Constructs and Evaluations Strategies for Intelligent Speculative Parallelism —Armageddon revisited

Adolfo Guzman and Manuel Hermenegildo

Advanced Computer Architectures Program
Microelectronics and Computer Technology Corporation
Austin, Texas.

ABSTRACT

This report addresses speculative parallelism (the assignment of spare processing resources to tasks which are not known to be strictly required for the successful completion of a computation) at the user and application level. At this level, the execution of a program is seen as a (dynamic) tree —a graph, in general. A solution for a problem is a traversal of this graph from the initial state to a node known to be the answer. Speculative parallelism then represents the assignment of resources to multiple branches of this graph even if they are not positively known to be on the path to a solution.

In highly non-deterministic programs the branching factor can be very high and a naive assignment will very soon use up all the resources. This report presents work assignment strategies other than the usual depth-first and breadth-first. Instead, *best-first* strategies are used. Since their definition is application-dependent, the application language contains *primitives* that allow the user (or application programmer) to a) indicate when intelligent OR-parallelism should be used; b) provide the functions that define "best," and c) indicate when to use them.

An abstract architecture enables those primitives to perform the search in a "speculative" way, using several processors, synchronizing them, killing the siblings of the path leading to the answer, etc. The user is freed from worrying about these interactions. Several search strategies are proposed and their implementation issues are addressed. "Armageddon," a global pruning method, is introduced, together with both a software and a hardware implementation for it.

The concepts exposed are applicable to areas of Artificial Intelligence such as extensive expert systems, planning, game playing, and in general to large search problems. The proposed strategies, although showing promise, have not been evaluated by simulation or experimentation.

1. SPECULATIVE PARALLELISM

In general, the execution of a program can be viewed as a (dynamic) graph, often also referred to as the program search space (see Fig. 1.0.). Often, the execution of some parts of the graph is known statically or dynamically to be required in order to successfully find a solution to the problem in hand. Given that enough resources (processors) are available, it is an attractive idea to use them to simultaneously explore different paths of the execution graph. Speculative parallelism (also referred to as "OR-parallelism") is herein defined as the assignment of spare (processing) resources to tasks which are not known to be strictly required for the successful completion of a computation.

This type of parallelism can appear at many different levels in the execution of a program. For example, at a low level, instruction prefetching or lookahead buffers are a form of speculative parallelism. At a higher level, taking both branches of a conditional can also be considered a form of speculative parallelism. At the user level, a given problem can be solved in a "speculative way" if the solution of the problem itself is formulated as a search space which has to be explored in order to find a solution. This last "user level" type of speculative parallelism will be the main subject of this report.

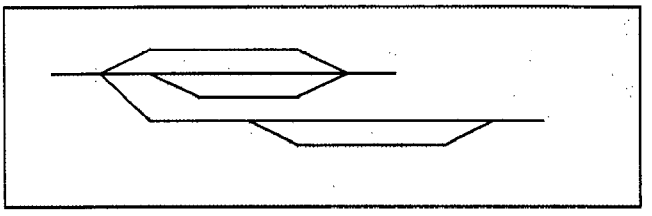


Figure 1.0. An Execution Graph.

1.1. Some Open Issues

The execution of a program in OR-parallel fashion raises a number of issues which need to be addressed:

Source-level constructs and evaluation strategies which make it easier and more efficient to take advantage of speculative parallelism need to be found.

Methods and perhaps hardware support for speculative *work assignment and control* are needed (i.e. so that required work is always given priority with respect to speculative work). Some previous work has been done in this area for example, using "engines" [Zink] or other scheduling mechanisms [Keller, Tinker], and through learning [Lipovski].

Multiple binding environments for program variables appear when alternative paths of computation are explored simultaneously. These have to be handled properly (such as by "Change Control"). Extensive work has been done in this area in support for multiple binding environments through mechanisms such as "hash windows" and binding arrays; implementation of symmetric "cut" and "commit" operators, etc. [Overbeek, Warren, Houri, Tinker, Ciepielewski].

Storage: extra storage has to be allocated for the alternative paths and (perhaps) node "memoing". This represents a classical space-time tradeoff (See Appendix 1 of [Guzman & Hermenegildo]). Semantic paging [Lipovski] can be used in this context.

Application areas should be further identified and characterized.

This report will address the two first issues mentioned above: it will present source-level constructs and evaluation strategies which will hopefully make it easier for the user to take advantage of speculative parallelism. In addition, it will address the problem of assigning resources to the different branches of speculative work generated during execution.

Intelligent *work assignment strategies* are needed because in highly deterministic programs, the branching factor of the execution graph is low, and exhaustive assignment of resources to all possible branches can be a viable solution. In highly nondeterministic programs, however, the branching factor can be very high and a naive assignment of resources will very soon use up all the resources on perhaps paths which hold little promise of yielding useful results.

The search space for such highly nondeterministic programs is generally seen as a tree: at different points several opportunities for "OR-parallelism" (more branches) appear. In general the tree contains several solutions; the problem here is to find one (and then to tell the other processors to stop working). Finding all the solutions requires an exhaustive traversal of the search space, which can be best done with a naive (for instance, breadth-first) strategy.

The work assignment strategies presented in this report will address the assignment of available resources to the branches of the search tree in ways which try to approach a "best-first" strategy (rather than the naive depth-first or breadth-first strategies). In fact, such a best-first strategy can be viewed as a "depth-first" search (i.e. if a tree is drawn with all its branches, the *leftmost* nodes are exploded first) but with the variation that the nodes are reordered at several points within the search so that the *best* are moved to the left (if not physically, at least logically).

