Report 84-14
Stanford -- KSL

A Variable Supply Model for Distributing Deductions.
Vineet Singh, Michael R. Genesereth,
May 1984

card 1 of 1

Heuristic Programming Project
HPP-84-14

May 1984

A Variable Supply Model

for

Distributing Deductions

Vineet Singh
Michael R. Genesereth

Department of Computer Science
Stanford University
Stanford, California 94305

Abstract

Multiple processors can be used to achieve a speedup of a backward-chaining deduction by distributing or-parallel deductions. However, the actual speedup obtained is strongly dependent on the task allocation strategy. Also, communication cost can be a significant part of the overall cost of a deduction. For the multiple processor scenario used in this paper, processors with replicated databases on a broadcast network, a variable supply model (VSM) is presented. VSM represents an infinite class of strategies with varying communication requirements. All strategies in VSM use a task supply protocol (ESP) that works better than other supply protocols proposed. The two extremes of the strategies in VSM are a supply-driven strategy, with the highest communication requirements and a demand-driven strategy, with the lowest communication requirements. The best choices in the two cases when the available network bandwidth is very plentiful and when it is severely limited are the supply-driven strategy and the demand-driven strategy respectively. The utility of VSM lies in the easy and powerful way it provides in selecting a strategy that works satisfactorily given certain communication constraints. Implementation results from a multiple processor simulation and theoretical results are used to demonstrate the ideas.

Table of Contents

List of Figures

BLANK PAGE

1. Introduction

Parallelism has been identified as a key to future high-performance reasoning machines - the fifth generation of computer systems [14]. Multiple processors must be made to cooperate to speed up a computation. Clearly, there is a need to identify parallel components of reasoning computations and there is a need to build multiple processor systems. In addition, there is a need to map parallel computations on to multiple processors keeping in mind the constraints of communication. This last need is being largely ignored at the present time and may become the bottleneck in achieving high performance from multiple processors. The purpose of this paper is to address the last need (i.e., to propose methods for task allocation that work well in the presence of communication constraints).

Task allocation in a multiple processor system strongly affects the overall speedup obtained for a parallel computation [6]. Also, communication cost can be a significant part of the cost of the computation. Therefore, a good task allocation strategy is needed and can be obtained only if both processing costs and communication costs are kept in mind.

The type of computation being considered in this paper is a backward-chaining deduction [3] with no side-effects and the parallelism employed is or-parallelism. In addition, all the processors can do backward-chaining deductions[1] and are connected by a broadcast[2] communication mechanism.[3] The goal is to complete the deduction in as short a time as possible. The task allocation strategies presented in this paper do not assume any knowledge of the domain of application (i.e., the database of facts and rules may be used only syntactically). The task allocation strategies can, however, use extra information to improve their performance.

The hope is that this work in task allocation for a case with relatively simple communication requirements will serve as a sound basis for task allocation strategies for cases with more complex communication patterns. Also, a broadcast communication network was chosen so that it would be possible to focus on issues of bandwidth constraints without the additional complexities introduced by unequal inter-processor message costs (e.g., localizing computation in neighboring processors).

Previous approaches to parallel task allocation [8] will not work well in this domain because of

---

[1] They may, however, work at different speeds.

[2] The network need not use a physical broadcast medium; "broadcast" just refers to the capability of sending a message from one processor to all others in a reasonably efficient manner.

[3] The multiple processor scenario applies to both multiple processors connected by a broadcast local area network (like the CSMA-CD Ethernet [12]) or multiple processors connected by a network on a single chip as suggested by Ullman [19].

assumptions that are not reasonable here. Most techniques can be eliminated as inapplicable because they assume that all tasks to be allocated are known beforehand along with their processing requirements. Another large class of techniques can be eliminated because communication cost is not considered or it is inaccurately modeled. More likely contenders will be considered later in the paper.

The contributions of this paper are mainly two-fold. First, the paper presents the Variable Supply Model (VSM) of task allocation. This model allows flexible usage of the inter-processor network bandwidth. At the two extremes for communication requirements are a supply-driven strategy at the high end and a demand-driven strategy at the low end. The model includes an infinite set of strategies with communications requirements ranging between the two extremes. All strategies in the model use the Efficient Supply Protocol (ESP) as the protocol to distribute tasks to remote processors. It is not necessary for the model to use ESP as the task supply protocol; the protocol just happens to be efficient for the present problem. By using a different task supply protocol, other task types besides backward-chaining deductions can be handled. Also, the model includes work-load balancing as a natural component of the possibilities allowed. Moreover, the paper includes hints to take advantage of any extra information about tasks that may be available.

The second main contribution of the paper is ESP, a protocol for transferring a task from one processor to another that is more efficient than the *announcement-bid-award* protocol of Contract Net [7] and Enterprise [9]. The protocol is useful for tasks that are side-effect free or for which repetitions of side-effects are acceptable; these computations can be dynamically distributed to processors on a broadcast network. Extensions to other processor interconnection structures are also possible. Another attractive feature of ESP is that it can handle failures from a realistic failure model of the multiple processor architecture.

This paper is organized as follows. Section 2 explains how to view a backward-chaining deduction as a tree of or-parallel tasks. Section 3 contains a detailed description of the multiple processor architecture. Section 4 then describes VSM and how it can be used to control communication. Section 5 describes ESP and how it can be used as an efficient task supply protocol for all the strategies in VSM. Sections 6 and 7 contrast the results of using the two extremes of VSM - a supply-driven strategy and a demand-driven strategy. Useful extensions to VSM and ESP and how they might fit into the current framework are described in Section 8. Sections 9 and 10 contain directions for future work and the conclusions respectively.

2. Viewing a Backward-Chaining Deduction as an Or-Parallel Computation

The Handbook of Artificial Intelligence [3] explains backward-chaining as an inference mechanism for automated deduction. Further, the handbook [2] explains and-or trees as a problem-reduction representation and this can be used to represent the problem of proving a proposition by backward-chaining as well.[4]

In this paper, only or-parallelism is used; no and-parallelism is exploited. Therefore, an *or* tree to represent a backward-chaining deduction, as described below, is of more interest than an *and-or* tree. Backward-chaining is used in the context of rule-based systems in which the data base consists of a dynamic set of propositions and the rule base consists of a static set of "rules". In this paper, even the data base is kept unchanged as the deduction proceeds. All of the propositions in the data base are literals (i.e., either atomic propositions or negations of atomic propositions[5]). All propositions in the rule base are required to be written in one of the forms shown below, where alpha, beta, and $alpha_1$...$alpha_n$ are all literals.

(IF alpha beta)
(IF (AND $alpha_1$...$alpha_n$) beta)

Similarly, a goal to be proved must also be a literal. All propositions in the data base, rule base, or in the goal may contain variables.

Each node in the *or* tree can be represented as a tuple of two sets: a set of goals and a set of bindings. The top-level node's set of goals contains one literal and its set of bindings is empty. If the set of goals of a node in the tree is non-empty, its children can be obtained in the following three ways: The first way takes one of the goals in the goal set and backward-chains with all possible rules. A child is created for each of the rules that can be used to backward-chain from the parent. A child's goal set is obtained from the parent's goal set by removing the goal that was backward-chained on, adding the antecedents of the rule applied, and applying the unifier from the backward-chaining to the resulting goal set. The set of bindings of the child is obtained by applying the unifier to the set of bindings of the parent and adding the unification bindings to the set of bindings of the parent. The second way to produce a child is to unify a goal from the goal set of a parent node with a proposition

---

[4] The reader should understand backward-chaining and and-or trees before proceeding any further.

[5] An atomic proposition is an expression consisting of an n-ary relation symbol and n terms enclosed in parentheses.

in the data base. In this case, the goal set of the child is obtained by removing the unified goal from the goal set of the parent node and applying the unifier to the resulting goal set. Again, the unifier is applied to the bindings of the parent node and added to the set of bindings. The third way to obtain a child is if the goal selected from the goal set of the parent cannot be backward chained on and the goal does not unify with any proposition in the data base. In this case, the child's goal set as well as its set of bindings are empty.

Clearly, a leaf is obtained if the goal set of a node is empty. Now, the leaf represents a positive result if the set of bindings is non-empty and it represents a negative result if the set of bindings is empty. It is possible for someone to be interested in all the positive results of a deduction tree or just one of the results of the deduction tree.

Any goal set to be proved may be referred to equivalently as a task. A unit deduction refers to the expansion of a node into its children in the *or* deduction tree. The result of a deduction is the binding list that satisfies the proposition in question, if only one positive result is desired, or it is the list of binding lists that satisfy the proposition, if all positive results are desired. The result is nil if no binding list satisfies the proposition.

