THE USE OF PARALLELISM TO IMPLEMENT A HEURISTIC SEARCH

William A. Kornfeld

MIT Artificial Intelligence Laboratory

ABSTRACT

The role of parallel processing in heuristic search is examined by means of an example (cryptarithmetic addition). A problem solver is constructed that combines the metaphors of constraint propagation and hypothesize-and-test. The system is capable of working on many incompatible hypotheses at one time. Furthermore, it is capable of allocating different amounts of processing power to running activities and changing these allocations as computation proceeds. It is empirically found that the parallel algorithm is, on the average, more efficient than a corresponding sequential one. Implication* of this for problem solving in general are discussed.

1. Introduction

Many AI systems that perform a "heuristic search" (i.e. they can be thought of as searching some space of possibilities for an answer) are based upon one or both of two programming techniques known as *constraint propagation* and *hypothesize-and-test*.

In a system based on constraint propagation, internal data structures represent (implicitly or explicitly) potentially acceptable points in the search space. Computation proceeds in narrowing down these possibilities by employing knowledge of the domain in the structure of the computation. There is not enough space here to properly introduce the concepts involved in constraint propagation. The reader is referred to some systems described in the literature [1, II) for an introduction. One point we wish to emphasize about pure constraint propagation is that at any time the internal data structures will be consistent with any solution to the problem. Thus, if more than one solution is possible, pure propagation of constraints will be unable to select only one of them. Further, even if a unique solution exists, a constraint propagation system may not be able to find it.

The hypothesize-and-test methodology allows the program to make assumptions that narrow the size of the search space; there is no guarantee that the assumption is consistent with any solution to the problem. The program continues to make hypotheses until a solution is located, or it has been determined that no solution is possible with the current set of assumptions. There is no requirement that any hypothesis be correct and so mechanisms must be available that prevent commitment to any hypothesis until it has been demonstrated to be acceptable. The most commonly available mechanism is known as *backtracking*. Backtracking allows the program to return to an environment that would exist had that assumption not been made.

As long as the search space is enumerable (a very weak assumption)' hypothesize-and-test can be easily seen to be logically more powerful. If there are several consistent solutions, a pure constraint propagation system has no way to establish preference for one of them. Even if only one solution is possible a constraint propagation system will not necessarily find it; this will be demonstrated later by example. The proponents of constraint propagation point out that hypothesize-and-test is grossly inefficient in situations where constraint propagation can function (see for example Waltz [13]). The example in this paper bears out this claim, though one recent study [4] suggests there are situations in which pure backtracking is more efficient than constraint propagation.

One can, however, imagine a composite system that has aspects of both constraint propagation and hypothesize-and-test. In such a system, constraint propagation can be used to prune the search space, yet allowing hypothesize-and-test to continue the search where constraint propagation is not able to. A constraint language that can support the creation of such systems has been constructed by Steele [12]. Steele allows assumptions to be made and backtracking performed. The current work discusses another such system in which the hypothesize-and-test methodology allows more than one assumption to be pursued concurrently. It is an extension of earlier work discussing parallel problem solving systems [7, 8] and a language, Ether, for implementing these systems. Here we examine one particular kind of search problem, *cryptarithmetic addition,* of the sort used by Newell and Simon [10} We study this problem, not because it is interesting in itself, but because it is well-defined and test cases are relatively easy to come by. This allows us to test the efficiencies of algorithms empirically. We have constructed a parallel problem solver for doing these cryptarithmetic problems.

There are two main points we wish to make:

1. That a system combining both constraint propagation and parallel hypothesize-and-test methodologies can be constructed. The code is simple to read, write, and understand. Example code is presented.

2. That, on the average, a parallel program for solving these puzzles can be constructed that requires less average run time *when the parallel program is executed by time-slicing on a single processor* than a sequential program executed on the same processor. Obviously, it matters which sequential and which parallel program we compare: the benchmarks for this comparison will be explained later and are, I think, quite reasonable. The speedup we are talking about here is not large, but is noticeable. The important point is that it is present at all. A similar effect has been noticed in other suidies for various problems [6, 8J. It suggests that concurrency may be a useful for the design of heuristic search algorithms whether or not the programs are executed on concurrent hardware or a conventional sequential computer.