Organization of the report is as follows: first, different assumptions and definitions regarding search spaces and strategies are introduced. Then, source-level constructs for expressing OR-parallelism are proposed and a series of parallel *best-first* evaluation strategies for these constructs are described. These strategies are based on the definition of two estimator functions (taken from well known "best-first" techniques): Fpmg, the plausible-move-generator, and Fn, the static board evaluator (alpha-beta pruning is not used herein). Some examples are then given of applications where guided OR-parallelism can offer significant performance advantages over other strategies. Finally some implementation issues are addressed.

2. SEARCH CONCEPTS

In existing systems, search procedures either have to be laboriously described using a non-search oriented language (such as LISP) or they are built into the programming language and then the user or programmer cannot easily specify a different search strategy from the one provided. For instance, Prolog has a built-in search procedure which is depth-first. Generating another search strategy is possible, but the programmer is forced to write it out in detail. Another very popular search strategy is breadth-first. These two are called "blind" strategies, since they do not take into account context information about the nodes of the tree already visited and no attempt is made to measure or assess whether "progress" is made towards the solution. The search continues until a solution is stumbled upon, or the computer time (or user's patience) is exhausted or the tree is completely explored.

In contradistinction to the above blind strategies, if it were possible to measure (albeit imperfectly) whether a node in the tree is closer or farther away from the solution than its predecessors or brothers, then this knowledge of the problem domain could be successfully used to guide the search [Kohli]. In this report, the main emphasis is put on the programmer supplying the needed intelligence or heuristic to measure "progress towards the goal," although the formerly mentioned blind strategies are also encompassed.

The following paragraphs list some assumptions and definitions regarding search spaces and strategies.

2.1. Assumptions

The reader should skimp or skip through these assumptions, as appropriate; they refer to well-known problem solving techniques. Artificial Intelligence texts [Rich] cover the material in this Chapter in greater detail.

Assumption: The *search* space is a graph containing no cycles. That is, once we are searching, the new points visited will never include some of the previous ones *along the current search path* (although they may include duplicates taken from other paths). This assumption is generally true because each step in the search somehow "refines" the criterion of the searcher, who gets "closer" to the end in some mathematical (although not necessarily explicitly specified) manner. For instance, if we are unifying several variables, at one point we may have three "unbound" logical variables; a step is such that one of them gets bound; so, a cycle in the path is impossible. The path gets "closer to completion" as we proceed along it. In most cases, the path ends at a dead end (which we call False, or failure), but in a few cases, the path ends in the answer looked for.

Assumption: The search space forms a tree. It may form a lattice, but generally the different branches of the tree do not communicate with each other, and thus a searcher has no way to know if it or another searcher was already in this point of the search space before. Thus, the lattice is broken into a tree, where perhaps certain nodes located at different branches in the tree are indeed the same; this sameness is ignored. The ignorance is not due to unwillingness or blindness; it is motivated by the current technological tradeoffs in that it takes too much memory to store all points of search space already visited, especially if they were found "not very interesting." *As the technology balance changes in the next 10 years and main storage becomes cheaper, it may be wise to return to the view of the search space as a lattice,* in order to achieve greater gains (speedups) in speculative parallelism. [Guzman & Hermenegildo] address this concept.

Assumption: The following concepts are equivalent: "OR-parallelism," "Speculative Parallelism," "If-then-else Parallelism," "Look-ahead Parallelism," "Case Parallelism," "Eager Evaluation," "Cond Parallelism," "Boolean And Parallelism." Since in some cases the equivalences may not

be obvious, the following paragraphs define "OR-parallelism" and demonstrate the equivalence of the other definitions to this kind of parallelism. Consequently, the remainder of the report will talk only about OR-parallelism, without loss of generality.

2.2.1. Equivalences with OR-parallelism

OR-parallelism refers to the execution in parallel of a disjunction of two or more branches of computation. Of course, success of at least one of the branches guarantees the success of the disjunction. As pointed out above, most programming constructs which can make use of speculative parallelism can be expressed as disjunctions and executed in an OR-parallel fashion.

2.2.2. If-then-else Parallelism

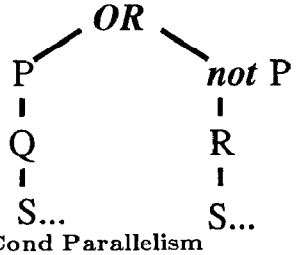
The programming construct

if P then Q else R;
S ...

can be expressed as

(or (and P Q S ...)
(and (not P) R S ...))

which can be executed in parallel as shown below:



2.2.3. Case or Cond Parallelism

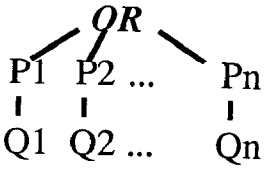
The programming construct

(cond (P1 Q1) (P2 Q2) ... (Pn Qn))

can be expressed as

(or (and P1 Q1)
(and P2 Q2)
...
(and Pn Qn))

which can be executed in parallel as shown below:



2.2.4. And Parallelism

Although AND-parallelism can be better considered to be "strict" (rather than speculative) parallelism, some types of independent AND-parallelism can also be conceivably reduced to OR-parallelism (although a direct support approach, as in the extended Warren Abstract Machine work presented in [Hermenegildo] is probably more efficient):

(and P Q)

can be expressed as

(not (or (not P) (not Q)))

and therefore reduces to OR-parallelism.

Support for AND-parallelism where there are logic variables in the paths explored in parallel requires additional mechanisms [Hermenegildo] to those presented in this report.