Notice that each of the subtasks of a task can be solved completely independently. No communication is required among the subtasks. Also, notice that the grain of the tasks in the deduction tree can range all the way from 1 unit backward-chaining deduction to an arbitrarily large size. Moreover, backward-chaining deductions are side-effect free.[6]

3. The Multiple Processor Architecture

The multiple processor architecture is depicted in figure 3-1. It consists of many identical sequential processors connected to a broadcast network. Each processor contains a certain amount of memory that is private to it; a processor may not directly access another processor's memory. Further, each processor has a unique name. Communication between processors is only via arbitrary-length messages sent on the broadcast network.[7] The arrows shown in the figure represent the direction of message flow and nothing else.

---

[6] They are side-effect free at least in their pure form. One could have side-effects by allowing caching or by some other means.

[7] If the network is an Ethernet, for example, the arbitrary-length messages will have to be built out of the fixed-length packets supported by Ethernet.

Clearly, the communication network connecting the processors has a bandwidth limitation. In general, this is true for any inter-processor communication network although it may be higher than what it is for a broadcast network. Depending on the number of processors and the rate of generation of messages from each processor, the bandwidth limit may or may not be a bottleneck.

All messages received for a processor are placed in its input queue (in the order of receipt of the messages). The messages can then be read by the processor in the order of receipt at any time. Messages to be sent from a processor are placed by the processor in its output queue and they are then broadcast as soon as possible on the network. Assume that both the broadcast of messages from the output queue and the insertion of messages into the input queue from the network are done by low level hardware and do not require any processing by the sequential processor.

A message can be either point-to-point (i.e., sent from one processor to one other processor) or broadcast (i.e., sent from one processor to all other processors). Each message includes a destination field that is either a processor name that is unique (see above) or a special tag to indicate that it is a broadcast message.

Processor clocks cannot be completely synchronized because they are physically separated and the task allocation strategies do not require them to be completely synchronized. However, a close synchronization is desirable and this can be obtained by a standard distributed clock synchronization algorithm [10]. This will not be discussed any further.

A couple of other requirements of the task allocation strategies are as follows. The first is to have globally unique task names. Globally unique task names can be obtained by combining locally unique names and the unique processor name of the processor that originated the task in the task name.

The second requirement is to be able to resolve a tie between two known processors in a globally self-evident fashion. A tie may arise between two processors as will be seen later when two processors contend for the same task at the same time. Again, the unique processor names can be used to break the tie. Timestamps, created out of time and the processor name, can be used to break the tie in one comparison. The timestamps used in this paper[8] obey the property that a timestamp with a lower (greater) time than another timestamp is always lower (greater) than the other. When the

---

[8] A timestamp is composed of the time in its higher order bits and a fixed number of bits for a unique processor number in its lower order bits.

times are equal, the timestamp with the lower numbered processor[9] is lower.

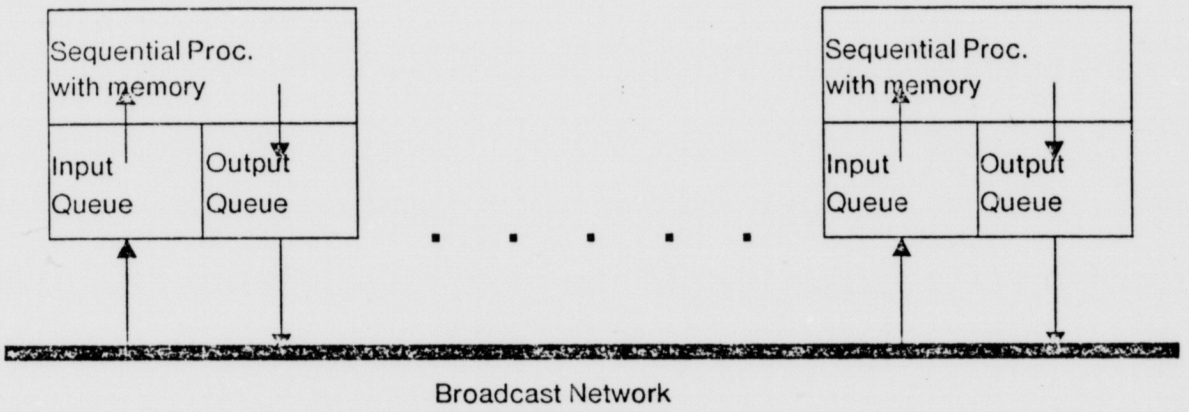


Figure 3-1: The Multiple Processor Architecture

4. VSM - a Variable Supply Model

VSM defines a class of strategies with varying communication requirements. It will be shown later that the strategy with the highest communication requirements will work best when inter-processor communication bandwidth is very plentiful relative to the message traffic that can be exchanged; at the same time the strategy with the lowest communication requirements will work best when the inter-processor communication bandwidth is very low. The utility of VSM lies in providing a unifying framework for an infinite set of strategies with varying communication requirements, in the ease of selection of these strategies, and in the demonstrated usefulness of the two extreme strategies. The use of intermediate strategies for intermediate communication conditions seems likely but is left unexplored in this paper.

VSM lays down a specific internal organization of the sequential processor with memory that was shown in figure 3-1. This organization is shown in figure 4-1. **Current Task** is the task being worked on by the processor, **Actual Work Set** is the set of tasks that the processor is committed to executing, and **Potential Work Set** is the set of tasks that no processor is committed to perform.

---

[9] A unique processor name can always be converted into a unique processor number.

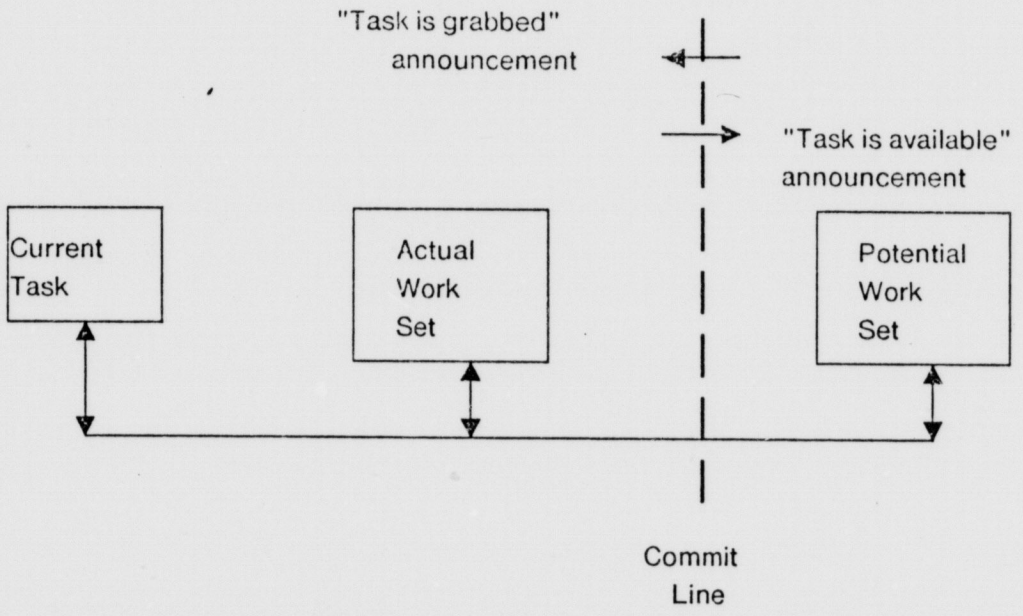


Figure 4-1: The Variable Supply Model

The lines with arrows show the directions of transfer of tasks. Any tasks that flow from left to right across the hypothetical **commit line** must be announced to other processors as being available so that they may place the task in their **potential work sets**. Similarly, any tasks that flow from right to left across the **commit line** must be announced as being grabbed by this processor so that other processors may not also grab the same task.

Ideally, the **potential work sets** of all the processors are the same at all times and each task is grabbed by one and only one processor. However, because of non-zero message delays, more than one processor may attempt to grab a task. A protocol understood by all the processors is required so that multiple grabs are resolved, results are properly handled, and so that tasks are killed when they are no longer needed. An efficient protocol (ESP) to handle these problems is described in the next section.