The remainder of this paper consists of a discussion of the problem being solved and the nature of the parallel solution. We show how the efficiency of the parallel program depends on the use of heuristic information for allocating resources of the parallel program. We then develop a series of allocation strategies, each one improving on the previous one. We finally discuss the importance of this experiment for a general theory of problem solving. We show how the allocation strategies represent a use of what has been celled *mcta-lefcl knowledge* in the literature, i.e. knowledge about

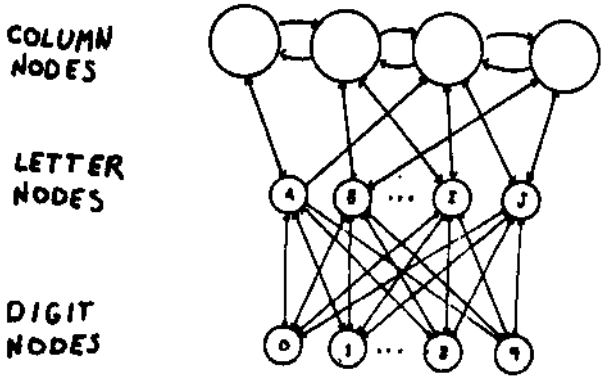
how to guide the search process to gain efficiency. In this study, concurrency is necessary to make use of this mete-level knowledge.

2. The problem

We ere given three strings of letters, t.g. "DONALD", "OKRALD", and "ROBKM" that represent integers when substitutions of digits are made for each of the letters. There is at least one possible assignment of digits for letters so that the numbers represented by the first two ("DONALD" and "orRAH/), when added, yield the number represented by the third ("ROttt-RT"). Any one of these assignments is a solution. In the problems we will be looking at, each will contain exactly ten letters. A solution consists of a mapping from these ten letters onto the ten digits 0 through 9.

3. A Constraint Propagation Solution

In our construction of the constraint network wt will use the actor model of computation.. We find it a very natural formalism for building these sorts of systems. In this formalism nodes of the network are implemented as actors. Constraint propagation between nodes is implemented by sending of messages containing the new constraints to the node being constrained. For our cryptarithmetic problem solver we have three kinds of nodes: letters, digits, and columns. They are arranged as shown:



Arcs in the diagram indicate flow of constraints. Thus column nodes can constrain their left and right neighbor columns and certain letter nodes (the ones representing letters contained within the column). Letter nodes can constrain digit nodes and column nodes that contain their respective letters. Digit nodes can constrain letter nodes. In the initial configuration, before constraint propagation begins, we store *at* each letter node a list *of* possible digits that contains all ten digits. Similarly, each digit node contains • "possible letters list" containing all ten letters. We will give a short description of what each node has to do when it receives a message informing it of a new constraint.

Column!. A column can receive messages informing it of new constraints on letters it contains and on possible values for its carry-in and carry-out If a column node receives any such messages, it computes possible new constraints on its letters, carry-in, and carry-out. If any one of these has no possible values a CONTRADICTION is asserted. When a CONTRADICTION is asserted the code implementing hypothesize-and-test is invoked to take an appropriate action. New constraints on letters are sent to the respective letter nodes. New constraints on carry-in and carry-out are sent to the right and left neighbor columns respectively.

Letter.. Letters receive messages
subsets of the digits

0 through 9 that they can possibly be. If they learn of digits that they cannot be, nodes representing those digits are sent messages. Also, each column that contains the letter receives a new message informing it of the new restrictions on the value of the particular letter. If the set of possible digits becomes null, a CONTRADICTION is asserted.

Digits. These receive messages from letter nodes indicating that they are or are not the respective letter. If the set of possible letters is reduced to a singleton, a message is sent to the particular letter. If the set of possible letters is reduced to null, a CONTRADICTION is asserted.