2.2. Definitions

Most of these definitions refer to work in search algorithms for artificial intelligence applications, as well as to work on parallel processing and in particular speculative parallelism. For clarity, however, the definitions will be exemplified in terms of game playing.

Search Space. The space of possible computations which may lead to solutions to the problem at hand; somewhere in that space, a solution (a tuple of values) exists. There may exist more than one solution (in this case, only one, the first found, is of interest to us) or none —in this case, "Failure" or "No possible solution" should be reported. Without loss of generality, it is assumed that the search space contains m dimensions, so that each point or node in it (each candidate for a solution) is an (ordered) list or a vector containing m elements. The meaning of these elements is, of course, application-dependent.

Plausible move generator. Given a point in search space which is not a solution, there are in general several possible branches ("moves") which execution can take. The function of the plausible move generator is to provide the "best" moves to make. For example, it can provide a list of all the moves to make, ordered from best to worst, or a list of the best n moves for consideration (ordered from best to worst). What is "best" to the plausible move generator is defined by a function **Fpmg**, which is application-dependent. Therefore, it is provided by the user. It is accepted that this function can only be an *estimation* of the quality of a move: if a perfect Fpmg could be found the problem could be solved deterministically by simply always following the path which receives the highest value from Fpmg. The purpose of Fpmg is to introduce some sort of intelligence (which is application dependent) into the search. However, the computation of Fpmg should require only a very limited amount of resources. Therefore, it is allowed to make some mistakes (to qualify as good a not so promising branch, or not to list the best one in first place)†.
The function **Fpmg** maps (*explodes*) a point or node in the search space into a sequence of points of the form

(p1 p2 ... pp &rest)

which has the following meaning: points p1 through pp are the *best* children of that node; the &rest are "the other children," which are perhaps implicitly represented (i. e., they are not fully developed; they could be represented by a function that *will* generate them, if called); or they may be explicitly represented.

Static evaluator. The static evaluator **Fn** evaluates the "goodness" of a given point of the search space, by computing a value that estimates its distance to the solution. It is also supplied by the user. In this paper it is assumed that Fn is a somewhat expensive computation: while it is easy to

† For instance, in chess, moves that place in check the piece being moved are bad moves; a move that produces check is a good move; a move that nullifies the previous move (The black bishop advanced from square u to square v last time; now it will move back from square v to square u) is considered bad; a move that generates two checks, one to the queen and the other to the king, is good, etc. It is accepted that a plausible move generator makes mistakes some times (it is imperfect); if not, *it* could be by itself the central part of a chess-playing program, rendering search unnecessary. These mistakes must be done, however, "not too often." Generally, one Fpmg is enough, although several can be used: one for the initial game, where the main strategy is to gain room, to dominate the center of the board, to advance pawns and to have a solid back line; another Fpmg for the middle game, and still another for the end-game. This is not precluded, but it requires more work and domain knowledge by the user.

determine whether a node is a solution or a failure, if it is *neither* of these, then Fn could measure how close this node is to a solution.‡ Such measurement will be used, presumably, to decide upon the expansion of this node. If the estimation takes a long time, it is better to go ahead and expand for a while without estimation (by Fn) of "intermediate" nodes. Eventually, Fn will have to be used, in order to prune the search space.

Family. The *family* of a given node are all the descendants, so far, of this node. When implementing this concept, usually a family "resides" inside one processor.

Killing. Killing a node means refusing to consider it further for development of descendants. Usually, a node is killed by Fn at Armageddon time, or by Fpmg at node generation time. Killed nodes will usually be stored in some place [Guzman & Hermenegildo, Martinez], so that if a complete strategy is needed, they can be resurrected. This meaning of *kill* will be used throughout the paper, and is similar to the kill concept in Lisp machines.

Deleting. A stronger version of *kill*: deleting a node means forgetting it forever; it will be impossible to resurrect later.

Armageddon. A global *killing* of nodes where Fn is used to judge, out of the z nodes being present, the (globally) best b survivors. Different families will have, in general, different number of survivors.

Global package. An array containing at least b (but preferably more, say β) nodes, ordered in descending order by Fn, together with other important parameters such as node identification, processor holding it, etc. This global package is formed by gathering information from *all* the families.

Complete Search Strategy. One that, if necessary, exhausts the search space looking for a solution.

Wave: the youngest members of the *d*th generation of a node. Starting from a node, a wave is generated by generating the first generation such node, then the second, etc., and *deleting* all but the *d*th generation nodes. Only *leaf* nodes remain.

2.3.1. Notation used

The following symbols will be used through this report; their meaning will become clear.

"z" is the number of nodes of the current wave or generation.

"m" is the number of coordinates or characteristics of each node or point in the search space.

"b" is the number of best children used as seed for the next wave.

"p" is the number of plausible children that Fpmg generates for each node.

"n" is the number of processors. b is often equal to n or a small multiple of it.

"β" is the number of nodes that the global package contains. $\beta \geq$ b.

‡ For instance, in chess, the static evaluator counts the quantity of matériel (pawns, bishops,...) found at a given point in search space; it also takes into account pieces in check, what pieces dominate the center; how are the kings (in check, castled); whether the queen is pinned down, etc., and gives back a number saying how this particular point (it is called a "board" in chess) is found from the static point of view. Positive numbers are favorable to white, negative numbers to black. Again, the static evaluator is an approximate function (in the sense that it sometimes fails) which is required to work fast. It is, of course, application-dependent. As another example, consider a program which simplifies algebraic equations, such as $(x + x) \rightarrow 2x$, $(y \times 0) \rightarrow 0$, etc. A good way to evaluate the distance from a particular node or point in the search space to a solution (a maximally simplified expression) is to count the number of characters in the current representation, since that is a good approximation of "simple."