In VSM, the mechanism for controlling the amount of communication lies in the fact that the tasks in the **actual work set** and the **potential work set** can be arbitrarily balanced. As will be seen later, if all newly generated tasks are placed in the **potential work set** and grabbed only when absolutely needed, one obtains the supply-driven strategy with the highest communication

requirements. Similarly, if all newly generated tasks are placed in the **actual work set** and only placed in the **potential work set** when absolutely needed, one obtains the demand-driven strategy with the lowest communication requirements. This will be explained in greater detail in sections 6 and 7.

For all the strategies in VSM, if a new **current task** is needed, it is picked out of the **actual work set** if it is non-empty. Otherwise, the **current task** is picked from the **potential work set**.

5. ESP - an Efficient Supply Protocol

This section first presents a rough description of ESP followed by a detailed description. Next, the domain of application of ESP, which is larger than the present application alone, is described. Finally, some previous work is contrasted with the work presented here.

5.1. Informal Description of ESP

Section 2 described how a deduction may be partitioned into or-parallel deductions. If the number of or-parallel tasks at a processor is more than one, the processor may keep one to work on and try to distribute some or all of the surplus tasks to other processors. Surplus tasks not to be distributed are placed in the **actual work set** of the processor. Each surplus task that needs to be distributed can be announced in a broadcast **new-task** message as a new task available for execution to other processors. All new tasks announced are placed in the **potential work set** at all processors (including the originating processor). Any free processor can start working on one of the new tasks. To avoid duplication of computation, it is necessary for a processor to announce its intention to start working on a task by sending out a broadcast **grabbed-new-task** message. Since the tasks are side-effect free, a processor may begin to start working on a task as soon as it sends out a **grabbed-new-task** message. A task is removed from the set of potential tasks at any processor when a **grabbed-new-task** message is received for it. The "best" processor among the ones submitting **grabbed-new-task** messages for a certain task gets to continue executing it; the "losers" abort their computations of the task and are free to grab some other new task. If there is a tie between processors trying to "grab" a task, it is broken by the timestamp mechanism. Note that if the processors are identical, there is an automatic tie when multiple processors attempt to grab the same task and, therefore, all multiple "grabs" are broken by the timestamp mechanism. However, if processors have unequal speeds, a fast processor may grab later and still win (see section 8.3).

It is not necessary for a processor to grab a task only if it needs to execute it immediately; a task may be grabbed and placed in the **actual work set**. Also, tasks moved from the **actual work set** to

the **potential work set** are treated identically to newly announced tasks.

The result of a deduction is returned to the processor that originated that deduction in a point-to-point **done-task** message.[10] Within the processor where the result of a deduction is received, the result of the deduction is reported to its parent deduction. When all pending sibling deductions are complete, the combined result from the sibling deductions is the result reported for the parent deduction. Note that all pending sibling deductions are terminated if only one positive result is desired and a positive result is received for any sibling deduction.

A deduction can be aborted by sending a **kill-task** message. Note that aborting a deduction also requires aborting other deductions spawned from it (recursively). Also note that a **kill-task** message may be point-to-point if it is reasonably certain that only one processor is working on the deduction in question. Otherwise, the **kill-task** message has to be a broadcast message.

5.2. Detailed Description of ESP

5.2.1. Data Structures

Each processor maintains the following data structures.

1. **Task Names**: As explained before, each task has a unique task name. Each task name has several pieces of book-keeping information attached to it. The most important book-keeping information consists of:

    a. Task Description: This is the description of the task to be done. An example task description is shown below.

    (*BC PROP-SET B-SET ALL?*),

    where *BC* is the name of the backward-chaining procedure, *PROP-SET* is the set of propositions to be proved, *B-SET* is the set of bindings obtained so far, and *ALL?* is a boolean variable that means all positive results must be found (if true) or only one positive result must be found (if false). Note that *PROP-SET* and *B-SET* together specify a node in the *or* deduction tree.

    b. Parent Task: This is the task name of the parent task. This information is kept only at the processor that originated the task.

---

[10] For the case in which a single answer is required for the top-level deduction, a positive answer can be sent to the processor that originated the top-level deduction and the rest of the computation can be terminated.

c. Pending Subtasks: This is the set of task names of subtasks whose results have not been reported yet. This information is kept only at the processor executing the task.

d. Results: This is the set of results received so far for this task. This information is kept only at the processor executing the task.

e. Grabbed Timestamp: This is the timestamp at which the task was grabbed. This can be revised if a future "grab" request for the same task beats out a previous "grab" request.

f. All-results?: This is a boolean variable. If it is "true", all positive results of the task are required. Otherwise, only one positive result of the task is required.[11]

2. **Current Task**: This is the task name of the task currently being worked on.

3. **Message Queue**: This is the queue of messages not yet processed by the processor.

4. **Actual Work Set**: This is the set of task names that the processor is committed to perform.

5. **Potential Work Set**: This is the set of task names that were "supplied", by this processor as well as by remote processors, but are not "grabbed" yet.

5.2.2. Messages

Each message is described here with the following syntax:

Message-Type(*Arguments*)

The following messages are required:

1. **New-Task**(*Task-Name Task-Description*): When a processor has a surplus task, it broadcasts a **new-task** message. When a processor receives a **new-task** message, the processor sets up the appropriate book-keeping information for the task name and puts the task name in the **potential work set**.

2. **Grabbed-New-Task**(*Task-Name Bid Timestamp*): Consider first the case in which all processors have the same speed. The *Bid* argument is null. When a processor selects a certain task as the next **current task** for itself, it broadcasts a **grabbed-new-task**

---

[11] Notice that if a deduction (or task) requires all results, all its subtasks require all results and if the task requires only one result, all its subtasks require one result.

message for that task. When a **grabbed-new-task** message is received for a task, it is removed from the **potential work set** of the processor where the message is received and the book-keeping information for the task is revised (if required). If the task for which the **grabbed-new-task** message is received is the current task or even if the task is awaiting completion, then it will have to be aborted if the timestamp in the new **grabbed-new-task** message is lower than the previous timestamp for the task name. Aborting a task also means aborting any pending subtasks for the task. A remote task is aborted by sending a **kill-task** message as explained below.

If all the processors do not have the same speed, *bid* is the estimated time of completion of the task. Its use is explained in section 8.3. Basically, ties between multiple bids are broken first on the basis of the completion time and then on the basis of the *Timestamp*.

3. **Kill-Task**(*Task-Name*): A **kill-task** message is sent to abort tasks remotely. This message is sent point-to-point if it is reasonably certain that it is being done at a certain processor. Otherwise, the message is broadcast to make sure it is aborted at all the processors. When a **kill-task** message is received, the task name is removed from the **potential work set**. In case the task in questions is the current task, it is stopped and another current task is chosen. Also, if the task is awaiting completion of some of its subtasks, the subtasks are recursively aborted. Again, if any of the subtasks could have been grabbed remotely, **kill-task** messages will have to be sent for each.

4. **Done-Task**(*Task-Name Binding-List*): This is the only message that is required to be reliable. It is sent when the answer for a remotely originated task is obtained. When a **done-task** message is received, the result (i.e., the binding list) is reported to the parent of the task in question. The result is added to the results already obtained for the parent task name. If there are no pending subtasks for the parent task name or if the answer is positive and only one positive answer is required, then all pending subtasks are aborted and the combined result for the parent task is then reported to the processor that originated the parent task. Again, a **done-task** message may have to be used if the originator of the parent task was a remote processor. The entire computation is complete when all needed results (one or all) for the top-level task are reported.

5.2.3. Heuristics

The following heuristics are used to reduce computation and communication.

When it is time for a processor to select a new **current task** out of the **potential work set**, locally generated tasks are preferred; this saves on sending a **done-task** message on that task when the task is completed.

Backward-chaining within a processor is done in a depth-first fashion; this reduces the search

(compared to breadth-first) if only one positive result is desired.[12]

To reduce the number of grabs for the same task, the following heuristic is used. Each processor keeps track of the busy/free status of all processors with names less than its own (in some well known sorting order). Initially, all processors are assumed to be free. A successful grab by any processor changes its status to bu. and a failed grab changes its status to free. When a processor is about to grab a new task, it checks to see whether a processor with a name lower than its own is free. If there is no such free processor, it goes ahead with it grab, otherwise it lets the other free processor grab the task within some reasonable time in the future.[13] If the expected grab of the task in question has not happened by the reasonable time in the future, any processor is allowed to grab that task.[14] This last conservative step ensures that the task is considered "grabbable" (albeit after a delay) even if the busy/free tracking of a certain processor incorrectly shows the processor to be free.

When deciding which tasks to move to the **potential work set**, pick the most costly. Also, when deciding which task to grab out of the **potential work set**, choose the most costly. This is done so that a free processor grabs the most costly task first out of the surplus tasks available from all the processors. The effect is that, in general, processors tend to remain busy longer between grabs and this reduces the message traffic.