We can observe some things about the ability of this system to satisfactorily derive a unique solution. First, if there is more than one possible solution it will not find any of them. Since the letter and digit assignments of each possible solution are certainly possible assignments, they will appear on the possibility lists attached to each node. Even if there is only one possible solution (or no possible solutions) the system may not find it (or discover that no solutions exist). For example, the "DONALD" + "OKRALD" - and "RODF.RT" puzzle has only *one* solution: the constraint network described will quiesce before finding it. Nevertheless, the knowledge can be said to be "present" in the network: if the nodes of the network are instantiated with an assignment of leters to digits, the network will assert a CONTRADICTION iff the assignment is not a solution. Our constraint network, then, needs the ability to make assumptions and test them if it is to be able to solve these puzzles.

4. Hypothesize and Test in Ether

The constraint network and hypothesize-and-test methodologies were written in the Ether language [7, 8). We will only give enough details about the implementation to support the ensuing discussion. The interested reader is referred to [9] for a more detailed discussion of the implementation.

The primitive operations of the Ether languages are based around the notion of an *assertion* rather than message passing. Rather than coding in a message passing formalism "Send the node for the letter D that is 5" we instead say "Assert that D is 5" and a process of compilation turns this assertionai code into a message passing implementation. For certain problems this process of compilation is important because certain ideas can be expressed quite naturally in the assertionai form that compile into very complex message passing code. These issues will be discussed in *[9*

Because we are interested in the possibility of pursuing more than one instantiation of the constraint network in parallel, we need the ability to have more than one available for processing. For this we introduce the notion of a *viewpoint.* Each viewpoint tags a mutually compatible collection *of* assumptions about the possible values *of* letters and digits together with the constraints that derive via propagation from these assumptions, (i.e. a viewpoint is one particular instantiation of the network). Viewpoints are related to each other by an inheritance mechanism. The viewpoint in which A is assumed to be 5 and B is assumed to be 4 might be a subviewpoint of the one in which A is assumed 5 and no other assumptions have been made. Viewpoints are the repositories of assumptions and facts derived from these assumptions.

In order to be able to hypothesize and test we need to introduce some control primitives. These primitives are built around a construct known as an *activity.* All processing that happens during execution happens under the auspices of some activity. There are language constructs for conveniently grouping parts of a related task into a single activity. For example, we can create an activity, make a new assumption in a viewpoint, and cause all further work within the viewpoint (i.e. all further constraint passing in the instance of the

network defined by the sumption) to be part of the activity.

Activities are of interest because they give us ways to control quantities of system resources available for the execution of alternative explorations. If we *itiflt* an activity, all execution with the activity stops; a stifled activity cannot be restarted. We also have the ability to control the rates that non-stifled activities run. Different activities can be assigned different amounts of *processing power,* the total amount of CPU time an activity will get during an interval of time is proportional to its processing power. The processing power of *tn* activity can be altered by the system asynchronously with the running of the activity.

Systems using hypothesize-and-test can be constructed in Ether by using *viewpoints* to represent assumptions made, and *octititics* to control which parts of the search space are explored, and with what vigor.

5. A Simple Parallel Solution

In this treatment we will ignore many details of how both the Ether system and the crypt arithmetic system implemented within it are constructed. If we wish to "create a new instance of the constraint network" that inherits from another, we create a new viewpoint (using the new-viewpoint construct). To add an assertion about a letter being associated with a digit within the context of this viewpoint, we execute (assert (one-of -letter (-digit))) where letttr and digit are bound to the respective letter and digit which we want to assume are identified in this viewpoint. The second argument to one-or is a list of possible digits that the letter can be. So, for example, we could execute (essert (one-of s (l J s ; 9))) to indicate that S is odd. Ether syntax makes use of a *quasi-quote* convention in which symbols prefaced by the character V are substituted with the values of the associated symbols. If letter were bound to "D" and digit were bound to T, the item actually asserted would be (one-of D (S)). If the titert is executed within the context of a certain activity, then all. work propagating constraints that follow from that assertion will happen within that activity.