2.3. Search Strategies

This Section defines (semantically) several strategies available to the user for performing search; their exact specification and their implementation is covered in the next Chapter. The proposed strategies have the flavor of *develop the best, then kill the worst;* they may be *complete* or not (Cf. Appendix 1).

2.4.1. Built-in Strategies

Several standard parallel search strategies are built into the proposed system and are available for use:

- Depth-first.
- Breadth-first.
- Serial depth-first and serial breadth-first. Only one processor is used.
- Default. The system supplies one of the above.
- Learn. In this case, the system begins with the *default* strategy but subsequently replaces it by an strategy derived by the system itself (how this is done is not discussed in the paper).

In all but last, no killing takes place; the nodes are expanded according to a standard strategy (implied by its name), which continues until a solution is found or no further expansion is possible.

2.4.2. Simple Strategies using User-defined Functions

If the user specifies additional information to guide the search, the question is: how is this information going to be used? As pointed out before, it is assumed that the user will supply two functions: Fn, and Fpmg.

Given those two functions, there are several possible search strategies that employ them, as shown in the next paragraphs. The first two strategies use only Fn or Fpmg. The other strategies try to combine the two functions.

Strategy 1.- From each father, only p children are generated

Fn is not used in this strategy, only Fpmg. Fpmg generates only p *plausible* descendants from each father: those having good likelihood of containing the answer. Each father thus exploded is then deleted. Fpmg is judged capable of deciding at birth whether a newborn is not very promising and can be immediately "killed." In a practical case, Fpmg is called to generate, instead of a fixed number, an average number p of descendants.

This strategy resembles exhaustive search but with Fpmg limiting the offsprings of each node. No Armageddon takes place. Thus, little global communication is needed.

Strategy 2.- The best b of the **dth** *generation (chosen by Fn) are selected*

Fpmg is not used; only Fn. Thus, every node explodes into *all* its children, and then gets deleted. Its children in turn explode, etc., until the (complete) dth generation is reached. [Only the leaves of the tree comprise the dth generation.] Then Armageddon takes place according to Fn, which choses the best b nodes. It can be convenient to make b equal to the number of processors available, but that is not a binding requirement. These best b nodes of the dth generation are "the seed" chosen for further expansion; these cycles *expand and kill* repeat until a solution or failure is found.

Because these b nodes are chosen globally, some families will have several survivors; others, none. So, some amount of node ("seed") exchange is necessary. See Consideration 2 below. Also note that resource consumption, rather than generation counting, can be used as a more practical means to trigger Armageddon. Consideration 1 below deals with this issue.

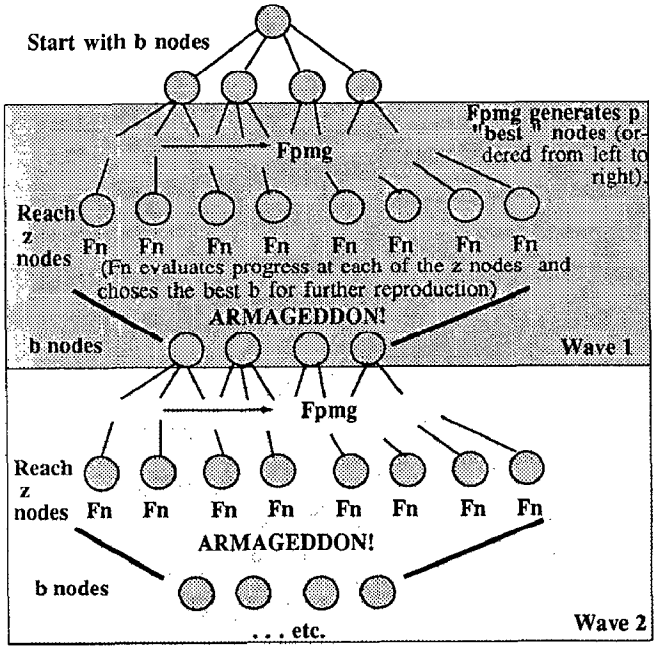


Figure 2.3.1. Wave Computation: Symmetric Strategy

2.4.3. Combination Strategies

Strategies 1 and 2 can be combined to arrive at a generation of *better* children than if only strategy 1 or if only strategy 2 were used. There are several manners to combine them, as well as practical (architectural) considerations that come into play.

The combination strategies are basically of two types: "Wave" strategies and "Continuous" or "Front" strategies. Wave strategies proceed in stages: i.e. there are recognizable (perhaps even system-wide) generation and pruning phases.† "Front" strategies intermingle the generation and pruning stages so that there is a continuous "front" of computation.

Strategy 3.- Symmetric Strategy

Starting with b nodes (See Figure 2.3.1), each one develops its best p children using Fpmg, and these children their own p children, etc., until the dth generation is reached. At this point, there are $z = b(p\dagger d)$ nodes. Once the dth generation is reached, the function Fn is called to cause Armageddon. The evaluation of Fn occurs in parallel, since there are several processors. Each processor, while pruning its own nodes, will take a look at the values that Fn attained in nodes in other processors. Chapter 3 shows a method for performing this lookup. Thus, the z nodes are reduced by Fn to the (globally) best b. These survivors will be the "seed" to again multiply up to the dth generation, etc. until a solution is found. ‡ The best nodes will be unequally spread among the families; thus, some amount of node ("seed") exchange is needed (See Consideration 2).