If no information is available about the cost of deductions, the least one can go by is that deductions higher up in the deduction tree are, on the average, more costly. Also, if the delays are not very high, all processors can be heuristically assumed to be at the same level. Therefore, the order of announcement of new tasks on the broadcast network can be heuristically assumed to be in the more costly to less costly order. This allows one to avoid keeping track of the levels of tasks in the deduction tree and costly insertions of tasks into arbitrary places in the cost-sorted order in the work sets. Instead one can implement the work sets as doubly-linked lists and add/delete tasks more efficiently.

---

[12] This pure depth-first strategy will have to be modified if there are infinite depth paths. In addition, other strong local heuristics could also be used to modify the depth-first strategy. In the current implementation, only depth-first is used within a processor.

[13] The task is removed from the **potential work set** till the future expected grab time.

[14] This is done by placing the task in a balanced heap [1] (ordered by the time in the future by when the task is expected to be grabbed). Tasks are removed from the heap, lowest time first, at the time of their expected grab and checked to see if they have been grabbed. If they have not been grabbed, they are re-inserted into the **potential work set**.

5.2.4. Implementation Details

It was mentioned before that locally generated tasks are grabbed before remotely generated tasks. To implement this preference strategy efficiently, the **potential work sets** and the **actual work sets** are split into two: a local part and a remote part.

5.3. Domain of Application of ESP

More than one processor can start executing a task at the same time. Therefore, either the tasks should be side-effect free as in the backward chaining case or repetitions of side-effects should be acceptable. Examples of acceptable repetitions of side-effects include 1) side-effects designed only to increase efficiency but not to affect the correctness of the computation (e.g., caching of propositions in the backward chaining case) and 2) idempotent computations (e.g., setting a certain global variable to a certain value).

Also, the dynamic supply of tasks is especially useful if the tasks are themselves dynamically generated as in the backward chaining case. However, the dynamic supply protocol described here can be useful even if tasks are not generated dynamically as shown in the following two examples.

1. Tasks are known beforehand but there is not enough information about the processor computation times required for the tasks or there is not enough information about their communication or dependency requirements to warrant static distribution.

2. There may be complete information about task execution times, their communication requirements and their dependencies, but the complexity of the algorithm required to statically allocate the tasks sufficiently well may be unacceptably high.[15]

So far, no information about the cost of a deduction was taken into account to reduce overall computation time (perhaps by cutting down on communications cost). If such information is available, it can be used effectively as will be seen later.

As mentioned before, ESP can handle a realistic failure set. Details of this can be found in Appendix I

---

[15] It is well known that optimal task allocation even for relatively simple problems is NP-complete [11].

5.4. Comparisons with Previous Work

Contract Net [7] was one of the first efforts to address the problem of dynamically distributing tasks among processors. In the Contract Net, the responsibility of doing a task is transferred from its originator to some other processor in three phases. In the first phase, the originator *announces* the task to processors it thinks may be interested in the task. This is done by a broadcast message if any processor may possibly do the task. In the second phase, all interested processors submit *bids* for the task to the originator. Finally, in the last phase, the originator of the task evaluates the bids and *awards* the task to the processor that submitted the *best* bid. Enterprise [9] uses a similar supply protocol but with significant specializations. In the discussion below, ABASP stands for the *Announcement-Bid-Award* type of protocol used in the Contract Net and Enterprise.

ESP allows task execution to begin as soon as a bid is submitted (in a **grabbed-new-task** message). This can increase processor utilization compared to ABASP in which task execution begins only when an award message is received.

Also, no award message is required as in ABASP. This already implies fewer messages. Additionally, there is the potential of drastically reducing the number of **grabbed-new-task** messages, the equivalent of *bid* messages in the ABASP case. The number of **grabbed-new-task** messages may, in the best case, be one per new task announced; this will happen if the first **grabbed-new-task** message is always sufficiently early to inhibit any other processor from making an attempt to "grab" the same task. Also, recall that a heuristic to reduce the number of **grabbed-new-task** messages was presented earlier. Also, one can, in fact, use special hardware to reduce the number of **grabbed-new-task** messages per new task announced. Two possibilities come to mind for such special hardware. These possibilities are described in Appendix II.

Note, however, that there is one source of extra messages the supply-driven strategy using ESP when compared with Contract Net and Enterprise. In ESP, multiple executions of the same task can occur until the best bid reaches all other processors executing the task. Now, these multiple executions may lead to further decomposition of the task into subtasks and, therefore, extra **new-task** messages for the sub-tasks in question. Moreover, when the extra executions of the task are finally killed by the best bid for the task, these sub-tasks have to be killed as well by more **kill-task** messages. In practice, experimental results indicate that this is not a problem when delays are reasonably short and bandwidth is not a constraint. Short delays lead to a quick killing of replicated executions before extra sub-tasks can be generated and announced from the short-lived replicated executions. The bandwidth constraint case will create a problem for Enterprise and Contract Net as well but they do not propose any specific solutions. This paper, however, presents a variable supply

model that can be used to reduce supply of new tasks[16] to deal with the bandwidth constraint problem.

Shapiro's Bagel [16] also deals with dynamic distribution of tasks in a multiprocessor system. Shapiro presents language constructs to provide for a pre-determined mapping of tasks to processors. This is in contrast to the present work where all mappings are dynamically done. The major difference between Shapiro's work and the work presented in this paper is in the choice of problem domain. The problem domain chosen by Shapiro deals with *systolic* problems in which subtasks are generated from tasks in a quite precisely predetermined fashion. In the backward-chaining case, this would mean that one would know ahead of time how many subtasks were going to be generated from each task. This paper makes no such assumption.

6. A Supply-Driven Allocation Strategy

Imagine a situation in which the communication capacity of the broadcast network is not a problem and messages can be sent from a processor as fast as it can transmit them. This could happen, for example, at great cost if there were dedicated communication links between every pair of processors. Alternatively, the communication bandwidth of a single broadcast link can be assumed to be infinitely larger than the throughput required for the computation . In this situation, one need not worry about when to stop supplying surplus tasks from a processor to other processors to avoid swamping the network. Note that this throughput consideration is quite different from not supplying a task because the task is too fine-grained to make remote execution practical. The infinite bandwidth assumption is, of course, not practical but serves to illustrate the extreme of very plentiful network bandwidth.

In the supply-driven strategy, all processors place all newly generated tasks, except one task that is chosen as the **current task**, in the **potential work set**. Of course, all tasks placed in the **potential work set** must be announced with **new-task** messages. A processor is allowed to grab a task from the **potential work set** only when it runs out of internally generated tasks. Since all surplus tasks are announced as being available, this strategy needs the highest network bandwidth of all strategies included in VSM.

---

[16] The extreme case is a demand-driven strategy.

6.1. Experimental Results

All results reported here were obtained by simulating the multiple processor architecture described in section 3. The simulator is described in Appendix III.

The unit of time for the results reported is the time taken to do a unit backward-chaining deduction.

Figure 6-1 gives some information about the *or* deduction tree that was experimented with. The database of each processor contains the behavioral and structural description of a piece of digital hardware - a 4-bit adder. The databases also contain the values set at the inputs of the adder. The top-level proposition is a fact to be proved about one of the outputs of the adder. It turns out that the top-level proposition has no positive answer and, therefore, the entire deduction tree is searched in trying to prove the proposition.