The implementation described in this section is quite simple. It first creates a viewpoint in which no assumptions are made and continues propagating constraints within this viewpoint until it has *quicsccd,* i.e. no more propagation can happen. When this state has been achieved, if each letter does not have a unique digit that it can be identified with, it is determined which letter has the least number of possible digits that it can be (excluding those letters that already have a unique assignment). For each one of these digits, a new viewpoint and a new activity are created. Within these (in parallel), the letter is asserted (assumed) to be the digit and propagation of constraints continues. If quiescence is reached in this new activity and the problem has not been solved, we recurse.

The function shown below takes a letter, a list of alternative digits, and a viewpoint. It uses the environment contained in the viewpoint to create new subviewpoints in which the letter is assumed to be each of the alternative digits. We first check to see if there is at least one possible digit. If not, there cannot be a possible solution to this problem consistent with the parent viewpoint and so we assert that there is a contradiction within the parent viewpoint. Otherwise we iterate over each digit in the alternatives list and for each one we create a new viewpoint whose parent is the parent viewpoint and a new activity with parent start-act and assert the letter is the particular digit; this initiates propagation of constraints. If we discover there is a contradiction within the viewpoint (this is accomplished by the code, fragment beginning with "(when ((contradiction)!") we assert within the parent viewpoint, that the letter cannot be the particular digit. We are justified in doing this because the only difference, in terms of assumptions made, between the current viewpoint and the parent viewpoint is the one assumption of the totter being identified with • particular digit that was a possible alternative in the parent viewpoint: if this assumption leads to a contradiction, we know that this is not a possible identification for the letter. In addition we stifle (stop from executing) the activity that was pursuing the now known to be invalid assumption. We further check to see if the activity quiesces in the section of code beginning with "(when ((euitictnt *•))". If this has occurred, we first check to see if the problem has been solved. If so we are done: otherwise we determine the letter in the viewpoint with the least number of possible digits (but greater than I) and recursively call parallel-seiva on this.

```
(defun parallel-solve (letter alternatives parent-viewpoint)
  (if (null alternatives)
      ;If there are no viable alternatives,
      ;then there is no consistent assignment possible
      (within-viewpoint parent-viewpoint (assert (contradiction)))
      ;Otherwise, fork on each alternative
      (foreach
         digit
         alternatives
         (let ((v (new-viewpoint parent parent-viewpoint))
               (a (new-activity  parent start-act)))
           (within-viewpoint v
              (assert (one-of ~letter (~digit)))
              (activate
                (when ((contradiction))
                   (within-viewpoint parent-viewpoint
                     (assert (cant-be ~letter ~digit)))
                   (stifle a))))
           (activate
             ;If the activity has quiesced, we must first check if
             ;the problem has been solved; if so, we are done.
             ;Otherwise we must pick a new branch to go
             ;down in a depth-first fashion.
             (when ((quiescent ~a))
               (if (total-solution
                      (quiescent-letter-constraints v))
                   (halt-other) ;indicates we are done
                   (let ((minpair
                           (minimum
                             #'(lambda (pair)
                                 (let ((L (length (cadr pair))))
                                   (if (= L 1) 11. L)))
                             (quiescent-letter-constraints v))))
                     (parallel-solve
                       (car minpair) (cadr minpair) v)))))))))
```

When a new activity has been created (and has something to do) it immediately begins executing concurrently with already existing activities. The default allocation of processing power, when no explicit allocation has been done, is such that each running activity gets approximately equal servicing (in terms of CPU seconds) by the scheduler.

6. Alternative Parallel Program

The simple parallel program described might well be reasonable if we had a large number of processors. With a small number of processors (in particular, only one processor, the case actually studied) it is considerably less efficient in terms of average total run time than some other solutions. All the solutions we wilt examine are elaborations of, or simple modifications to the basic parallel program already presented.