Consideration 1: Equal time per Family. If (as it is likely to be the case) one processor is used for each one of the b nodes to generate its p†d descendants, in practice some processors will finish before others, and will sit idle waiting for Armageddon. This waiting, however, can be avoided: assume that a processor has reached the dth generation, has used Fn to evaluate each of its dth generation nodes, but Armageddon has not arrived yet. Instead of sitting idle, it uses its spare pre-Armageddon time to explode (using Fpmg) some of its best dth generation children, giving rise to some d+ith generation nodes that may possibly contain the solution sought. Since Fn is a relatively slow function, it is not practical for these "last minute" nodes to be evaluated by Fn. The parents of these "last minute" nodes *will* meet Armageddon, thus having an opportunity to further multiply. Thus, unless these "last minute" nodes contain the solution, they will be re-created again, after Armageddon. This slight drawback applies also to the "last minute" nodes of the Foster Children Strategy (see below), but it is overcome by the Continuous Armageddon Strategy.

In summary, each family has the same time to procreate children. This time is determined by the slowest family to reach its dth generation. Meanwhile, all other families have gone beyond their dth generation. Then Armageddon comes.

Consideration 2: Imported Seed. The possibly uneven distribution of the best b survivors among the families (or processors) makes necessary for some families to *import* some nodes *after* Armageddon has chosen the best b. Section 3.2 explains how this is done, making use of a global communication facility and the *global package* concept.

Strategy 4: Foster Children

This is a slight variant of strategy 3, motivated by Considerations 1 and 2. Once a family has reached its dth generation descendants, has used Fn to evaluate them, and is waiting for Armageddon, instead of using its spare pre-Armageddon time for developing only some of *its* best d generation children, it may also consider for development (importing them, if needed) some of the *globally best* children, taking them away from their families. Thus, children developed by a family after its dth generation will be a mixture of its own children plus imported (or *foster*) children. Since importation or adoption of children from other family takes extra cpu and communication time, the value assigned by Fn to foreign children has to be weighted by a factor (akin to an *import tax*) before comparing it with the value of its own (local, domestic) children. This strategy gives exceptionally good children† the opportunity of adoption by other families.

Strategy 5.- Continuous Armageddon

Like in the "foster children strategy," all children are candi-

† In a family, only the *leaf* nodes —those of the dth generation— are evaluated by Fn; all the intermediate nodes were either *killed* at birth, because they were not worthy (according to Fpmg) of further descendance, or they were *deleted* once their descendants had been generated by Fpmg. Thus, Fn never evaluates a node and an ancestor of such node. It is therefore correct to assert that Fn only evaluates nodes of a *wave*.

‡ Armageddon is a global process; it would be easy to ask each family to kill most of its own d generation children, without taking into account the merits of the children of other nodes. That is *not* Armageddon: in Armageddon only the *globally best* b nodes survive.

† In order to be considered among the best children, these children were already evaluated (through Fn) by their respective families.

dates for adoption by other families. The previous strategies, however, work in cycles or "waves": reproduce, kill, reproduce, kill, ... Nevertheless, it is possible to intermix the reproduction and the killing, as follows. If each family has access to a global package containing the best b nodes *in the whole population* (as judged by Fn) which are waiting for expansion, this information can be used locally to select which nodes to expand next, import, or kill.

Expansion. Normally, a processor begins in expansion mode; given a node, it will generate (using Fpmg) a given number (such as p†d) of descendants.

Evaluation. After expansion, a processor proceeds to evaluate (using Fn) all its created descendants; each of the evaluated nodes tries to gain access to the global package; presumably a few succeed, displacing from the global package other less valuable nodes (which then will have to be killed by the processors owning them). Nodes which did not succeed in getting into the global package are killed, as well as nodes expelled from the global package.

Selecting the next node for expansion. The processor now selects the best node from the global package for expansion, perhaps affected by an import tax. The node is *deleted* from the global package. Then, the processor goes to "expansion" state again.

As shown in figure 2.3.2, at any point in time there is a "front" of nodes being expanded in the tree, and others which are waiting for expansion and are tagged with their "goodness". In summary, after a number of nodes or generations, a processor evaluates Fn for all its children (P1 in Fig. 2.3.2) and it selects the most promising node in the system to expand next (this could be one of P1's own nodes, or a foreign node, as in Fig. 2.3.3).

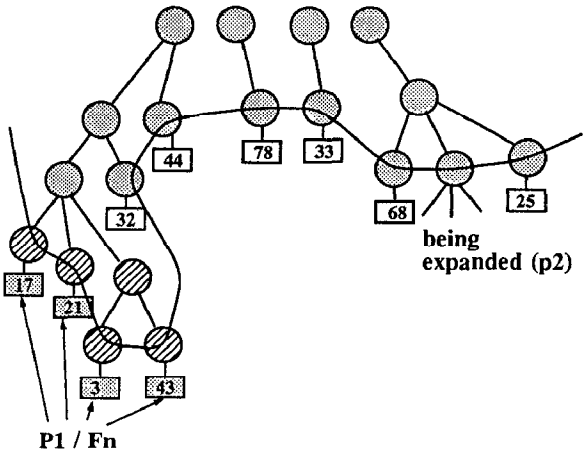


Figure 2.3.2. Strategy 5: Simultaneous Evaluation and Expansion

3. IMPLEMENTATION ISSUES

The characteristics that the different functions and parameters should have are described here, as well as the implementation of some of the previously discussed strategies.