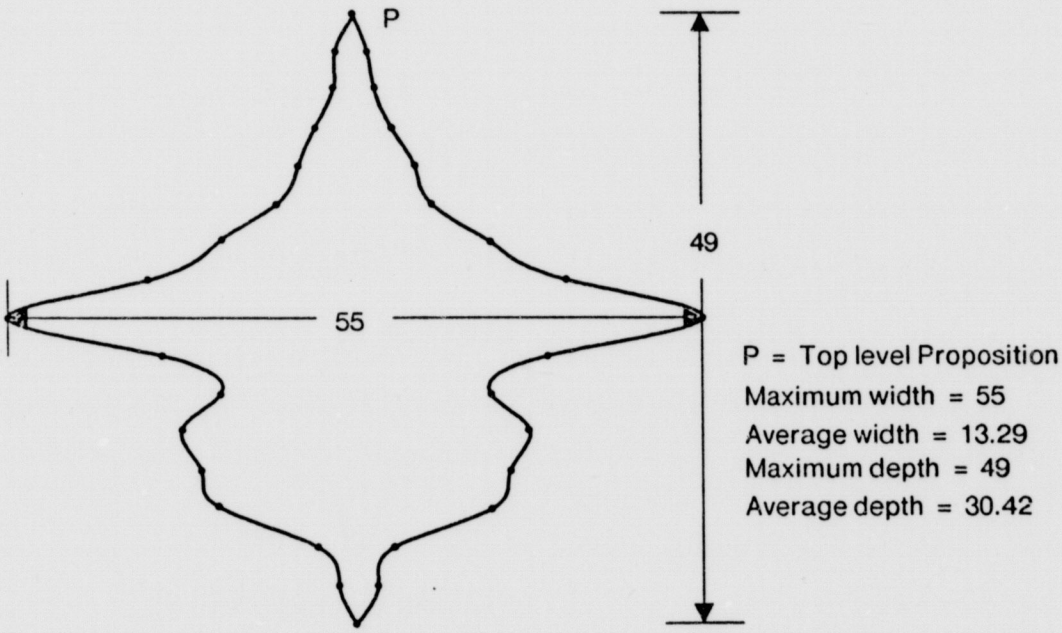


Figure 6-1: Or Deduction Tree for Adder Example

The time taken for 1 processor to do the adder example is 652 unit deduction time units. Figure 6-2 shows the results obtained for the adder example with 10 processors. The delay was 1 and the throughput was infinity.

Time Taken = 93
Speedup = 7.01

New-task messages = 82
Grabbed-new-tasks messages = 102
Kill-task messages = 0
Done-task messages = 29
Total number of messages sent = 213

Figure 6-2: Results for Adder Example with 10 Processors

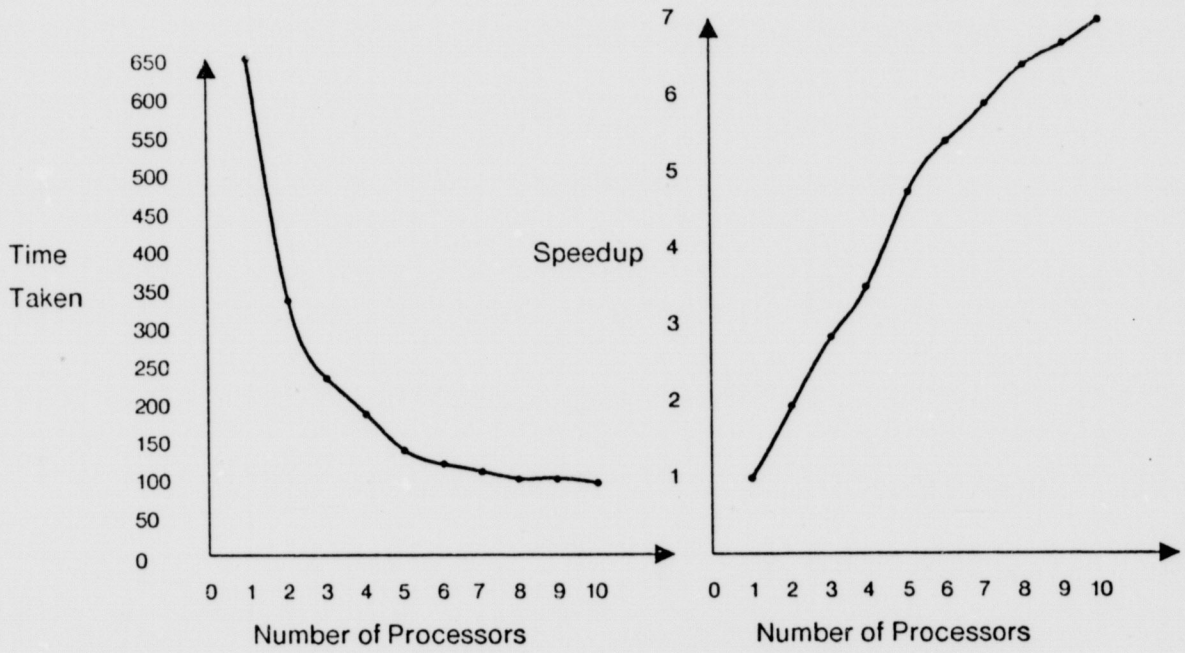
As can be seen from figure 6-2, a substantial speedup was obtained by using ten processors. To get an idea of how the time taken decreases with the increase in the number of processors, look at figure 6-3.

Notice from the figure that initially the speedup is almost linear but becomes less so as the number of processors increases. Some of the less than linear speedup is due to the inter-processor message delay but some is certainly due to the fact that the deduction tree for the example does not permit very large speedup due to its shape (see figure 6-1). The deduction tree is not very wide at some depths and, therefore, all processors cannot be kept busy at all times by any possible supply protocol.

The message traffic does not vary much when the number of processors in varied from 1 to 10. The total number of messages sent increases from 165 for 1 processor to 213 for 10 processors.

The results so far were obtained with an allowed throughput of infinity. It is certainly possible that the throughput is not enough to send all the messages that are queued at some time in the next unit deduction time. One might expect that with a throughput bottleneck the performance of the supply-driven strategy will degenerate. This is exactly what is observed in the simulation (as shown in figure 6-4), when a fairly severe throughput limitation of 2 is imposed; the delay is one and the number of processors is 10 as before.

Notice in figure 6-4 that the number of **kill-task** messages is 24 as opposed to 0 in the infinite throughput case with ten processors. This happens because **grabbed-new-task** messages are delayed due to the limited throughput and this allows multiple copies of the same computation to get executed longer than before. In turn, the longer multiple executions cause other redundant computations which eventually have to be aborted with **kill-task** messages.



Number of Processors	Time Taken	Speedup	Number of Messages *
1	652	1	82/83/0/0/165
2	332	1.96	82/83/0/14/179
3	228	2.86	82/85/0/11/178
4	185	3.52	82/90/0/15/187
5	138	4.72	82/85/0/16/183
6	120	5.43	82/88/0/16/186
7	110	5.93	82/91/0/20/193
8	102	6.39	82/98/0/24/204
9	98	6.65	82/100/0/27/209
10	93	7.01	82/102/0/29/213

* a/b/c/d/e stands for a new-task, b grabbed..., c kill-task, d done-task, and e total messages.

Figure 6-3: Time Taken for Adder Example for Varying Numbers of Processors

Time taken = 213
Speedup = 3.06

New-task messages = 112
Grabbed-new-task messages = 220
Kill-task messages = 24
Done-task messages = 95
Total number of messages sent = 451

Figure 6-4: Results for Throughput Bottleneck Case

6.2. Conclusions

The supply-driven strategy works well when the throughput is not a bottleneck. However, the performance can be quite bad if the throughput does become a bottleneck.

It is important to note here that most of the time, it was not really necessary to announce all surplus tasks when all processors were already busy. The next section offers an alternative that can reduce the communication requirements and thereby decrease overall task completion time.

7. A Demand-Driven Allocation Strategy

The supply-driven strategy always supplies any surplus tasks and, therefore, uses up a lot of throughput. The other extreme is to only supply surplus tasks when all previously announced tasks have been grabbed; this is the demand-driven strategy. As will be shown later, this can drastically reduce message traffic and lead to faster completion of tasks when the available throughput is a bottleneck.

All surplus tasks are not supplied and, therefore, not placed in the **potential work set**. The surplus tasks that are not supplied are put in the **actual work set**. More precisely, when a processor has more than one or-parallel deduction component, it keeps one as its next current task, supplies a task to the **potential work set** if the **potential work set** is empty[17], and keeps the remaining tasks in its **actual work set**. When a processor needs to get a new **current task**, it first gets it out of its own **actual work set**, without sending a **grabbed-new-task** message. However, if the **actual work set** is empty a task is obtained from the **potential work set**, after sending a **grabbed-new-task** message as before. With this strategy, the tasks are supplied only when they are demanded (i.e., when the **potential work set** gets empty).

---

[17] If a task is supplied to the potential work set, it has to be announced with a new-task message.

7.1. Comparison with Supply-Driven Allocation Strategy

To get a feeling for the reduction of message traffic by using the demand-driven strategy, consider the following example in figure 7-1. The example is for a balanced, binary, or deduction tree of height *n* and the number of processors is 2^p. A few simplifying assumptions made to make the analysis easier (but retaining the essential idea of message traffic) are that message delays are zero, throughput is infinity, there is one **grabbed-new-task** message for each **new-task** message and all tasks supplied end up being grabbed by remote processors and, therefore, require a **done-task** message.