We observe that a traditional depth-first search (with backtracking) is but a trivial modification of the code above. When new alternative digits are proposed for a letter, instead of starting them up in concurrent viewpoints as was done above, they are placed on a list. Only the activity for the first one on the list is given any processing power. If it quiesces we recursively call parallel-solve. If it is discovered that the viewpoint is contradictory, the next one is begun (if a next one exists); otherwise, the parent viewpoint is asserted to be inconsistent. Asserting that it u inconsistent will trigger the

activity monitoring the next higher viewpoint to pick the next possibility on its list. Depth-first b a degenerate case of parallel anarch in which only one activity at a time b given non-xero processing power.

6.1 HeurIftIc Information to Control Itaoarce Allocation

A simple elaboration wt can make to the parallel implementation presented that preserves to parallel character is to vary the processing power based on an assessment of how likely the assumptions we have made within its associated viewpoint are to lead to useful information (either leading to a solution or determining that the viewpoint is contradictory). We base the quantity of processing power allocated to the activity doing the exploration on the numerical value of this judgement. For this particular problem, we are more likely to learn in a short period of time whether a viewpoint contains a valid solution or is contradictory if it is already fairly well constrained, Le. if the letters in the viewpoint only have a few possible digits that they could be. After some experimentation we came upon the following formula for determining relative processing power allocations for the various different activities participating in the search:

$$((10-n_1)^2 + \ldots + (10-n_{10})^2)^2$$

where each ni is the number of possible digit assignments for the letter i in the viewpoint If the letters tend to have fewer possible digit possibilities, the sum terms (IO^nj) will tend to be large. Squaring this number, and squaring the final sum serves to accentuate the relative differences between the different viewpoints. When the system is first set up, a separate activity known as the *manager activity* continually monitors each of the other running activities and evaluates this function for each associated viewpoint. The processing power allocations to these activities are adjusted in proportion to the numerical value of this formula. The Ether command we use for modifying the processing power allocations of an activity is called support-in-rattot. It takes three arguments: an activity, a list of activities (that are children of the first) and a list of non-negative numbers with the same number of elements as the list of activities. The processing power assigned to the parent activity is (re)divided among the children activities in proportion to the numbers in this list Thus, if a factor for a given activity is 0 the activity gets no processing power; if the factor associated with the activity is twice the factor associated with another, then the former activity gets twice as much processing power as the latter. The allocator described is implemented as follows:

```
(defunc square-both-allocator ()
  (support-in-ratios
    parent     start-act
    activities currently-exploring-activities
    factors    (forlist
                 vpt
                 currently-explored-viewpoints
                 (let ((status (quiescent-letter-constraints vpt))
                       (sum 1))
                   (foreach
                     pair
                     status
                     (increment
                       sum
                       (expt (- 10  (length (cadr pair))) 2)))
                   (max (expt sum 2) 1)))))
```

We create a separate activity at top-level called the nenaaer-activity and execute the following to have the allocation strategy continually called asynchronously with the activities doing the actual search:

```
(within-activity manager-activity
    (continuously-execute
      (funcall #'square-both-allocator)))
```

The manager-activity is given a processing power of .1 (meaning it will use, on the average, a tenth of the total CPU time for the entire run).

This scheme gives considerably better performance than the simple parallel solution. It does better than the backtracking solution on some examples with a single processor implementation, although on the average the back trie king solution is more efficient. It is important to understand the source of this improvement. We have a scheme for estimating the likelihood that a running activity will return useful information in a short period of time. We allocate more resources to those activities that we estimate will supply us with information for the least amount of resource expenditure. Assuming our heuristic is reasonable, the average time to complete the search is reduced.

There are three more improvements we have made to the processing power allocation strategy before reaching the final strategy for which we have collected data in the next section. Each will be described in turn.