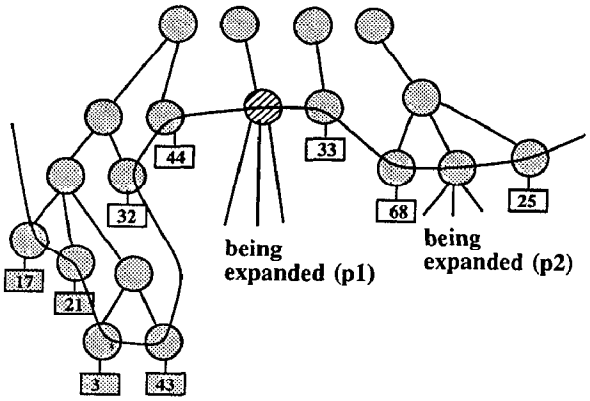


Figure 2.3.3. Processor P1 expands a Foster Child

3.1. Specifications needed

It is time to say how the user gives and constructs the "intelligent" functions. The exact syntax is left somewhat undefined; a few examples are given in Lisp.

3.2.1. Specification of Fn

Fn is a function of one argument that maps a node into a number: the smaller this number is, the closer this node is to a solution. A solution need not have a value of Fn = 0; these values are used only for relative comparison among nodes. For example,

(defun Fn (p) (string-length (car p)))

will measure the length of the string which is the first (and only) constituent of point p. Such Fn will be useful, for instance, for an algebraic simplification program (See footnote "‡" in Section 2.2).

3.2.2. Specification of Fpmg

User-provided, Fpmg maps a node into a list of its best children and a &rest that could be implicit or explicit (Section 1.2 "Definitions."). Thus, Fpmg should generate at least b (but preferably more) nodes starting from the given point; it should order these so that the best b are (in descending order of worthiness) in its result (a list). It is up to Fpmg how the nodes are generated, and what criterion is used for ranking them. For example, given a node of the form

(1 1 3 X Y)

which represents some point with five natural number coordinates (two of them not yet specified), Fpmg may return

((1 1 3 1 Y) (1 1 3 2 Y) (1 1 3 X 1) (1 1 3 X 2)
'rest (1 1 3 (>2) Y) (1 1 3 X (>2))

meaning that the first four nodes are the best children of their parent, and the rest is a formula for further generation of "not so good" children. In this example, Fpmg considered "better" those nodes with small natural numbers. The list (> 2) stands for a natural number greater than two; it will be "expanded" if necessary by the completion strategy; it is an example of a representation for a "rest." Fpmg should generate a "rest" if a complete strategy (Cf. Appendix 1) is required.

3.2.3. Specifying other Parameters

The following parameters need to be specified before an intelli-

gent search can start:

- The *search strategy,* to be chosen from Section 2.3, as well as the parameters that apply to that particular strategy (Cf. Section 1.2 "Definitions"), such as b, p, d, etc.
- The starting point or node.
- The function that determines if a node is a failure.
- The function that determines if a node is a solution (success).
- Whether this search is going to be complete or not.
- If the search is complete, how to handle the **rest** of both Fpmg (see previous paragraph) and those fertile children killed by Armageddon (Cf. previous chapter).

3.2.4. Starting the Search

In order to start the search, the user will simply say

(parallel-search initial-node Fn Fpmg ...),

where parameters not supplied acquire default values.

3.2. Implementation of Some Strategies

While the system is not implemented, some thought has already been given to the implementation of several functionalities, including the most promising strategies. Strategies 2, 3 and 4 use simple Armageddon; Strategy 5 uses Continuous Armageddon. It is sufficient to explain, then, these two killing procedures.

3.3.1. Simple Armageddon

Depending somewhat on the number of processors and type of architecture being used (*shared cluster* [Guzman & Krall], *global memory* [Guzman], or another type —such as *message passing*— as well as its bandwidth and latency), this killing procedure can be implemented in slightly different manners. Two will be described; several others are easy to imagine. The common idea is that a global ordered package of the best b nodes considered as candidates for surviving Armageddon is passed (together with ancillary information such as the node identification) around the processors, which have an opportunity to update it, until a final package is reached, containing the surviving nodes. This package is passed back to the processors; each of them takes one (or a fixed small number) of the survivors for further reproduction by Fpmg.

1. Serial build-up of the best b

A kind of serial sort-merge. If the processors are numbered 1 through n, then processor **1** builds a package containing its best b nodes (of processor **1**), and passes such package to processor 2, in a daisy-chain fashion. In general, processor i merges *its* best b nodes with those of the package, and the resulting package is passed to processor i+1. The package arriving back to processor 1 obviously contains the globally best b nodes.

Then, processor 1 picks one (or b/n, if b > n) node from the package, *deletes* such node(s) from the package, passes the smaller package to processor 2, and starts to work in developing the chosen node(s). Each processor choses b/n nodes from the package and passes the remainder to its successor. In deciding which node(s) to select from the package, a processor will prefer a local one: one which resides within the processor, or in a nearby processor. The topology of the interconnection fabric among processors dictates which processors are "near" which others.

2. Simultaneous build-up of the best b

The n-1 merge operations could also be done in parallel; for instance, each even-numbered processor (2, 4, 6, ...) could merge its best b candidates with those of the preceding processor (2 merges data from 1 and 2; 4 merges data from 3 and 4, etc.); then, each processor whose number is a multiple of four (4, 8, 12, ...) merges its result with the result obtained by the preceding processor (4 merges results from 2 and 4; 8 from 6 and 8; etc.), and so on. The result of the final merge could also go down in this binary-tree fashion.