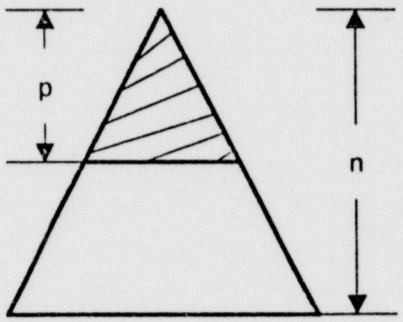


Figure 7-1: Reduction of Message Traffic by using Demand Strategy

The number of messages required for the supply-driven case is $3*2^n-3$ and the number of messages required for the demand-driven case is $3*2^p$ (see Appendix IV for the details).

Essentially, what happens in the demand-driven case is that the **new-task** messages need to be sent out only when the shaded part of the deduction tree in figure 7-1 is being searched. Whereas in the supply-driven case, all surplus tasks in the deduction tree have to announced.

In the limit, as *n* and p increase to infinity, the ratio of messages in the demand-driven case to the messages in the supply-driven case is 2^{n-p}. Clearly, this can lead to a big message traffic reduction for the demand-driven case. Experimental results will also be presented later to confirm a reduction in message traffic.

Notice one problem with the demand-driven strategy. It may happen sometimes that more than one processor becomes free at the same time. One of them will get to grab the single task in the **potential work set**; the others will have to wait until the **potential work set** is replenished. This

delay need not have taken place for the supply-driven strategy because all surplus tasks would have been supplied as they were generated. Therefore, the demand-driven strategy may have, in general, a higher delay in transferring tasks among processors.

There is another attractive feature of the demand-driven strategy that was not mentioned so far. This has to do with storage requirements. Since the **actual work set** of each processor is private to it, only one copy of that is maintained. In the supply-driven strategy, all the surplus tasks would have been in the **potential work set** and each processor would have had to maintain a copy of that. Therefore, the demand-driven strategy can do with a lot less storage than the supply-driven strategy.

7.2. Experimental Results

Figure 7-2 illustrates the performance of the demand-driven strategy for the adder example with ten processors, unit delay, and infinite throughput.

```
Time taken = 105
Speedup = 6.21

New-task messages = 24
Grabbed-new-task messages = 42
Kill-task messages = 0
Done-task messages = 24
Total number of messages sent = 90
```

Figure 7-2: Results for Adder Example using Demand Strategy without Throughput Bottleneck

As can be seen by comparing figure 7-2 with figure 6-2, the time taken with the demand-driven strategy (105) is slightly more that the time taken with the supply-driven strategy (93). The extra time taken is due to the extra delay in supplying tasks. However, the number of messages sent with the demand-driven strategy (90) is much lower than the number of messages sent with the supply-driven strategy (213). Therefore, it should be expected that the demand-driven strategy will not degrade as much in the presence of the same throughput limitation imposed on the supply-driven strategy. This, in fact, is the case as illustrated in figure 7-3; the delay is one and the maximum throughput is 2.

Figure 7-3 shows that the time taken for the demand-driven strategy only increases from 105 to 106 by changing the maximum throughput from infinity to 2. In the bottleneck case, the demand-driven strategy (with a time of 106) completely outperforms the supply-driven strategy (with a time of 213).

Time taken = 106
Speedup = 6.15

New-task messages = 24
Grabbed-new-task messages = 43
Kill-task messages = 0
Done-task messages = 24
Total number of messages sent = 91

Figure 7-3: Results for Adder Example using Demand Strategy with Throughput Bottleneck

7.3. Conclusions

The demand-driven strategy can perform better than the supply-driven strategy in the presence of throughput constraints because it requires fewer messages.

It is possible that the tradeoff between delay in supplying tasks (high for the demand-driven case) and the message traffic requirements (high in the supply-driven case) may not have been optimally resolved in either of the extremes.[18] VSM allows any intermediate strategy to be selected with great ease and it seems quite probable that intermediate strategies will be found useful for intermediate communication situations. However, no results are available are available at the present time about these intermediate strategies.

8. Extensions

The variable supply idea is actually more general than what its application so far in VSM, in conjunction with ESP, might suggest. The extensions described in this section retain the essential goal of VSM, the ability to vary the supply of new tasks given that communication constraints may exist.

8.1. Extension to Different Task Domains

In section 5.3, it was mentioned that ESP could only be used in cases where the tasks were side-effect free or where side-effects were acceptable. In cases where tasks do have side-effects or side-effects are not acceptable, one can use a different supply protocol in VSM that does not allow multiple instances of a task to start executing. For example, one could use the announcement-bid-

---

[18] In the specific case of the adder example, it turns out that the demand-driven strategy is, in fact, the best strategy. No intermediate strategy is better.

award supply protocol (of Contract Net or Enterprise).

8.2. Extension to Different Processor Interconnection Structures

By allowing the **potential work sets** of different processors to be different, it is possible to apply VSM to cases where the processor interconnection structure is not a single broadcast network. Basically, the idea is to allow a new task to be placed in the **potential work sets** of a certain subset of the processors. This subset is known globally given the name of the processor that originated the task. Now, when a task is grabbed by a processor, the grabbing processor must make sure that the **grabbed-new-task** message is sent out to the subset of all processors in whose **potential work sets** the task was originally placed. Similarly, a **kill-task** message that might have been broadcast in the broadcast net case is now sent to the same subset of processors. Essentially, a broadcast message in the broadcast network case is replaced by a *limited broadcast* message. The recipients of any *limited broadcast* message depend on the originator of the task. Consider two examples of different processor interconnection structures mentioned below.

The first case is a grid of networks suggested by Ullman [19]. Processors are arranged in a grid with each row and each column being connected by a separate but single network. The obvious way to choose the subsets mentioned above is to make the subset for each processor be the row and column of processors in which the processor lies.

The second case is a grid of processors in which only neighboring processors are connected separately. There are several variations of this. One suggested grid structure has each processor connected to 6 neighboring processors, called a hex-connected plane, and also provides wrap-around connections for the processors on the edge of the grid. A possible subset for the *limited broadcast* from each processor could be, for example, all processors up to a certain number of point to point connections away. The *limited broadcast* in this case will have to be accomplished by several point to point messages and sophisticated routing hardware to accomplish this automatically would certainly help.

Therefore, by allowing for a different supply protocol than ESP and/or by allowing **potential work sets** to be different (in a structured way), VSM can be used in variety of situations.

8.3. Using Additional Information

In this section, the use of two kinds of additional information will be demonstrated - estimates of costs of deductions and estimates of probabilities of finding positive results in deduction sub-trees.

8.3.1. Costs of Deductions

So far no information about the cost of proving various propositions has been assumed. Let us say we now have reasonable estimates of these costs.

The first application of cost estimates can be to stop distribution of certain propositions if their costs are too low to justify the overhead of distribution to other processors.

Also, estimates can be used to break ties between processors with different speeds in a more efficient way than by just comparing timestamps. Instead of just comparing timestamps, the tie can be broken first on the basis of which processor can complete the task first (regardless of when the task is grabbed and starts executing). Again, a tie on this basis can be broken by comparing unique processor names.

Another application of estimates can be to do load balancing. The **potential work set** is known to each processor but each processor's **actual work set** is private to it. An extra message type, **actual-work-set-profile**, could be introduced to have processors exchange information about the total costs of the tasks in their **actual work sets**.[19] Now, by a publicly known load balancing formula (known to each processor), each processor could grab tasks from the **potential work set** to balance out the load. The public load balancing formula may dictate that some processors with high loads actually send some of their tasks from their **actual work sets** to the **potential work set** before some processors with low loads can grab the requisite load of tasks from the **potential work set.**[20]

8.3.2. Probabilities of Proving Propositions

Now, let us introduce another type of additional information. This information is the probability of success of finding a positive answer for a proposition by searching its deduction tree. This information is not useful if the entire deduction tree has to be searched anyway[21] but it can be used effectively, as described below, in the case that only one positive answer is desired for the top-level

---

[19] Of course, this cost information could be piggy-backed on the other messages sent by processors for extra efficiency.

[20] Again, to increase efficiency, one can imagine direct transfer of tasks between **actual work sets** of processors.

[21] Remember that the entire tree has to be searched when all positive answers are desired.

proposition.

Recall that the problem deals with or-parallel deductions that are side-effect free. In the case that only one positive answer is desired, the computation can be terminated as soon as one answer is obtained. Therefore, if there are several or-parallel propositions to be proven, one should try proving them in the order of most promising to least promising. Formally, the problem can then be described as follows. There are *n* or-parallel tasks, t_1 through t_n. Each task t_i has a cost estimate (e_i) and a probability of success (p_i) that is independent of the order in which the tasks are executed. Additionally, these tasks are spread out over the various processors. What is the "best" order of doing these tasks and "how much effort" can one expend in trying to do them in the "best" order?