6.2 Concurrency Factors

We have observed in the allocation strategy discussed thus far that even though activities are running with different amounts of processing power that *wt* related to our estimate of the utility of getting useful information back from them, there still seems to be so many activities running that they tend to thrash against one another. We would like to limit the amount of concurrency so that the running activities can get something done. For this purpose we introduce the notion of a *concurrency factor.* Instead of letting all runnable activities run, we pick the n most promising activities (using the metric above), where n is the concurrency factor, and give only those activities processing power and in the ratios defined by the metric. The optimal value for the concurrency factor is picked experimentally and is discussed below.

The value of the concurrency factor that yields the best result is a reflection of two aspects of the problem: the quality of our heuristic knowledge and the distribution of computational expense for picking bad branches in the search. Obviously if our heuristic knowledge were perfect, i.e. it could always point to the correct branch to explore next, the optimal concurrency factor would be I it should simply explore this best branch. If we are less sure we are about which is the best, more branches should be explored. Also, if the computational cost of exploring a bad branch is always small, a small concurrency factor would be appropriate. If, however, the cost of a bad branch can bo vary largo wo would want to use a larger concurrency factor. With a small concurrency factor we increase the probability that the problem solver will become stuck for a very long time. A limiting case of this is with a search space that is infinite (introducing the possibility of a bad branch that never runs out of possibilities) and a concurrency factor of I. If the problem solver happens to pick one of these branches it will diverge.

Hayes-Roth has noted an analogy with portfolio theory, the purpose of which is to pick an investment strategy that will yield the greatest expected capital appreciation. Uncertainty about the future performance of certain industries and volatility in the market place argue for greater diversification of the portfolio.

6.3 Eettmatiiig Which Assumptions Are Moat Valuable

Our strategy so far has been to use hypothesize-andtest on *one tetter only* in each viewpoint. We sprout one new viewpoint and activity to test the hypothesis that that letter is each one of the digits it could possibly be in the parent viewpoint This is not necessarily the best strategy. By hypothesizing a letter is a certain digit we may learn a lot or a little. We have "learned a lot" if we (I) discover quickly that a viewpoint is contradictory, or (2) cause a lot of constraint propagation activity that significantly increases our evaluation of the new viewpoint. One thing we have observed is that the amount we learn from assuming a letter is a particular digit *does not significantly depend on which digit we use.* In other words, if we assume the letter N is 2 and discover a contradiction, then we are likely to either discover a contradiction or signficantly constrain our solution by assuming N is any other digit on its list of alternatives. To take advantage of this phenomenon the program remembers what happened when it makes particular assumptions. When it creates a new viewpoint to study the result of assuming a letter is a particular digit the result is recorded in the parent viewpoint when it has completed. There are two possible results. If it led to a contradiction this fact is recorded. If it led to a quiescent (but consistent) state it records the difference of the evaluation metric applied to the parent viewpoint and the evaluation metric on the quiescent viewpoint - our estimate of the amount of reduction that is likely to be obtained by assuming this letter to be a digit Our new evaluation metric attempts to take this information into consideration. When assuming a letter L is a specific digit we use the old evaluation metric if we do not have have never assumed L to be a particular digit from this viewpoint: otherwise, we use the average of the evaluations for each of the resultant viewpoint*. We then multiply this figure by the factor I 4 .5 • n where n is the number of letters that we have assumed L to be and determined that they lead to contradictions.

Now that we have a mechanism for taking advantage of information learned by making different assumptions we would like to ensure that a variety of choices are tried at each branching point We will slightly modify the technique for picking the activities to be run at any given time (in accordance with the concurrency factor). Where c is the concurrency factor, we. use the following algorithm to pick the c activities to run at a given time:

1. The activity with the highest evaluation is scheduled.

2. If n < c activities have been selected for running, the n+lst activity is (a) the one with the highest metric if it does not duplicate any of the first n activities in terms of which letter it is making an assumption about for a given viewpoint, or (b) the highest rated non-duplicated activity unless the highest rated activity has a rating at least three times higher in which case we use the highest rated activity. The factor three was picked experimentally and is based on the following argument. There is a certain advantage in having a diversity of letters being tested because this gives us a greater chance to discover assumptions that will cause significant shrinkage by constraint propagation. However, there is also an advantage to running the activity that we have estimated will give us the best result. The factor three is the ratio of estimates for expected gain for which we would rather run the higher estimated test than one that will increase our diversity.