3. What to do while waiting

Some time passes since a processor submits its best b nodes to Armageddon until the same processor decides what node(s) to develop next. Instead of idling, the processor could expand "tentatively" its most promising nodes; perhaps it will hit the solution while waiting for the package to come back. Strategies 4 and 5 referred to this.

3.3.2. Continuous Armageddon

To implement this strategy, it is necessary to have a system-wide array (containing the global package) to which access can be gained from any processor. A processor may try to read the highest node in the global package, or it may try to write into the global package a node with a given value **v**. A write operation will only succeed if

$$\mathbf{v} > \text{Min}$$

where Min is the smallest of the values of the nodes in the global package.

1. The necessary hardware

The inclusion of a MAX network simplifies the operations with the global package.

The System-wide Array. This hardware holds the global package, comprising β elements, ordered. It contains also two registers where the values Max and Min are stored. Of course, such a facility is readily available in a shared memory machine.

The MAX network. This network can be implemented using a wired-or bus (described below), through which all the processors could be simultaneously trying to write values **v1**, **v2**, ... into the system-wide array. Only the value of one of the processors, that with the maximum value of **v**, will be chosen. If this value is larger than Min, it will be inserted in the proper place in the system-wide array, through hardware supplied with such array. In practice, many processors will not be trying to access the system-wide array, but only a few. When a new node gets inserted in the array, the lowest node goes out, and the system-wide array has to inform its proprietor that such node should be killed.

For reasons that will become clear soon, the MAX network is time-multiplexed (in a cyclic fashion) among the following functions: 1a. Select highest processor; 1b. Give to it the best node of the system-wide array; 2. Select node with value **v** for possible insertion in the system-wide array; 3. notify to certain processor that a particular node of it should be killed because it was expelled from the system-wide array. If one of these cycles is not used, it is wasted.

2. The operations

A few operations are necessary for using of the system-wide array through the MAX network; this use is simultaneous by all the processors.

Reading the best node. This is accomplished by using the MAX network to select the highest processor (highest processor number) trying to read the best node; in the next cycle of the MAX network, such reading does indeed take place. This uses cycles 1a and 1b.

Trying to write a node with value **v** *into the system-wide array.* This is done by using the MAX network during cycle 2, in the manner described under "The wired-OR bus" below.

Killing the lowest node. If necessary, the system-wide array notifies the corresponding processor that one of its nodes should be killed. This happens during cycle 3; the array uses the bus directly to access the needed processor.

3. The wired-or bus

This bus should have as many data lines as the value of **v** has bits, plus a few control lines to specify what cycle (see above) it is in. Each processor wishing to write a value **v** in it (during cycle 2) writes the bits of **v** serially, starting from the most significant one. After writing a bit, the processor *reads* such bit; if it is different than what it wrote (meaning that some other processor has written a bigger bit), it should withdraw from additional writing tries on the remaining lower significant bits of **v** during this cycle 2. In this manner, only the highest value of **v** gets written into the wired-or bus.

At the end of cycle 2, each processor knows whether he succeeded or failed in writing its value to the system-wide array. Whether a processor succeeds or fails, it keeps trying to write its best node into the global package. A processor that succeeds now will try to write what used to be its *second* best node, but is now its best node. Also, as a processor develops more nodes and evaluates them with Fn, its own list of best nodes changes. Thus, at any moment, the system-wide array contains the globally best β nodes.

4. CONCLUSIONS

The previous sections have addressed the issue of efficient management of speculative parallelism. It was pointed out how in highly non-deterministic programs the search tree to be explored can be very large, so that a naive assignment of resources will result in very inefficient execution. The use of resource assignment algorithms based on best-first search strategies and the addition to the user language of primitives for the use of such algorithms were proposed. Such primitives allow the user to indicate when intelligent OR-parallelism is to be used and to specify the functions that define "best" in the best-first search algorithms. Two such functions, Fn and Fpmg, were proposed for this purpose. Several parallel evaluation strategies for performing resource management in an OR-parallel search based on the use of those functions were also proposed. "Armageddon", a pruning method based on global knowledge, was introduced. Both software and hardware implementations of "Armageddon" and the global knowledge maintenance system associated with it were proposed. It is believed that a system based on the techniques presented in this paper will make it possible for the naive user to perform search over a large search space, in an efficient way, and achieving speedup through the use of OR-parallelism without having to be concerned with issues such as processor synchronization, pruning of computations, scheduling, resource management, etc.

The next step in this work would be to implement (or simulate) and evaluate (a) some of the proposed strategies, perhaps with modifications suggested by the evaluation; and (b) to do likewise with the corresponding hardware associated to the most promising strategies. these evaluations must be done against real symbolic applications involving search.

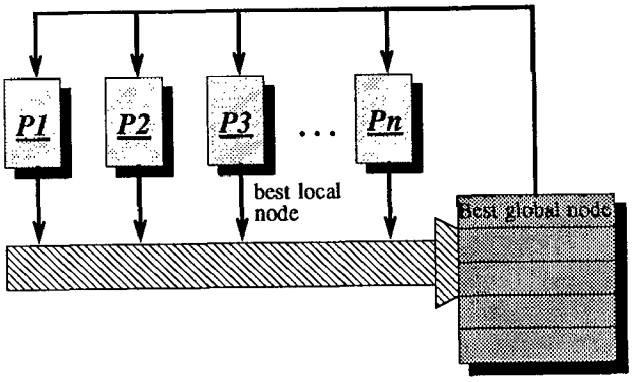


Figure 3.2. Architecture for Continuous Armageddon

Appendix I. Complete Strategies

The best-first strategy is complete if nodes killed are resurrected when the best nodes do not contain a solution. The solution now has to be sought in parallel among the resurrected or their descendants, most likely using again a best-first approach.