The problem presented above is equivalent to the one treated by Rosenschein and Singh in [15]. The problem considered there is as follows. There are *n* methods of solving a problem. We have their estimated costs and their estimated probabilities of success that are independent of the order of using the methods. What is the "best" order of using the methods and "how much effort" can one expend in doing them in the "best" order? Rosenschein and Singh quoted previously reported results by Simon and Barnett on the "best order" [4, 18] for an arbitrary number of methods[22] and then went on to put an upper limit on the "effort" that one can expend to do them in the "best order" when the number of methods is two. Presumably, the latter result can be generalized for an arbitrary number of methods but that is not necessary. There is a brief description below of how to use the "how much effort" result in the present problem.

Clearly, if the number of processors is p, the p most promising tasks should be pursued. The fastest processor should work on the most promising task, the slowest processor on the least promising of the p most promising tasks and so on. If the p most promising tasks are not already distributed in the optimal manner[23], each processor can make a decision about whether or not to redistribute its tasks (to achieve the optimal ordering) in the following approximate way. For a task (t_A) that a processor should send to another processor, whose most promising task would otherwise be t_B), the key decision is as follows. Can task t_A be sent with less "effort" than the upper limit of the effort that can be spent in exchanging the order of execution of t_A and t_B? The upper limit is obtained from the "how much effort" result of [15]. Each processor can make these calculations every so

---

[22] Applying the results by Simon and Barnett shows that in the special case where the or deduction tree is balanced and the results are uniformly spread out the best order is arbitrary (i.e., it makes no difference which task is executed first).

[23] This can be found out easily. Let the **actual-work-set-profile** message now include the "promising" metric of its first few most "promising" tasks.

many time units to approximate the best possible task execution ordering over the entire deduction.

9. Future Work

A major effort of future work will be to remove the two major bottlenecks present in the current method of distribution of deductions: the replicated database and the shared communication network. The architecture to be used will consist of large numbers of processors connected with local connections to neighbors (as in a hex-connected plane, for example). Each processor will have a limited amount of local memory (of a few thousand words).

And parallelism may also be taken advantage of in a limited way. Taking full advantage of *and* parallelism is a very hard problem.

10. Conclusions

VSM allows an easy and powerful mechanism to choose strategies that work best under different communication constraints. The two extreme strategies were shown to be useful by theoretical and experimental results. For the specific case of processors with replicated databases communicating via a broadcast network, an efficient supply protocol (ESP) was presented. In addition, VSM and ESP can be adapted for use with other tasks, interconnection structures, and additional information about tasks.

I. Failure Model

ESP can handle message and processor failures. Details are provided below.

A message may fail to reach any destination with some low non-zero probability. It is possible to ensure reliable point to point messages by requiring a positive acknowledgement for each message. The best method known for reliable broadcast also requires a minimum of one control message per broadcast message [5]. ESP does not require reliable broadcast messages at all. It also does not require reliable point-to-point messages for the most part except for reliable **done-task** messages.

New-task, **grabbed-new-task**, and **kill-task** messages do not need to be reliable. If a **new-task** message fails, the task is not lost because it is maintained at the originator anyway; only potential "grabbers" may be lost. If **grabbed-new-task** and **kill-task** messages fail, multiple executions of the same task may take place; no other damage is done. Therefore, lost **new-task**, **grabbed-new-task**, or **kill-task** messages can lead to an efficiency degradation but the correctness of the deduction is preserved.

It is assumed that a message that is garbled in transmission can be detected as such and discarded. This case is then equivalent to a message failing to reach a destination processor. The low non-zero probability (mentioned above) of a message failing to reach a destination is meant to include this case.

It is assumed that the broadcast communication network does not fail completely. This is reasonable because experience on a broadcast communication network like the Ethernet has shown that its overall reliability for system-wide failures is "very high" (according to Shoch and Hupp [17]).

Failure of processors can be handled with traditional checkpointing techniques[24] and will not be described in this paper.

---

[24] The state of the set of tasks awaiting completion at each processor has to be checkpointed. This is not done in the current implementation.

II. Reducing Grabs with Special Hardware

The first possibility concerns itself with a CSMA-CD type broadcast network like the Ethernet. In this case, only one message can be sent at one time. Assume that at some instant, *n* free processors attempt to grab the same task with **grabbed-new-task** messages. All these messages will be queued for delivery in their respective Ethernet buffers. One will get through first. If all processors are identical, one wants the first **grabbed-new-task** message to win out and the others to be not sent at all. The special hardware in the other n-1 processors, where **grabbed-new-task** messages are still queued, should be able to see the first **grabbed-new-task** message and withdraw their own **grabbed-new-task** messages and also abort their own execution of the task in question. The latter aborting of tasks could be initiated with a hardware interrupt from the special piece of hardware.

The second possibility concerns itself with a ring type network like the Cambridge Digital Communication Ring [20]. In this case, all messages go past all the processors one by one in the order that the processors are arranged in the ring network. In this case, special hardware could be used as follows by free processors. A free processor could instruct its network interface to tag a **new-task** message as it goes by iff it has not been tagged already. This mechanism will ensure that only one processor grabs the new task.

It should be noted here that the above special hardware only prevents multiple grabs from already free processors of newly announced tasks. However, it does not prevent multiple grabs if processors complete a task and then seek to grab tasks from the **potential work set**. The number of multiple grabs from the latter case will be low if the task completion times are well spread out over time.

III. Simulating the Multiple Processor Architecture

The multiple processor architecture described in section 3 and the task allocation strategies described were all simulated on a sequential processor (Symbolics 3600). A preliminary version of the supply-driven strategy was implemented with multiple (real) 3600's communicating over a Chaosnet [13] to obtain the parameters for the simulation.

In the simulation, each (simulated) processor uses a backward-chaining deduction procedure. A single copy of a set of facts (including rules), to represent the replicated databases of the processors, is shared by the processors in the simulation. Each processor has its own address space that is not shared with any other processor. It is in this private address space that any state information associated with the task allocation strategy is stored.

Processors are timesliced in the simulation in a round-robin fashion; each processor is allowed to do one unit backward-chaining deduction (see section 2) in a timeslice. A simulator clock starts the simulation with time zero and increments the time at the beginning of each new round of timeslices for the processors. One processor P_A starts out with the top level deduction at time zero. The simulation is stopped when the answer for the top level deduction is received at processor P_A; also, the time taken is recorded.

Message flow during the simulation is shown in figure III-1. Each processor has its own input queue and its own output queue. Messages may be generated from a processor to be sent to other processors during a timeslice; these messages are put in the processor's output queue. After each processor has been granted a timeslice, the simulator does the actual message sending. The simulator retrieves all the messages from all the output queues of the processors being simulated and adds them to a message queue internal to the simulator. At this point, the simulator computes the delay and throughput, which may be a function of the offered throughput (i.e., the numbers of messages queued in the internal message queue of the simulator). An example delay-throughput characteristic is shown in figure III-2. The throughput computed is in terms of how many messages to actually send to destination processors. The delay computed decides the future time at which the messages will be received at its destination. The future time is adjusted to be at least the future time when the last round of messages were sent by the simulator.[25] The adjusted future time is tagged on to the messages to be sent and the appropriate messages are placed in the input queues of the appropriate destination processors (and removed from the internal message queue of the simulator).

---

[25] This is intended to disallow messages sent later being received before messages sent earlier.

There may be some messages left over in the simulator; these are the messages that could not be sent because the offered throughput was greater than the maximum throughput possible. These left-over messages and the next batch of queued messages from the processors will decide the offered throughput and delay for the next round of message sending by the simulator.

Messages are read in by processors at the beginning of each timeslice. Note that only those messages tagged with a future time less than or equal to the present time are readable.

Message delay between two processors was experimentally found to be approximately 1/3 of one timeslice (or one unit backward-chaining deduction time) in the preliminary implementation on an unloaded Chaosnet with 3600's. Message delay for an unloaded network was, therefore, rounded off to 1 timeslice (i.e., messages sent at timeslice n are received at timeslice $n + 1$).