7. An Experiment

In order to test for the existence of a speed-up with concurrency we timed 10 problems using the final parallel algorithm described above for several concurrency factors. The problems tested are:

1) DONALD + GERALD = ROBERT
2) CRIME + TRIAL = THIEF
3) POTATO + TOMATO = VEGIES
4) MIGHT + RIGHT = MONEY
5) FUNNY + CLOWN = SHOWS
6) FEVER + CHILL = SLEEP
7) SHOVEL + TROWEL = WORKER
8) TRAVEL + NATIVE = SAVAGE
9) RIVER + WATER = SHIPS
10) LONGER + LARGER = MIDDLE

They were picked by • trial-end-error process of selecting possible problems and then running them to we if they have a solution. It Is not known whether they have one or more than one solution. The program finishes when it has found one solution. These tests were run on the MIT Lisp machine, a single user machine designed for efficient execution of Lisp programs. The times represent processor run time only and are adjusted for time lost due to paging. The manager activity, which continually monitors the state of the search activities and readjusts processing power accordingly, receives a processing power allocation of .1. We tested with concurrency factors between I and 7. Numbers 2 through 7 each gave some improvement with 4 being the best. Here we report the results for concurrency factors I and 4. Times reported are in seconds:

	concurrency factor = 1	concurrency factor = 4	ratio
1)	377	148	2.69
2)	85	153	.56
3)	187	192	.87
4)	78	246	.32
5)	663	227	2.92
6)	2858	348	8.24
7)	241	112	2.15
8)	78	335	.23
9)	1920	554	2.35
10)	474	212	2.24
total:	8952	2519	2.76

With a concurrency factor of I the algorithm becomes, functionally, a depth-first search. A concurrency factor of 4 represents the value which yields least average run time for the problems examined. Concurrency factors larger and smaller yield higher average values. We caution the reader not to take the numbers too seriously. We only wish to demonstrate that the parallel algorithm runs with some improvement of efficiency *ovtt* the sequential algorithm.

Some interesting facts can be learned by examining the data. Although the parallel solution beat out the sequential solution in only 6 of the 10 cases, these six cases are the ones for which the sequential solutions take the longest. In particular, problems 6 and 9 have show by far the longest times for the sequential solution and the time saving of the parallel solution is considerable. Similarly, for the cases in which the sequential solution finished quickly, the parallel solution tended to take longer. This phenomenon is fairly easy to explain. The parallel solution supplies "insurance" against picking bad branches in the search space. If the sequential solution happened to pick a bad branch (or several bad branches) there was no recourse but to follow it through. Similarly, if the sequential program found a relatively quick path to the solution, the extra efficiency of the parallel solution was not needed.

b\ Conclusions

We have demonstrated that cryptarithmetk puzzles can be solved with a certain increase in average efficiency by the parallel algorithm described over a more traditional depth-first search solution. While this result in and of itself is of little use it does demonstrate a tool that may be of great use in heuristic programming - the use of parallelism to control a heuristic search. Several writers have pointed to the use of *mcto-letd knowledge* (04.

Davis (2J) in controlling a March, Mete-level knowledge is knowledge about how to use the problem solving loob at hand in a way that increases overall search efficiency. The allocation strategies we have examined are mete-level knowledge for cryptarithmetic problems. By allowing a few to run in parallel, and with controllable amounts of processing power we art able to increase the efficiency of the search. Although the increase we gained is not dramatic there is reason to suspect that it would be more significant in more interesting problems. The silt of the search space in these problems is relatively quite small. Thus picking a "bad branch" in the search can't be too catastrophic. With a search space that is much larger, and possibly infinite (as is the case with many interesting problems), a bad branch using a parallel search can only do. a bounded amount of harm, bounded by the quantity of processing power allocated to it. Very similar results have been obtained in speech understanding research projects (14,3] in which competing hypotheses are used to control the allocation of resources for further investigation.