I.1. Complete and Incomplete Strategies

If, for some reason (lack of time to make a complete search; lack of memory to store *killed* nodes or pointers to "&rest" of expansion lists, availability of alternate search or procedural strategies, etc.) it is desired not to complete the search, then "failure" can be reported after the best nodes are found sterile —that is, just before resurrection.

Thus, the best-first strategy is *complete* if killed nodes are resurrected (and searched, expanded, etc.) if necessary; and it is *incomplete* if killed nodes are considered deleted: resurrection is not allowed.

1.2.1. Incomplete Search

Having failed to find a solution among the best nodes (non-resurrected nodes), this alternative reports a failure. It should be interpreted as "the best nodes do not contain the solution." The user probably wants to try other strategies or algorithms or alternatives in seeking the solution; for the *best first* approach, he gives up.

1.2.2. Complete Search

For this completion, the *killed* nodes (comprised by (a) the remainder or &rest of the nodes which were not generated by Fpmg; and (b) the remainder of the nodes that were not selected by Fn as worthy of further expansion) are resurrected now. A solution is sought among them, or among their descendants. Search continues, perhaps in a best-first manner again. Probably this time the solution will be found. Else, when it is time for the *second* resurrection, the user has the same two choices: either to *give up* or to complete the search. If he gives up, he has used an *incomplete* search.

1.2.3. Getting less and less incomplete

The best-first strategy proceeds in cycles. Starting from one (or b) node(s), and going through a series of Armageddon cycles, either a solution is found or sterility (no further descendants) is met. It is then time for Resurrection of all the nodes killed. At each resurrection, the user has the choice to complete or to give up the search. Each resurrection is a step towards completion, towards a complete search strategy. Thus, resurrection [a major cycle] comprises many Armageddons [minor cycles].

References

Batcher, K. E. Sorting Networks and Their Application. *AFIPS Conf. Proc.* 32:307-314, 1968.

Bowen, D. L., Byrd, L., Pereira, L. M., Pereira, F. C. N. and Warren, D. H. D. *Prolog on the DEC system-10 User's Manual.* Technical Report, Dept. of Artificial Intelligence, Univ. of Edinburgh, Oct. 1981.

Ching, Wai-Mee. Evon: an extended von Neumann Model for Parallel Processing. *Proc. 1986 Fall Joint Computer Conf.* 363-371. IEEE Catalog # 86CH2345-7.

Ciepielewski, A. and Haridi, S. Control of Activities in the Or-Parallel Token Machine. In *1984 Int'l. Symp. on Logic Programming,,* 49-58. IEEE Computer Soc. Press.

Guzman, A. AHR: a Parallel Computer for Pure Lisp. In *Parallel Computation and Computers for Artificial Intelligence,* J. S. Kowalik (ed.), Kluwer. 1987 (In press).

Guzman, A. and Hermenegildo, M. *Constructs and Evaluation Strategies for Intelligent Speculative Parallelism - Armageddon revisited.* MCC Report PP-220-87. This report is an expanded version of the Conference paper of the same name.

Guzman, A., Krall, E. J., McGehearty, P. F., and Bagherzadeh, N. Performance of Symbolic Applications on a Parallel Architecture. MCC Technical Report PP-163-87. Also submitted to the *International Journal of Parallel Programming.*

Hermenegildo, M. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* Ph. D. Thesis, Dept. of Comp. Science, U. of Texas at Austin, TR-86-20, 1986.

Houri, A. and Shapiro, E. *A Sequential Abstract Machine for Flat Concurrent Prolog,* Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel.

Keller, R. M., Lin, F. C. H., and Tanaka, J. Rediflow Multiprocessing, *Digest of Papers, Spring COMPCON '84,* February 1984, IEEE Computer Society.

Kohli, M. *Controlling the execution of Logic Programs: Specification and Compilation of Control Knowledge.* Ph. D. Thesis, U. of Maryland, College Park, 1987.

Kumar, V., and Kanal, L. N. Parallel Branch-and-Bound Formulations for AND/OR Tree Search. *IEEE Trans. on Patt. Analysis and Mach. Int.* 6:768-778, Nov. 1984.

Lipovski, G. J. and Hermenegildo, M. V. B-log: A Branch and Bound Methodology for the Parallel Execution of Logic Programs. *Proc. 1985 Int'l Conf. on Parallel Processing,* IEEE Computer Society.

Martinez, T. *Smart Memories: Potential Functionality and a Proposed Architecture.* MCC Technical Report PP-178-87.

Overbeek, R. A., Gabriel, J., Lindholm, T., and Lusk, E. L. *Prolog on Multiprocessors,* Argonne National Laboratory, Argonne, Ill. 60439.

Rich, E. *Artificial Intelligence,* 1983, McGraw-Hill, N.Y.

Tighe, S., Zink, K., Brice, R., and Alexander, B. *A flexible approach to the study of graph reduction architectures.* MCC Technical Report PP-295-86. Oct. 86.

Tinker, P. and Lindstrom, G. A performance-Oriented Design for OR-Parallel Logic Programming. *Proc. 4th. Int'l Conf. on Logic Programming,* 1987. MIT Press.

Warren, D. H. D. OR-Parallel Execution Models of Prolog. in *Proceedings of TAPSOFT 87.* Springer-Verlag, 1987.

Zink, Ken, *Engines as a Method of Controlling Eager Evaluation in Graph Reduction,* MCC Technical Report PP-393-86.