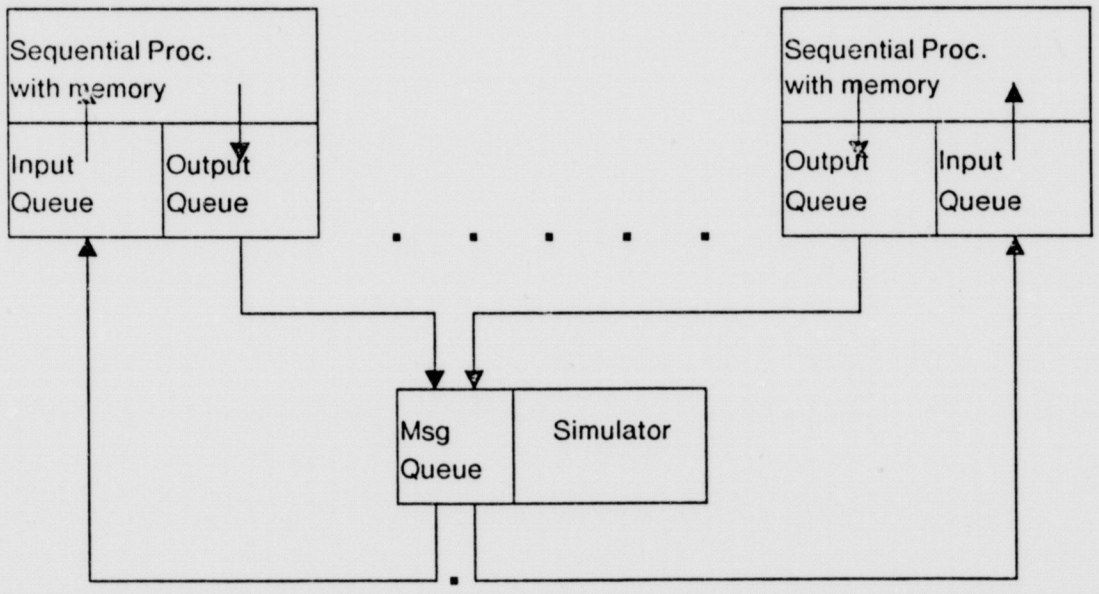


Figure III-1: Message Flow during Simulation

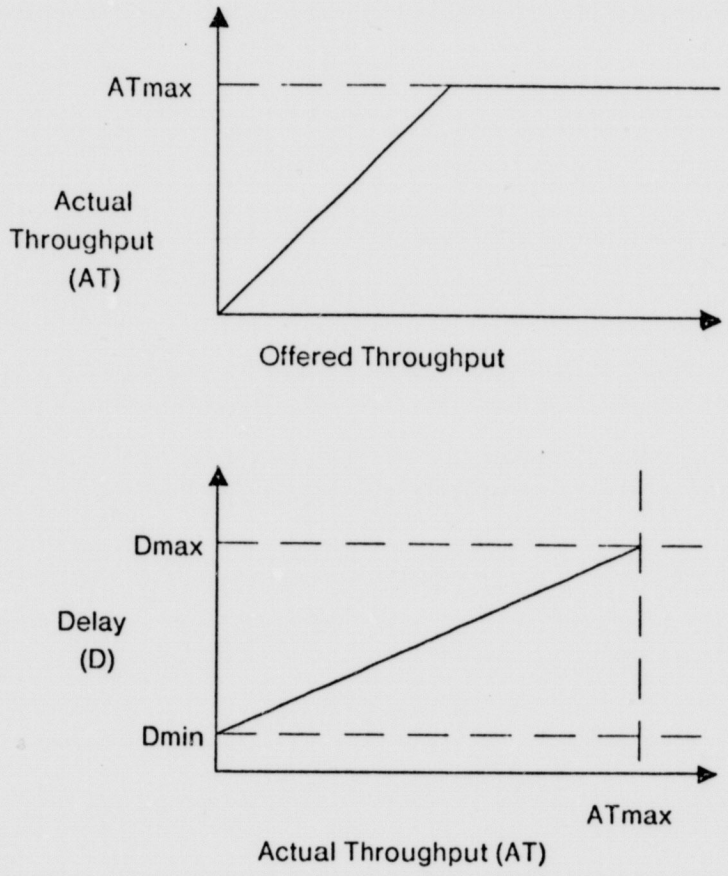


Figure III-2: Example Delay-Throughput Characteristic

IV. Number of Messages for Supply-Driven and Demand-Driven Strategies

As shown in figure 7-1, the balanced, binary or deduction tree has height *n* and the number of processors used is 2^n.

For the supply-driven case, the number of **new-task** broadcast messages required is 2^n-1. Remember that each non-leaf node leads to one **new-task** message because one of the two children is kept at the processor and the other is announced. The number of **grabbed-new-task** broadcast messages and the number of point-to-point **done-task** messages is also (separately) the same as the number of **new-task** messages. Therefore, the total number of messages required is $3*2^n$-3.

For the demand-driven case, **new-task** messages are only sent out until all processors get busy. After that one additional **new-task** message is sent out to keep the **potential work set** at size one. Since the number of processors is 2^p, the number of **new-task** broadcast messages required is 2^p. Therefore, the number of **grabbed-new-task** messages and the number of **done-task** point-to-point messages is also 2^{*p} each. This leads to a total of $3*2^p$ messages.

References

[1] Aho, A. E. et al.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Menlo Park, 1976.

[2] Eds. Barr, Avron and Feigenbaum, Edward A.
The Handbook of Artificial Intelligence.
William Kauffman, Inc., Los Altos, California, 1981, pages 39, chapter 2.

[3] Eds. Barr, Avron and Feigenbaum, Edward A.
The Handbook of Artificial Intelligence.
William Kauffman, Inc., Los Altos, California, 1982, pages 80, chapter 12.

[4] Barnett, Jeffrey A.
How Much is Control Knowledge Worth?: A Primitive Example.
ISI Working Paper, USC/Information Sciences Institute, June, 1982.

[5] Chang, Jo-Mei and Maxemchuk, N.F.
Reliable Broadcast Protocols.
Technical Note 4, Bell Labs, Murray Hill, January, 1983.

[6] Conway, R.W., Maxwell, W.L., and Miller, L.W.
Theory of Scheduling.
Addison-Wesley, Reading, Massachusetts, 1967.

[7] Davis, R. and Smith, R.G.
Negotiation as a Metaphor for Distributed Problem Solving.
Artificial Intelligence 20(1), January, 1983.
Also available as MIT AI memo # 624.

[8] Lawler, E.L., Lenstra, J.K., and Rinooy Kan, A.H.G.
Recent Developments in Deterministic Scheduling.
Reidel, Dordrecht, 1982, .

[9] Malone, T.W., Fikes, R.E. and Howard, M.T.
Enterprise: A Market-like Task Scheduler for Distributed Computing Environments.
Working Paper, Cognitive and Instructional Sciences Group, Xerox Palo Alto Research Center, Palo Alto, California, October, 1983.

[10] Marzullo, Keith.
Maintaining the Time in a Distributed System - An Example of a Loosely-Couple Distributed Service.
PhD thesis, Stanford University, February, 1984.

[11] Mayr, E. W.
Well Structured Parallel Programs Are Not Easier to Schedule.
Report No. STAN-CS-81-880, Stanford University, September, 1981.

[12] Metcalfe, R.M., and Boggs, D.R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM (19, 7):395-404, July, 1976.

[13] Moon, David A.
Chaosnet
Symbolics, Inc., 1982.
Printed by permission of Massachusetts Institute of Technology.

[14] Moto-oka, Tohru.
Fifth Generation Computer Systems: A Japanese Project.
Computer :6-13, March, 1984.

[15] Rosenschein, Jeffrey S. and Singh, Vineet.
The Utility of Meta-level Effort.
Report No. HPP-83-20, Heuristic Programming Project, Stanford University, March, 1983.

[16] Shapiro, Ehud Y.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, ICOT, Japan, January, 1983.

[17] Shoch, John F. and Hupp, Jon A.
Measured Performance of an Ethernet Local Network.
Communications of the ACM 23(12):711-721, December, 1980.

[18] Simon, Herbert A., and Joseph B. Kadane.
Optimal Problem-Solving Search: All-or-None Solutions.
Artificial Intelligence 6:235-247, 1975.

[19] Ullman, Jeffrey D.
Some Thoughts about Supercomputer Organization.
In *Proceedings of COMPCON*, pages 424-432. IEEE Computer Society, IEEE Computer Society Press, February, 1984.

[20] Wilkes, M.V., and Wheeler, D.J.
The Cambridge Digital Communication Ring.
In *Proceedings of the Local Area Communications Network Symposium*, pages 47-61. MITRE Corporation, Bedford, Massachusetts, May, 1979.