not very effective. In the remainder of this paper, we examine the dependence of performance on the underlying architecture, the structure of the input graphs, and the number of desired isomorphisms, on a range of shared-memory multiprocessors.

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 pruning be parallelizable. (Section 4 discusses situations for which parallelizing both search and pruning is useful.) As a result, good performance requires the ability to parallelize the problem using both loop parallelism and tree parallelism.

This surprising 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, solving varying problems.

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 and Table 2. We study four input data sets: one in which solutions are extremely plentiful; two in which solutions exist but are relatively rare; 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 parallizations, amounting to over 37,000 data points.

In this presentation, we will not address variability in performance due to the number of processors used and the choice of whether or not to exploit word parallelism. To ensure fairness, 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
8CE	7	IBM 8CE
Balance	20	Sequent Balance
Symmetry	19	Sequent Symmetry
Butterfly	39	BBN Butterfly One
TC2000	21	BBN Butterfly TC2000
Iris	8	Silicon Graphics Iris
KSR1	32	Kendall Square Research 1

Table 2: Machines used in this study.

The IBM 8CE uses the ROMPC processor (as found in the IBM RT). In addition to a global shared memory, each processor has its own local memory. Access to non-local memory is through a shared bus, and the access times to local, global and another processor's local memory are in the ratio 1:2:5. The 8CE has an experimental operating system [Bolosky *et al.*, 1989] that dynamically places shared pages in global shared memory and leaves unshared pages in local memories. There is no memory cache.

The Sequent Balance uses the National Semiconductor 32032 processor. There is a single global shared memory, accessed through a shared bus. Each processor has an 8KB write-through cache.

The Sequent Symmetry uses the Intel 80386 processor. There is a single global shared memory, accessed through a shared bus. Each processor has an 64KB write-back cache.

The BBN Butterfly One uses the Motorola 68000 processor, each with its own local memory. Each processor may access another's memory through an FFT switching network. The access times to local memory and another processor's memory are in the ratio 1:5. There is no memory cache.

The BBN Butterfly TC2000 uses the Motorola 88100 processor, each with its own local memory. Each processor may access another's memory through an FFT switching network. The processor data caches are not used for shared data because they are not kept consistent.

The SGI Iris uses the MIPS R3000 processor. There is a single global memory accessed through a shared bus. Each processor has a two-level cache. The second level cache is 1 MB, which is large enough so that the Iris is most effectively programmed as though it had local memories.

The KSR 1 uses a custom 64-bit processor. All memory in the system is managed as a set of caches, with each processor containing a 32 MB cache 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 is through a high-speed ring network.[3]

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^{-13}). We see that loop is best on two machines, tree is best on four other machines, and one machine (the Balance) is a toss-up. For this and subsequent tables, we underline those times which are significantly better.[4]

		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: Searching for 128 solutions; varying machines.

The second example in Table 3 is searching for multiple solutions in dense input. (Looking for 128 isomorphisms with solution space density 10^{-2}). Again, there is a number of machines for which loop is best (here, 3), and a number for which tree is best (here, 2). In this case the two parallelizations are a toss-up on two machines.

There seems to be no simple method to determine which parallelization is better for a given machine, other than to run both versions. This is because the choice of which parallelization is better seems to depend on processor speed, memory speed, and interconnect latency and bandwidth.

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

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

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 in best in two cases, and one case is about even:

		10^{-2}	10^{-13}	10^{-18}	empty
8CE	loop	<u>0.2912</u>	13.7951	163.8678	0.6636
	tree	1.1243	<u>11.0926</u>	<u>3.0933</u>	0.5906
Butterfly	loop	<u>0.7298</u>	33.7188	541.5130	1.7667
	tree	2.3331	<u>3.7634</u>	<u>8.0026</u>	1.4941
Iris	loop	<u>0.0227</u>	1.0993	13.2450	0.0513
	tree	0.0753	<u>0.7400</u>	<u>0.2407</u>	0.0413

Table 4: Searching for one solution; varying inputs.

The results that Table 4 shows are basically consistent across all the machines we studied, unlike the problems in the previous section. 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 intrinsically "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.

3.4 Varying the Problem

For a given machine and input, each parallelization outperforms the other on some problems. On each machine in the example of Table 5, 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^{-2}).