We introduced several concepts that were used in the construction of the allocation strategy. Processing power b allocated in proportion to an. estimate of how likely we are to get useful information out of the exploration of a branch. *Concurrency factors* have been introduced to keep the problem solver reasonably focused. A certain amount of diversity b incorporated in the algorithm to increased the likelihood of discovering assumptions that can be made that will lead to valuable information quickly. Although the only problem we have examined b cryptarithmetic, there is nothing about these general strategies that b specific to cryptarithmetic. They contribute to a general theory of parallel problem solving.

The form of the code is quite simple to write and understand. The algorithm consists of a mixture of constraint propagation and parallel hypothesize-and-test. The programs involve asynchronous, concurrent activities processing different sets of assumptions. Furthermore, the resources allocated to these activities can be altered asynchronously with the execution of the activities.

We have demonstrated that introducing concurrency in the search process does actually increase overall efficiency, in particular it *does no harm.* This lends support to efforts to design a computing system for message passing languages that involves many intercommunicating autonomous processors (e.g. Hewitt (5]). It suggests there is inherent concurrency in search problems that could be gainfully run on multiple processors. We are interested in generalizing the control notions we have developed, such as processing power and quiescence, to be implementable on truly parallel architectures.

9. Acknowledgements

Beppe Atfardi. Roger Duffey, Carl Hewitt Kurt Konotige, David Levitt, Reid Smith, and Barbara White were kind enough to read earlier drafts of this work and have substantially aided the presentation.

I offer my sincere thanks to the lisp machine development group at MIT. Without the superb computing environment available on the lisp machine the program development necessary to carry out this research would have been impossible.

10. Reference*

(I] Bowing, Alan, *Thinghb -- A Cousteoiut-Orkutod Smuktion Laboratory.* XEROX PARC report SSL-79-3, July 1979.

[2] Davis, Randall; *Meta-Rules: Reasoning About Control*, MIT Artificial Intelligence Laboratory memo 576, March 1980.

[3] Erman, L., F. Hayes-Roth, V. Lesser, D. R. Reddy, *The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty*, ACM Computing Surveys, Volume 12, Number 2, June 1980.

[4] Gaschnig, John, *Performance Measurement and Analysis of Certain Search Algorithms*, Carnegie-Mellon report CMU-CS-79-124, May 1979.

[5] Hewitt, Carl, *Design of the APIARY for Actor Systems*, Proceedings of the 1980 Lisp Conference, Stanford, CA, August 1980.

[6] Imai, Masaharu, Yuuji Yoshida, Teruo Fukumura, *A Parallel Searching Scheme for Multiprocessor Systems and its Application to Combinatorial Problems*, Sixth International Joint Conference on Artificial Intelligence, August 1979.

[7] Kornfeld, William, *ETHER -- A Parallel Problem Solving System*, Sixth International Joint Conference on Artificial Intelligence, August 1979.

[8] Kornfeld, William, *Using Parallel Processing for Problem Solving*, MIT Artificial Intelligence Laboratory memo 561, December 1979.

[9] Kornfeld, William A., *PhD thesis*, in preparation, 1981.

[10] Newell, Alan, Herbert A. Simon, *Human Problem Solving*, Prentice Hall, 1972.

[11] Steele, Guy L., Gerald Sussman, *Constraints*, MIT Artificial Intelligence Laboratory memo 502, November 1978.

[12] Steele, Guy L., *The Definition and Implementation of a Computer Programming Language Based on Constraints*, MIT Artificial Intelligence Laboratory TR-595, August 1980.

[13] Waltz, David, *Generating Semantic Descriptions from Drawings of Scenes with Shadows*, PhD thesis Massachusetts Institute Of Technology, 1972.

[14] Woods, William A., *Shortfall and Density Scoring Strategies for Speech Understanding Control*, Proceedings of the Fifth Internation Joint Conference on Artificial Intelligence, 1977.