Here the cost and benefits of speculative parallelism can be seen in another way; speculative parallelism wastes cycles when searching for 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

		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: Searching a dense solution space; varying the problem.

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 for parallelism that is 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 dense inputs, in which speculation doesn't help, and sparse or empty inputs, 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; each group works together using loop parallelism on a single subtree. The allocation of processors is chosen by the user. We ran this hybrid program on the Balance, using the graph inputs discussed in Section 3, searching for a single solution (which benefits from speculation). Table 6 presents running times with four processors in each group.

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

Processors	Loop	Tree	Hybrid
4	3.016	5.366	1.800
8	2.572	5.200	1.789
12	2.472	4.916	1.841
16	2.377	4.900	1.846

Table 6: Performance of loop, tree, and hybrid programs.

dense is approximately 10^{-20} for these larger graphs) and moving into the intermediate range (about 10^{-36}). The results are shown in Figure 3. This figure shows problem space density decreasing to the left. As problem 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).

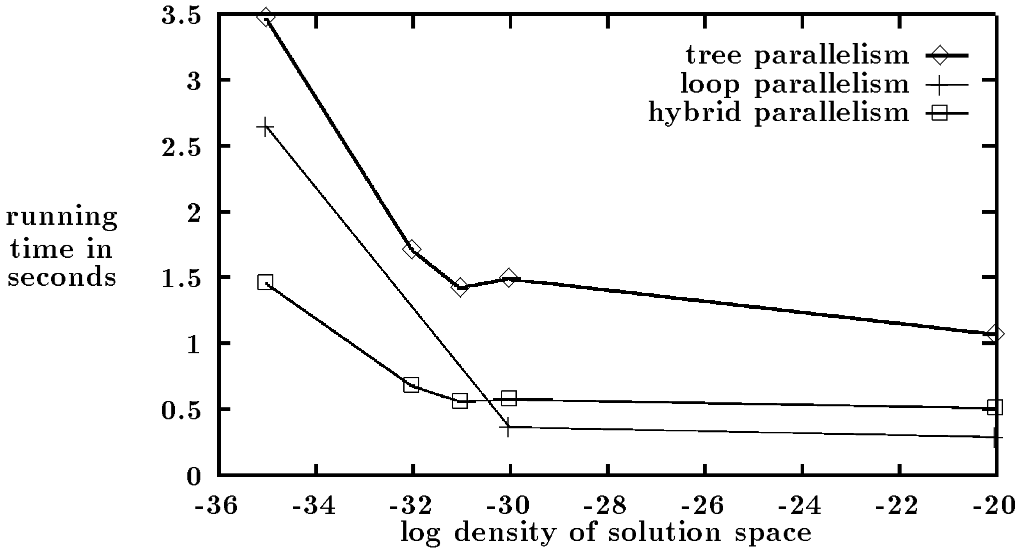

Figure 3: Performance of the three approaches over varying solution space densities.

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 descending 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 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 tree parallelism (possibly implemented by explicit lightweight threads) would be unable to exploit all the readily available parallelism in this problem.

In order to exploit the various types of parallelism we've considered in this study, 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. Our previous work with control abstraction [Crowl and LeBlanc, 1991; Crowl and LeBlanc, 1992] explored one such approach to expressing and implementing multiple parallelizations within a single source code program. Our experiences with subgraph isomorphism not only illustrate the need for multiple parallelizations, but also indicate the need to tune the parallelization based on the problem, the input, or the machine. Therefore, an approach such as ours that incorporates multiple parallelizations within a single source program, and that allows the programmer to select alternative parallelizations easily, is particularly welcome.

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

[Bolosky *et al.*, 1989] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott, "Simple But Effective Techniques for NUMA Memory Management," In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona, December 1989, also appeared in Operating Systems Review 23(5), December 1989.

[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.

[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.

[Crowl and LeBlanc, 1991] Lawrence A. Crowl and Thomas J. LeBlanc, "Architectural Adaptability in Parallel Programming Via Control Abstraction," Technical Report TR359, Computer Science Department, University of Rochester, February 1991.

[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.

[Gustafson *et al.*, 1988] J.L. Gustafson, G.R. Montry, and R.E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal on Scientific and Statistical Computing*, 9:609–638, 1988.

[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.

[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.